

A Survey of Research and Practices of Network-on-Chip

TOBIAS BJERREGAARD AND SHANKAR MAHADEVAN

Technical University of Denmark

The scaling of microchip technologies has enabled large scale systems-on-chip (SoC). Network-on-chip (NoC) research addresses global communication in SoC, involving (i) a move from computation-centric to communication-centric design and (ii) the implementation of scalable communication structures. This survey presents a perspective on existing NoC research. We define the following abstractions: system, network adapter, network, and link to explain and structure the fundamental concepts. First, research relating to the actual network design is reviewed. Then system level design and modeling are discussed. We also evaluate performance analysis techniques. The research shows that NoC constitutes a unification of current trends of intrachip communication rather than an explicit new alternative.

Categories and Subject Descriptors: A.1 [**Introductory and Survey**]; B.4.3 [**Input/Output and Data-Communications**]: Interconnections; B.7.1 [**Integrated Circuits**]: Types and Design Styles; C.5.4 [**Computer System Implementation**]: VLSI Systems; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design; C.0 [**General**]: *System Architectures*

General Terms: Design

Additional Key Words and Phrases: Chip-area networks, communication-centric design, communication abstractions, GALS, GSI design, interconnects, network-on-chip, NoC, OCP, on-chip communication, SoC, sockets, system-on-chip, ULSI design

1. INTRODUCTION

Chip design has four distinct aspects: computation, memory, communication, and I/O. As processing power has increased and data intensive applications have emerged, the challenge of the communication aspect in single-chip systems, Systems-on-Chip (SoC), has attracted increasing attention. This survey treats a prominent concept for communication in SoC known as Network-on-Chip (NoC). As will become clear in the following, NoC does not constitute an explicit new alternative for intrachip communication but is rather a concept which presents a unification of on-chip communication solutions.

In this section, we will first briefly review the history of microchip technology that has led to a call for NoC-based designs. With our minds on intrachip communication,

This paper is a joint author effort, authors in alphabetical order.

S. Mahadevan was funded by SoC-MOBINET (IST-2000-30094), Nokia and the Thomas B. Thrigå Foundation.

Authors' address: Technical University of Denmark, Informatics and Mathematical Modelling, Richard Petersens Plads, Building 321, DK-2800 Lyngby, Denmark; email:{tob,sm}@imm.dtu.dk.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.

©2006 ACM 0360-0300/06/0300-ART1 \$5.00 http://doi.acm.org/10.1145/1132952.1132953

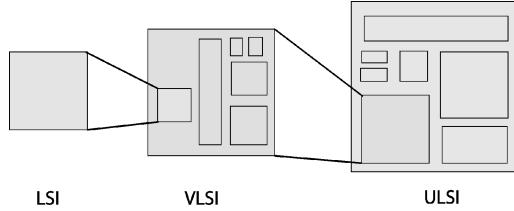


Fig. 1. When a technology matures, it leads to a paradigm shift in system scope. Shown here is the chip scope in LSI, VLSI, and ULSI, the sequence of technologies leading to the enabling of SoC designs.

we will then look at a number of key issues of large-scale chip design and finally show how the NoC concept provides a viable solution space to the problems presently faced by chip designers.

1.1. IntraSoC Communication

The scaling of microchip technologies has lead to a doubling of available processing resources on a single chip every second year. Even though this is projected to slow down to a doubling every three years in the next few years for fixed chip sizes [ITRS 2003], the exponential trend is still in force. Though the evolution is continuous, the system level focus, or system scope, moves in steps. When a technology matures for a given implementation style, it leads to a paradigm shift. Examples of such shifts are moving from room- to rack-level systems (LSI-1970s) and later from rack- to board-level systems (VLSI-1980s). Recent technological advances allowing multimillion transistor chips (currently well beyond 100M) have led to a similar paradigm shift from board- to chip-level systems (ULSI-1990s). The scope of a single chip has changed accordingly as illustrated in Figure 1. In LSI systems, a chip was a component of a system module (e.g., a bitslice in a bitslice processor), in VLSI systems, a chip was a system-level module (e.g., a processor or a memory), and in ULSI systems, a chip constitutes an entire system (hence the term System-on-Chip). SoC opens up the feasibility of a wide range of applications making use of massive parallel processing and tightly interdependent processes, some adhering to real-time requirements, bringing into focus new complex aspects of the underlying communication structure. Many of these aspects are addressed by NoC.

There are multiple ways to approach an understanding of NoC. Readers well versed in macronetwork theory may approach the concept by adapting proven techniques from multicomputer networks. Much work done in this area during the 80s and 90s can readily be built upon. Layered communication abstraction models and decoupling of computation and communication are relevant issues. There are, however, a number of basic differences between on- and off-chip communication. These generally reflect the difference in the cost ratio between wiring and processing resources.

Historically, computation has been expensive and communication cheap. With scaling microchip technologies, this changed. Computation is becoming ever cheaper, while communication encounters fundamental physical limitations such as time-of-flight of electrical signals, power use in driving long wires/cables, etc. In comparison with off-chip, on-chip communication is significantly cheaper. There is room for lots of wires on a chip. Thus the shift to single-chip systems has relaxed system communication problems. However on-chip wires do not scale in the same manner as transistors do, and, as we shall see in the following, the cost gap between computation and communication is

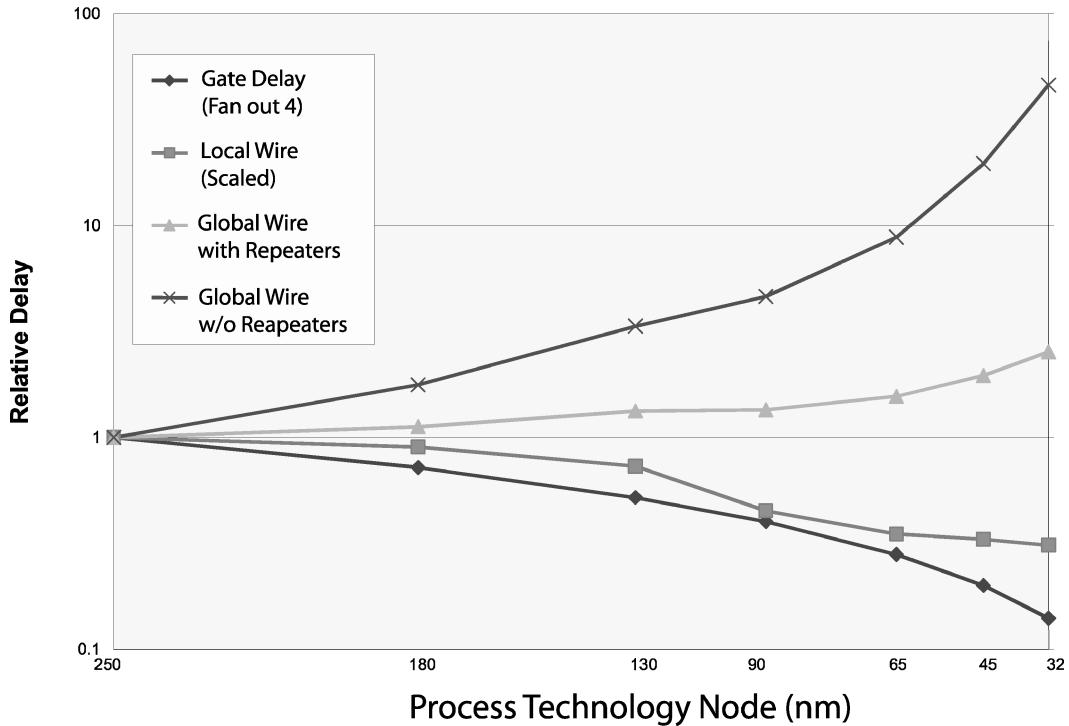


Fig. 2. Projected relative delay for local and global wires and for logic gates in technologies of the near future. [ITRS 2001].

widening. Meanwhile the differences between on- and off-chip wires make the direct scaling down of traditional multicomputer networks suboptimal for on-chip use.

In this survey, we attempt to incorporate the whole range of design abstractions while relating to the current trends of intrachip communication. With the Giga Transistor Chip era close at hand, the solution space of intrachip communication is far from trivial. We have summarized a number of relevant key issues. Though not new, we find it worthwhile to go through them as the NoC concept presents a possible unification of solutions for these. In Section 3 and 4, we will look into the details of research being done in relation to these issues, and their relevance for NoC.

—*Electrical wires.* Even though on-chip wires are cheap in comparison with off-chip wires, on-chip communication is becoming still more costly in terms of both power and speed. As fabrication technologies scale down, wire resistance per-mm is increasing while wire capacitance does not change much; the major part of the wire capacitance is due to edge capacitance [Ho et al. 2001]. For CMOS, the approximate point at which wire delays begin to dominate gate delays was the $0.25\text{ }\mu\text{m}$ generation for aluminum, and $0.18\text{ }\mu\text{m}$ for copper interconnects as first projected in SIA [1997]. Shrinking metal pitches, in order to maintain sufficient routing densities, is appropriate at the local level where wire lengths also decrease with scaling. But global wire lengths do not decrease, and, as local processing cycle times decrease, the time spent on global communication relative to the time spent on local processing increases drastically. Thus in future deep submicron (DSM) designs, the interconnect effect will definitely dominate performance [Sylvester and Keutzer 2000]. Figure 2, taken from the International Technology Roadmap for Semiconductors [ITRS 2001], shows the

projected relative delay for local wires, global wires, and logic gates in the near future. Another issue of pressing importance concerns signal integrity. In DSM technologies, the wire models are unreliable due to issues like fabrication uncertainties, crosstalk, noise sensitivity etc. These issues are especially applicable to long wires.

Due to these effects of scaling, it has become necessary to differentiate between local and global communication, and, as transistors shrink, the gap is increasing. The need for global communication schemes supporting single-chip systems has emerged.

—*System synchronization.* As chip technologies scale and chip speeds increase, it is becoming harder to achieve global synchronization. The drawbacks of the predominant design style of digital integrated circuits, that is, strict global synchrony, are growing relative to the advantages. The clocktree needed to implement a globally synchronized clock is demanding increasing portions of the power and area budget, and, even so, the clock skew is claiming an ever larger relative part of the total cycle time available [Oklobdzija and Sparsø 2002; Oberg 2003]. This has triggered work on skew-tolerant circuit design [Nedovic et al. 2003], which deals with clockskew by relaxing the need for timing margins, and on the use of optical waveguides for on-chip clock distribution [Piguet et al. 2004], for the main purpose of minimizing power usage. Still, power hungry skew adjustment techniques such as phase locked loops (PLL) and delay locked loops (DLL), traditionally used for chip-to-chip synchronization, are finding their way into single-chip systems [Kurd et al. 2001; Xanthopoulos et al. 2001].

As a reaction to the inherent limitations of global synchrony, alternative concepts such as GALS (Globally Asynchronous Locally Synchronous systems) are being introduced. A GALS chip is made up of locally synchronous islands which communicate asynchronously [Chapiro 1984; Meincke et al. 1999; Muttersbach et al. 2000]. There are two main advantageous aspects of this method. One is the reducing of the synchronization problem to a number of smaller subproblems. The other relates to the integration of different IP (Intellectual Property) cores, easing the building of larger systems from individual blocks with different timing characteristics.

—*Design productivity.* The exploding amount of processing resources available in chip design together with a requirement for shortened design cycles have pushed the productivity burden on to chip designers. Between 1997 and 2002, the market demand reduced the typical design cycle by 50%. As a result of increased chip sizes, shrinking geometries, and the availability of more metal layers, the design complexity increased 50 times in the same period [OCPIP 2003a]. To keep up with these requirements, IP reuse is pertinent. A new paradigm for design methodology is needed which allows the design effort to scale linearly with system complexity.

Abstraction at the register transfer level (RTL) was introduced with the ASIC design flow during the 90s, allowing synthesized standard cell design. This made it possible to design large chips within short design cycles, and synthesized RTL design is, at present, the defacto standard for making large chips quickly. But the availability of on-chip resources is outgrowing the productivity potential of even the ASIC design style. In order to utilize the exponential growth in number of transistors on each chip, even higher levels of abstraction must be applied. This can be done by introducing higher level communication abstractions, making a layered design methodology that enables a partitioning of the design effort into minimally interdependent subtasks. Support for this at the hardware level includes standard communication sockets which allow IP cores from different vendors to be plugged effortlessly together. This is particularly pertinent in complex multiprocessor system-on-chip (MPSoC) designs. Also, the development of design techniques to further increase the productivity of designers, is important. Electronic system level (ESL) design tools are necessary for supporting a design flow which make efficient use of such communication abstraction

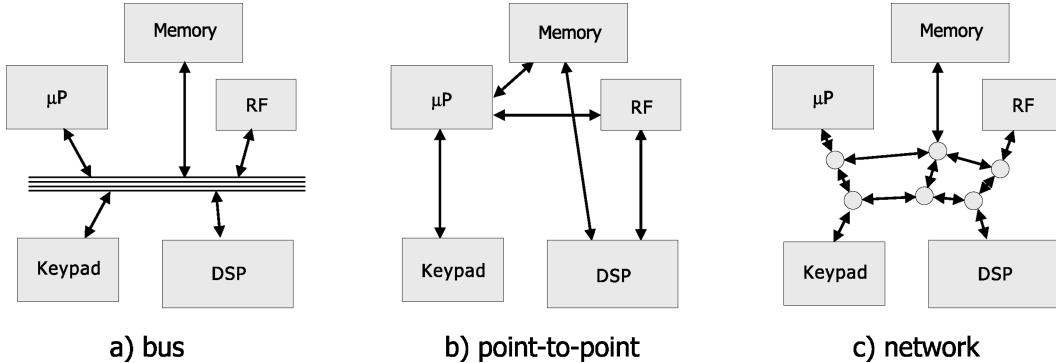


Fig. 3. Examples of communication structures in Systems-on-Chip. a) traditional bus-based communication, b) dedicated point-to-point links, c) a chip area network.

and design automation techniques and which make for seamless iterations across all abstraction levels. Pertaining to this, the complex, dynamic interdependency of data streams—arising when using a shared media for data traffic—threatens to foil the efforts of obtaining minimal interdependence between IP cores. Without special quality-of-service (QoS) support, the performance of data communication may become unwarrantably arbitrary [Goossens et al. 2005].

To ensure the effective exploitation of technology scaling, intelligent use of the available chip design resources is necessary at the physical as well as at the logical design level. The means to achieve this are through the development of effective and structured design methods and ESL tools.

As shown, the major driving factors for the development of global communication schemes are the ever increasing density of on-chip resources and the drive to utilize these resources with a minimum of effort as well as the need to counteract the physical effects of DSM technologies. The trend is towards a subdivision of processing resources into manageable pieces. This helps reduce design cycle time since the entire chip design process can be divided into minimally interdependent subproblems. This also allows the use of modular verification methodologies, that is, verification at a low abstraction level of cores (and communication network) individually and at a high abstraction level of the system as a whole. Working at a high abstraction level allows a great degree of freedom from lower level issues. It also tends towards a differentiation of local and global communication. As intercore communication is becoming the performance bottleneck in many multicore applications, the shift in design focus is from a traditional processing-centric to a communication-centric one. One top-level aspect of this involves the possibility to save on global communication resources at the application level by introducing communication aware optimization algorithms in compilers [Guo et al. 2000]. System-level effects of technology scaling are further discussed in Catthoor et al. [2004].

A standardized global communication scheme, together with standard communication sockets for IP cores, would make Lego brick-like plug-and-play design styles possible, allowing good use of the available resources and fast product design cycles.

1.2. NoC in SoC

Figure 3 shows some examples of basic communication structures in a sample SoC, for example, a mobile phone. Since the introduction of the SoC concept in the 90s, the solutions for SoC communication structures have generally been characterized by custom designed ad hoc mixes of buses and point-to-point links [Lahiri et al. 2001]. The

Table I. Bus-versus-Network Arguments (Adapted from Guerrier and Greiner [2000])

Bus Pros & Cons	Network Pros & Cons		
Every unit attached adds parasitic capacitance, therefore electrical performance degrades with growth.	-	+	Only point-to-point one-way wires are used, for all network sizes, thus local performance is not degraded when scaling.
Bus timing is difficult in a deep submicron process.	-	+	Network wires can be pipelined because links are point-to-point.
Bus arbitration can become a bottleneck. The arbitration delay grows with the number of masters.	-	+	Routing decisions are distributed, if the network protocol is made non-central.
The bus arbiter is instance-specific.	-	+	The same router may be reinstated, for all network sizes.
Bus testability is problematic and slow.	-	+	Locally placed dedicated BIST is fast and offers good test coverage.
Bandwidth is limited and shared by all units attached.	-	+	Aggregated bandwidth scales with the network size.
Bus latency is wire-speed once arbiter has granted control.	+	-	Internal network contention may cause a latency.
Any bus is almost directly compatible with most available IPs, including software running on CPUs.	+	-	Bus-oriented IPs need smart wrappers. Software needs clean synchronization in multiprocessor systems.
The concepts are simple and well understood.	+	-	System designers need reeducation for new concepts.

bus builds on well understood concepts and is easy to model. In a highly interconnected multicore system, however, it can quickly become a communication bottleneck. As more units are added to it, the power usage per communication event grows as well due to more attached units leading to higher capacitive load. For multimaster busses, the problem of arbitration is also not trivial. Table I summarizes the pros and cons of buses and networks. A crossbar overcomes some of the limitations of the buses. However, it is not ultimately scalable and, as such, it is an intermediate solution. Dedicated point-to-point links are optimal in terms of bandwidth availability, latency, and power usage as they are designed especially for this given purpose. Also, they are simple to design and verify and easy to model. But the number of links needed increases exponentially as the number of cores increases. Thus an area and possibly a routing problem develops.

From the point of view of design-effort, one may argue that, in small systems of less than 20 cores, an ad hoc communication structure is viable. But, as the systems grow and the design cycle time requirements decrease, the need for more generalized solutions becomes pressing. For maximum flexibility and scalability, it is generally accepted that a move towards a shared, segmented global communication structure is needed. This notion translates into a data-routing network consisting of communication links and routing nodes that are implemented on the chip. In contrast to traditional SoC communication methods outlined previously, such a distributed communication media scales well with chip size and complexity. Additional advantages include increased aggregated performance by exploiting parallel operation.

From a technological perspective, a similar solution is reached: in DSM chips, long wires must be segmented in order to avoid signal degradation, and busses are implemented as multiplexed structures in order to reduce power and increase responsiveness. Hierarchical bus structures are also common as a means to adhere to the given communication requirements. The next natural step is to increase throughput by pipelining

these structures. Wires become pipelines and bus-bridges become routing nodes. Expanding on a structure using these elements, one gets a simple network.

A common concept for segmented SoC communication structures is based on networks. This is what is known as Network-on-Chip (NoC) [Agarwal 1999; Guerrier and Greiner 2000; Dally and Towles 2001; Benini and Micheli 2002; Jantsch and Tenhunen 2003]. As presented previously, the distinction between different communication solutions is fading. NoC is seen to be a unifying concept rather than an explicit new alternative. In the research community, there are two widely held perceptions of NoC: (i) that NoC is a subset of SoC, and (ii) that NoC is an extension of SoC. In the first view, NoC is defined strictly as the data-forwarding communication fabric, that is, the network and methods used in accessing the network. In the second view NoC is defined more broadly to also encompass issues dealing with the application, system architecture, and its impact on communication or vice versa.

1.3. Outline

The purpose of this survey is to clarify the NoC concept and to map the scientific efforts made into the area of NoC research. We will identify general trends and explain a range of issues which are important for state-of-the-art global chip-level communication. In doing so, we primarily take the first view of NoC, that is, that it is a subset of SoC, to focus and structure the diverse discussion. From our perspective, the view of NoC as an extension of SoC muddles the discussion with topics common to any large-scale IC design effort such as partitioning and mapping application, hardware/software codesign, compiler choice, etc.

The rest of the survey is organized as follows. In Section 2, we will discuss the basics of NoC. We will give a simple NoC example, address some relevant system-level architectural issues, and relate the basic building blocks of NoC to abstract network layers and research areas. In Section 3, we will go into more details of existing NoC research. This section is partitioned according to the research areas defined in Section 2. In Section 4, we discuss high abstraction-level issues such as design space exploration and modeling. These are issues often applicable to NoC only in the view of it as an extension of SoC, but we treat specifically issues of relevance to NoC-based designs and not to large scale IC designs in general. In Section 5, performance analysis is addressed. Section 6 presents a set of case studies describing a number of specific NoC implementations, and Section 7 summarizes the survey.

2. NOC BASICS

In this section, the basics of NoC are uncovered. First a component-based view will be presented, introducing the basic building blocks of a typical NoC. Then we will look at system-level architectural issues relevant to NoC-based SoC designs. After this, a layered abstraction-based view will be presented, looking at network abstraction models, in particular, OSI and the adaption of such for NoC. Using the foundations established in this section, we will go into further details of specific NoC research in Section 3.

2.1. A Simple NoC Example

Figure 4 shows a sample NoC structured as a 4-by-4 grid which provides global chip-level communication. Instead of busses and dedicated point-to-point links, a more general scheme is adapted, employing a grid of routing nodes spread out across the chip, connected by communication links. For now, we will adapt a simplified perspective in

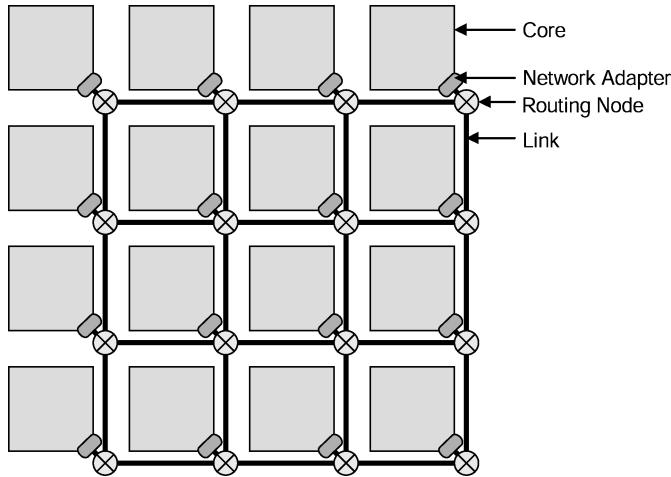


Fig. 4. Topological illustration of a 4-by-4 grid structured NoC, indicating the fundamental components.

which the NoC contains the following fundamental components.

- Network adapters* implement the interface by which *cores* (IP blocks) connect to the NoC. Their function is to decouple computation (the cores) from communication (the network).
- Routing nodes* route the data according to chosen protocols. They implement the routing strategy.
- Links* connect the nodes, providing the raw bandwidth. They may consist of one or more logical or physical channels.

Figure 4 covers only the topological aspects of the NoC. The NoC in the figure could thus employ packet or circuit switching or something entirely different and be implemented using asynchronous, synchronous, or other logic. In Section 3, we will go into details of specific issues with an impact on the network performance.

2.2. Architectural Issues

The diversity of communication in the network is affected by architectural issues such as system composition and clustering. These are general properties of SoC but, since they have direct influence on the design of the system-level communication infrastructure, we find it worthwhile to go through them here.

Figure 5 illustrates how system composition can be categorized along the axes of *homogeneity* and *granularity* of system cores. The figure also clarifies a basic difference between NoC and networks for more traditional parallel computers; the latter have generally been homogeneous and coarse grained, whereas NoC-based systems implement a much higher degree of variety in composition and in traffic diversity.

Clustering deals with the localization of portions of the system. Such localization may be logical or physical. Logical clustering can be a valuable programming tool. It can be supported by the implementation of hardware primitives in the network, for example, flexible addressing schemes or virtual connections. Physical clustering, based on preexisting knowledge of traffic patterns in the system, can be used to minimize global communication, thereby minimizing the total cost of communicating, power and performancewise.

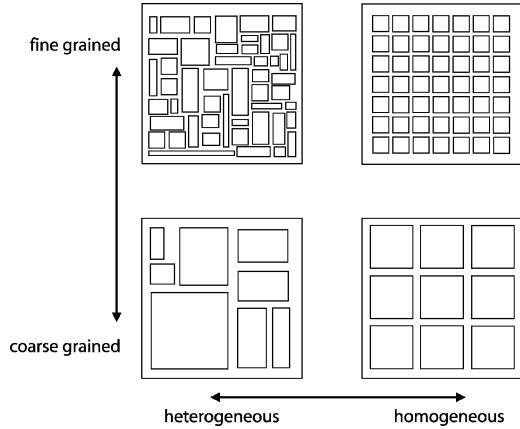


Fig. 5. System composition categorized along the axes of homogeneity and granularity of system components.

Generally speaking, *reconfigurability* deals with the ability to allocate available resources for specific purposes. In relation to NoC-based systems, reconfigurability concerns how the NoC, a flexible communication structure, can be used to make the system reconfigurable from an application point of view. A configuration can be established for example, by programming connections into the NoC. This resembles the reconfigurability of an FPGA, though NoC-based reconfigurability is most often of coarser granularity. In NoC, the reconfigurable resources are the routing nodes and links rather than wires.

Much research work has been done on architecturally-oriented projects in relation to NoC-based systems. The main issue in architectural decisions is the balancing of flexibility, performance, and hardware costs of the system as a whole. As the underlying technology advances, the trade-off spectrum is continually shifted, and the viability of the NoC concept has opened up to a communication-centric solution space which is what current system-level research explores.

At one corner of the architectural space outlined in Figure 5, is the Pleiades architecture [Zhang et al. 2000] and its instantiation, the Maia processor. A microprocessor is combined with a relatively fine-grained heterogeneous collection of ALUs, memories, FPGAs, etc. An interconnection network allows arbitrary communication between modules of the system. The network is hierarchical and employs clustering in order to provide the required communication flexibility while maintaining good energy-efficiency.

At the opposite corner are a number of works, implementing homogeneous coarse-grained multiprocessors. In Smart Memories [Mai et al. 2000], a hierarchical network is used with physical clustering of four processors. The flexibility of the local cluster network is used as a means for reconfigurability, and the effectiveness of the platform is demonstrated by mimicking two machines on far ends of the architectural spectrum, the Imagine streaming processor and Hydra multiprocessor, with modest performance degradation. The global NoC is not described, however. In the RAW architecture [Taylor et al. 2002], on the other hand, the NoC which interconnects the processor tiles is described in detail. It consists of a static network, in which the communication is preprogrammed cycle-by-cycle, and a dynamic network. The reason for implementing two physically separate networks is to accommodate different types of traffic in general purpose systems (see Section 4.3 concerning traffic characterization). The Eclipse [Forsell 2002] is another similarly distributed multiprocessor architecture

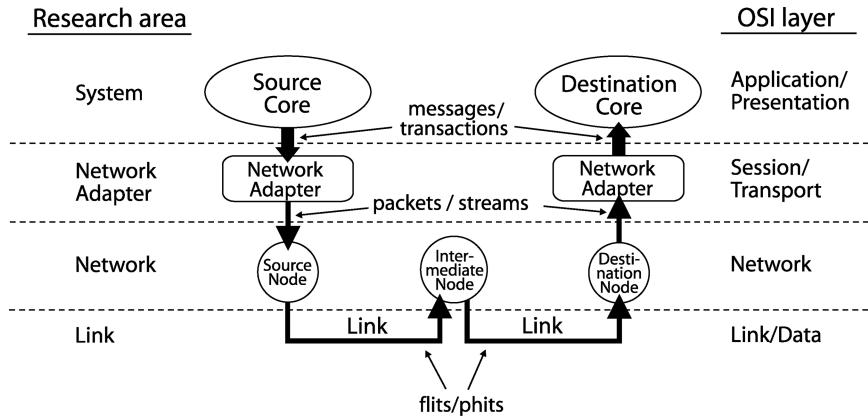


Fig. 6. The flow of data from source to sink through the NoC components with an indication of the types of datagrams and research area.

in which the interconnection network plays an important role. Here, the NoC is a key element in supporting a sofisticated parallel programming model.

2.3. Network Abstraction

The term NoC is used in research today in a very broad sense ranging from gate-level physical implementation, across system layout aspects and applications, to design methodologies and tools. A major reason for the widespread adaptation of network terminology lies in the readily available and widely accepted abstraction models for networked communication. The OSI model of layered network communication can easily be adapted for NoC usage as done in Benini and Micheli [2001] and Arteris [2005]. In the following, we will look at network abstraction, and make some definitions to be used later in the survey.

To better understand the approaches of different groups involved in NoC, we have partitioned the spectrum of NoC research into four areas: 1) system, 2) network adapter, 3) network and 4) link research. Figure 6 shows the flow of data through the network, indicating the relation between these research areas, the fundamental components of NoC, and the OSI layers. Also indicated is the basic datagram terminology.

The *system* encompasses applications (processes) and architecture (cores and network). At this level, most of the network implementation details may still be hidden. Much research done at this level is applicable to large scale SoC design in general. The *network adapter* (NA) decouples the cores from the network. It handles the end-to-end flow control, encapsulating the *messages* or *transactions* generated by the cores for the routing strategy of the Network. These are broken into *packets* which contain information about their destination, or connection-oriented *streams* which do not, but have had a path setup prior to transmission. The NA is the first level which is network aware. The *network* consists of the routing nodes, links, etc, defining the topology and implementing the protocol and the node-to-node flow control. The lowest level is the *link* level. At this level, the basic datagram are *flits* (flow control units), node level atomic units from which packets and streams are made up. Some researchers operate with yet another subdivision, namely *phits* (physical units), which are the minimum size datagram that can be transmitted in one link transaction. Most commonly flits and phits are equivalent, though in a network employing highly serialized links, each flit could be made up of a sequence of phits. Link-level research deals mostly with

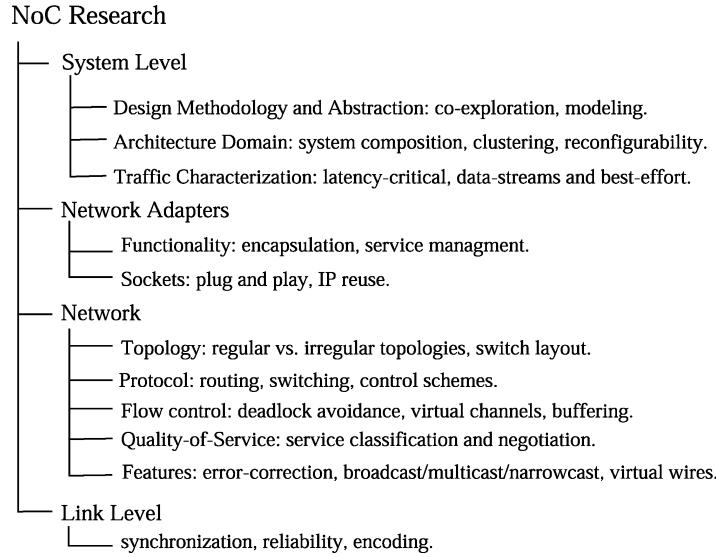


Fig. 7. NoC research area classification. This classification, which also forms the structure of Section 3, is meant as a guideline to evaluate NoC research and not as a technical categorization.

encoding and synchronization issues. The presented datagram terminology seems to be generally accepted, though no standard exists.

In a NoC, the layers are generally more closely bound than in a macronetwork. Issues arising often have a more physically-related flavor even at the higher abstraction levels. OSI specifies a protocol stack for multicomputer networks. Its aim is to shield higher levels of the network from issues of lower levels in order to allow communication between independently developed systems, for example, of different manufacturers, and to allow ongoing expansion of systems. In comparison with macronetworks, NoC benefits from the system composition being completely static. The network can be designed based on knowledge of the cores to be connected and also possibly on knowledge of the characteristics of the traffic to be handled, as demonstrated in for example, Bolotin et al. [2004] and Goossens et al. [2005]. Awareness of lower levels can be beneficial as it can lead to higher performance. The OSI layers, which are defined mainly on the basis of pure abstraction of communication protocols, thus cannot be directly translated into the research areas defined here. With this in mind, the relation established in Figure 6 is to be taken as a conceptual guideline.

3. NOC RESEARCH

In this section, we provide a review of the approaches of various research groups. Figure 7 illustrates a simplified classification of this research. The text is structured based on the layers defined in Section 2.3. Since we consider NoC as a subset of SoC, system-level research is dealt with separately in Section 4.

3.1. Network Adapter

The purpose of the network adapter (NA) is to interface the core to the network and make communication services transparently available with a minimum of effort from

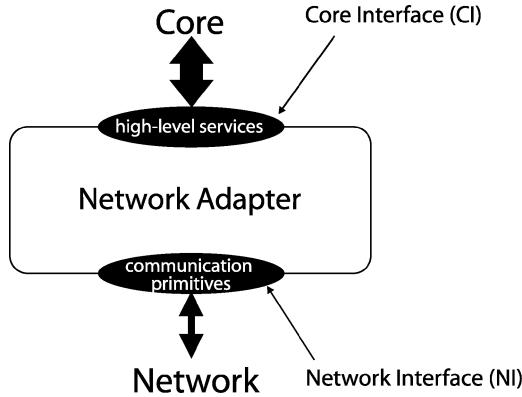


Fig. 8. The network adapter (NA) implements two interfaces, the core interface (CI) and the network interface (NI).

the core. At this point, the boundary between computation and communication is specified.

As illustrated in Figure 8, the NA component implements a core interface (CI) at the core side and a network interface (NI) at the network side. The function of the NA is to provide high-level communication services to the core by utilizing primitive services provided by the network hardware. Thus the NA *decouples* the core from the network, implementing the network end-to-end flow control, facilitating a layered system design approach. The level of decoupling may vary. A high level of decoupling allows for easy reuse of cores. This makes possible a utilization of the exploding resources available to chip designers, and greater design productivity is achieved. On the other hand, a lower level of decoupling (a more network aware core) has the potential to make more optimal use of the network resources.

In this section, we first address the use of standard sockets. We then discuss the abstract functionality of the NA. Finally, we talk about some actual NA implementations which also address issues related to timing and synchronization.

3.1.1. Sockets. The CI of the NA may be implemented to adhere to a SoC socket standard. The purpose of a socket is to orthogonalize computation and communication. Ideally a socket should be completely NoC implementation agnostic. This will facilitate the greatest degree of reusability because the core adheres to the specification of the socket alone, independently of the underlying network hardware. One commonly used socket is the *Open Core Protocol* (OCP) [OCPIP 2003b; Haverinen et al. 2002]. The OCP specification defines a flexible family of memory-mapped, core-centric protocols for use as a native core interface in on-chip systems. The three primary properties envisioned in OCP include (i) architecture independent design reuse, (ii) feature-specific socket implementation, and (iii) simplification of system verification and testing. OCP addresses not only dataflow signaling, but also uses related to errors, interrupts, flags and software flow control, control and status, and test. Another proposed standard is the *Virtual Component Interface* (VCI) [VSI Alliance 2000] used in the SPIN [Guerrier and Greiner 2000] and Proteo [Siguenza-Tortosa et al. 2004] NoCs. In Radulescu et al. [2004], support for the Advanced eXtensible Interface (AXI) [ARM 2004] and Device Transaction Level (DTL) [Philips Semiconductors 2002] protocols was also implemented in an NA design.

3.1.2. NA Services. Basically, the NA provides *encapsulation* of the traffic for the underlying communication media and *management* of services provided by the network. Encapsulation involves handling of end-to-end flow control in the network. This may include global addressing and routing tasks, reorder buffering and data acknowledgement, buffer management to prevent network congestion, for example, based on credit, packet creation in a packet-switched network, etc.

Cores will contend for network resources. These may be provided in terms of service quantification, for example, bandwidth and/or latency guarantees (see also Sections 3.2.4 and 5). Service management concerns setting up circuits in a circuit-switched network, bookkeeping tasks such as keeping track of connections, and matching responses to requests. Another task of the NA could be to negotiate the service needs between the core and the network.

3.1.3. NA Implementations. A clear understanding of the role of the NA is essential to successful NoC design. Muttersbach et al. [2000] address synchronization issues, proposing a design of an asynchronous wrapper for use in a practical GALS design. Here the synchronous modules are equipped with asynchronous wrappers which adapt their interfaces to the self-timed environment. The packetization occurs within the synchronous module. The wrappers are assembled from a concise library of predesigned technology-independent elements and provide high speed data transfer. Another mixed asynchronous/synchronous NA architecture is proposed in Bjerregaard et al. [2005]. Here, a synchronous OCP interface connects to an asynchronous, message-passing NoC. Packetization is performed in the synchronous domain, while sequencing of flits is done in the asynchronous domain. This makes the sequencing independent of the speed of the OCP interface, while still taking advantage of synthesized synchronous design for maintaining a flexible packet format. Thus the NA leverages the advantages particular to either circuit design style. In Radulescu et al. [2004], a complete NA design for the ÆTHEREAL NoC is presented which also offers a shared-memory abstraction to the cores. It provides compatibility to existing on-chip protocols such as AXI, DTL, and OCP and allows easy extension to other future protocols as well.

However, the cost of using standard sockets is not trivial. As demonstrated in the HERMES NoC [Ost et al. 2005], the introduction of OCP makes the transactions up to 50% slower compared to the native core interface. An interesting design trade-off issue is the partitioning of the NA functions between software (possibly in the core) and hardware (most often in the NA). In Bhojwani and Mahapatra [2003], a comparison of software and hardware implementations of the packetization task was undertaken, the software taking 47 cycles to complete, while the hardware version took only 2 cycles. In Radulescu et al. [2004], a hardware implementation of the entire NA introduces a latency overhead of between 4 and 10 cycles, pipelined to maximize throughput. The NA in Bjerregaard et al. [2005] takes advantage of the low forward latency of clockless circuit techniques, introducing an end-to-end latency overhead of only 3 to 5 cycles for writes and 6 to 8 cycles for reads which include data return.

3.2. Network Level

The job of the network is to deliver messages from their source to their designated destination. This is done by providing the hardware support for basic communication primitives. A well-built network, as noted by Dally and Towles [2001], should appear as a logical wire to its clients. An on-chip network is defined mainly by its topology and the protocol implemented by it. Topology concerns the layout and connectivity of the nodes and links on the chip. Protocol dictates how these nodes and links are used.

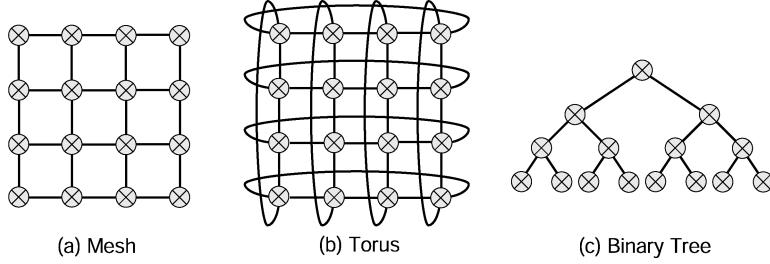


Fig. 9. Regular forms of topologies scale predictably with regard to area and power. Examples are (a) 4-ary 2-cube mesh, (b) 4-ary 2-cube torus and (c) binary (2-ary) tree.

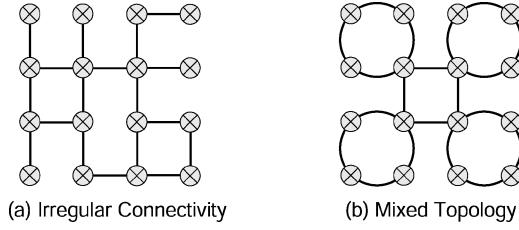


Fig. 10. Irregular forms of topologies are derived by altering the connectivity of a regular structure such as shown in (a) where certain links from a mesh have been removed or by mixing different topologies such as in (b) where a ring coexists with a mesh.

3.2.1. Topology. One simple way to distinguish different regular topologies is in terms of k -ary n -cube (grid-type), where k is the degree of each dimension and n is the number of dimensions (Figure 9), first described by Dally [1990] for multicomputer networks. The k -ary tree and the k -ary n -dimensional fat tree are two alternate regular forms of networks explored for NoC. The network area and power consumption scales predictably for increasing size of regular forms of topology. Most NoCs implement regular forms of network topology that can be laid out on a chip surface (a 2-dimensional plane) for example, k -ary 2-cube, commonly known as grid-based topologies. The Octagon NoC demonstrated in Karim et al. [2001, 2002] is an example of a novel regular NoC topology. Its basic configuration is a ring of 8 nodes connected by 12 bidirectional links which provides two-hop communication between any pair of nodes in the ring and a simple, shortest-path routing algorithm. Such rings are then connected edge-to-edge to form a larger, scalable network. For more complex structures such as trees, finding the optimal layout is a challenge on its own right.

Besides the form, the nature of links adds an additional aspect to the topology. In k -ary 2-cube networks, popular NoC topologies based on the nature of link are the mesh which uses bidirectional links and torus which uses unidirectional links. For a torus, a folding can be employed to reduce long wires. In the NOSTRUM NoC presented in Millberg et al. [2004], a folded torus is discarded in favor of a mesh with the argument that it has longer delays between routing nodes. Figure 9 shows examples of regular forms of topology. Generally, mesh topology makes better use of links (utilization), while tree-based topologies are useful for exploiting locality of traffic.

Irregular forms of topologies are derived by mixing different forms in a hierarchical, hybrid, or asymmetric fashion as seen in Figure 10. Irregular forms of topologies scale

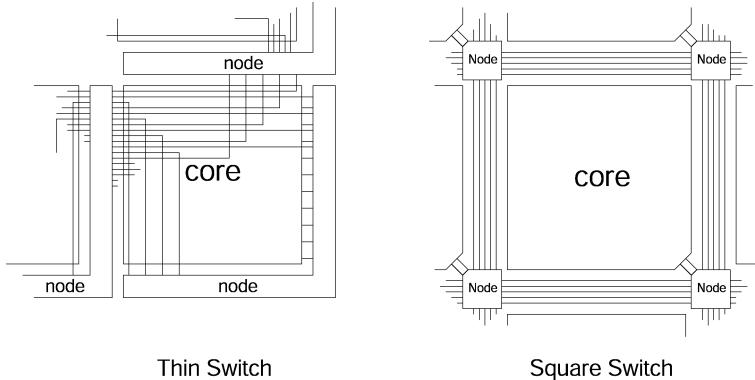


Fig. 11. Two layout concepts. The thin switch is distributed around the cores, and wires are routed across it. The square switch is placed on the crossings in dedicated channels between the cores.

nonlinearly with regards to area and power. These are usually based on the concept of clustering. A small private/local network often implemented as a bus [Mai et al. 2000; Wielage and Goossens 2002] for local communication with k-ary 2-cube global communication is a favored solution. In Pande et al. [2005], the impact of clustering on five NoC topologies is presented. It shows 20% to 40% reduction in bit-energy for the same amount of throughput due to traffic localization.

With regard to the presence of a local traffic source or sink connected to the node, *direct networks* are those that have at least one core attached to each node; *indirect networks*, on the other hand, have a subset of nodes not connected to any core, performing only network operations as is generally seen in tree-based topology where cores are connected at the leaf nodes. The examples of indirect tree-based networks are fat-tree in SPIN [Guerrier and Greiner 2000] and butterfly in Pande et al. [2003]. The fat-tree used in SPIN is proven in Leiserson [1985] to be most hardware efficient compared to any other network.

For alternate classifications of topology, the reader is referred to Aggarwal and Franklin [2002], Jantsch [2003], and Culler et al. [1998]. Culler et al. [1998] combine protocol and geometry to bring out a new type of classification which is defined as topology.

With regards to the routing nodes, a layout trade-off is the *thin switch* vs *square switch* presented by Kumar et al. [2002]. Figure 11 illustrates the difference between these two layout concepts. A thin switch is distributed around the cores, and wires are routed across them. A square switch is placed on the crossings of dedicated wiring channels between the cores. It was found that the square switch is better for performance and bandwidth, while the thin switch requires relatively low area. The area overhead required to implement a NoC is in any case expected to be modest. The processing logic of the router for a packet switched network is estimated to be approximately between 2.0% [Pande et al. 2003] to 6.6% [Dally and Towles 2001] of the total chip area. In addition to this, the wiring uses a portion of the upper two wiring layers.

3.2.2. Protocol. The protocol concerns the strategy of moving data through the NoC. We define switching as the mere transport of data, while routing is the intelligence behind it, that is, it determines the path of the data transport. This is in accordance with Culler et al. [1998]. In the following, these and other aspects of protocol commonly

addressed in NoC research, are discussed.

- Circuit vs packet switching.* Circuit switching involves the circuit from source to destination that is setup and reserved until the transport of data is complete. Packet switched traffic, on the other hand, is forwarded on a per-hop basis, each packet containing routing information as well as data.
- Connection-oriented vs connectionless.* Connection-oriented mechanisms involve a dedicated (logical) connection path established prior to data transport. The connection is then terminated upon completion of communication. In connectionless mechanisms, the communication occurs in a dynamic manner with no prior arrangement between the sender and the receiver. Thus circuit switched communication is always connection-oriented, whereas packet switched communication may be either connection-oriented or connectionless.
- Deterministic vs adaptive routing.* In a deterministic routing strategy, the traversal path is determined by its source and destination alone. Popular deterministic routing schemes for NoC are source routing and X-Y routing (2D dimension order routing). In source routing, the source core specifies the route to the destination. In X-Y routing, the packet follows the rows first, then moves along the columns toward the destination or vice versa. In an adaptive routing strategy, the routing path is decided on a per-hop basis. Adaptive schemes involve dynamic arbitration mechanisms, for example, based on local link congestion. This results in more complex node implementations but offers benefits like dynamic load balancing.
- Minimal vs nonminimal routing.* A routing algorithm is minimal if it always chooses among shortest paths toward the destination; otherwise it is nonminimal.
- Delay vs loss.* In the delay model, datagrams are never dropped. This means that the worst that can happen is that the data is delayed. In the loss model, datagrams can be dropped. In this case, the data needs to be retransmitted. The loss model introduces some overhead in that the state of the transmission, successful or failed, must somehow be communicated back to the source. There are, however, some advantages involved in dropping datagrams, for example, as a means of resolving network congestion.
- Central vs distributed control.* In centralized control mechanisms, routing decisions are made globally, for example, bus arbitration. In distributed control, most common for segmented interconnection networks, the routing decisions are made locally.

The protocol defines the use of the available resources, and thus the node implementation reflects design choices based on the listed terms. In Figure 12, taken from Duato et al. [2003], the authors have clearly identified the major components of any routing node that is, buffers, switch, routing and arbitration unit, and link controller. The switch connects the input buffers to the output buffers, while the routing and arbitration unit implements the algorithm that dictates these connections. In a centrally controlled system, the routing control would be common for all nodes, and a strategy might be chosen which guarantees no traffic contention. Thus no arbitration unit would be necessary. Such a scheme can be employed in a NoC in which all nodes have a common sense of time as presented in Millberg et al. [2004]. Here the NOSTRUM NoC implements an explicit time division multiplexing mechanism which the authors call *Temporally Disjoint Networks* (TDN). Packets cannot collide if they are in different TDNs. This is similar to the slot allocation mechanism in the *ÆTHEREAL* NoC [Goossens et al. 2005].

The optimal design of the switching fabric itself relates to the services offered by the router. In Kim et al. [2005], a crossbar switch is proposed which offers adaptive bandwidth control. This is facilitated by adding an additional bus, allowing the crossbar to be bypassed during periods of congestion. Thus, the switch is shown to improve the

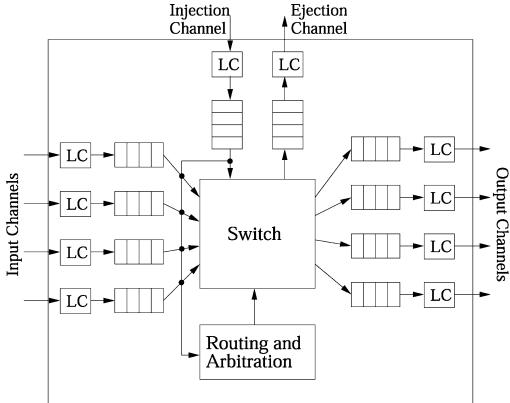


Fig. 12. Generic router model. LC = link controller (reprinted from Duato et al. [2003] by Jose Duato, Sudhakar Yalamanchili and Lionel Ni, Fig. 2.1, ©2003, with permission from Elsevier).

throughput and latency of the router by up to 27% and 41%, respectively, at a modest area and power overhead of 21% and 15%, respectively. In Bjerregaard and Sparsø [2005a], on the other hand, a nonblocking switch is proposed which allows for hard performance guarantees when switching connections within the router (more details in Section 3.2.4). By utilizing the knowledge that, only a limited number of flits can enter the router through each input port, the switch can be made to scale linearly rather than exponentially with the number of connections on each port. In Leroy et al. [2005], a switch similarly provides guaranteed services. This switch, however, switches individual wires on each port rather than virtual connections.

A quantitative comparison of connection-oriented and connectionless schemes for an MPEG-2 Video Decoder is presented in Harmanci et al. [2005]. The connection-oriented scheme is based on *ÆTHEREAL*, while the connectionless scheme is based on DiffServ—a priority-based packet scheduling NoC. The conclusions of tests, conducted in the presence of background traffic noise, show that (i) the individual end-to-end delay is lower in connectionless than in connection-oriented scheme due to better adaptation of the first approach to variable bit-rates of the MPEG video flows, and (ii) the connectionless schemes present a higher stability towards a wrong decision in the type of service to be assigned to a flow.

Concerning the merits of adaptive routing versus deterministic, there are different opinions. In Neeb et al. [2005], a comparison of deterministic (dimension-order) and adaptive (negative-first and planar-adaptive) routing applied to mesh, torus, and cube networks, was made. For chips performing interleaving in high throughput channel decoder wireless applications, the dimension-order routing scheme was found to be inferior compared to adaptive schemes when using lower dimension NoCs topologies. However, it was shown to be the best choice, due to low area and high throughput characteristics, for higher dimension NoC topologies. The impact on area and throughput of input and output buffer queues in the router, was also discussed. In de Mello et al. [2004], the performance of minimal routing protocols in the HERMES [Moraes et al. 2004] NoC were investigated: one deterministic protocol (XY-routing) and three partially adaptive protocols (west-first, north-last and negative-first routing). While the adaptive protocols can potentially speed up the delivery of individual packets, it was shown that the deterministic protocol was superior to the adaptive ones from a global

Table II. Cost and Stalling for Different Routing Protocols

Protocol	Per router cost		Stalling
	Latency	Buffering	
store-and-forward	packet	packet	at two nodes and the link between them
wormhole	header	header	at all nodes and links spanned by the packet
virtual cut-through	header	packet	at the local node

point. The reason is that adaptive protocols tend to concentrate the traffic in the center of the network, resulting in increased congestion there.

The wide majority of NoC research is based on packet switching networks. In addition, most are delay-based since the overhead of keeping account of packets being transmitted and of retransmitting dropped packets is high. In Gaughan et al. [1996], however, a routing scheme is presented which accommodates dropping packets when errors are detected. Most often connectionless routing is employed for best effort (BE) traffic (Section 4.3), while connection-oriented routing is used to provide service guarantees (Section 3.2.4). In SoCBUS [Sathe et al. 2003], a different approach is taken in that a connection-oriented strategy is used to provide BE traffic routing. Very simple routers establish short-lived connections set up using BE routed control packets which provide a very high throughput of 1.2GHz in a 0.18 μ m CMOS process. Drawbacks are the time spent during the setup phase, which requires a path acknowledge, and the fact that only a single connection can be active on each link at any given time. A similarly connection-oriented NoC is aSoC [Liang et al. 2000] which implements a small *reconfigurable communication processor* in each node. This processor has interconnect memory that programs the crossbar for data transfer from different sources across the node on each communication cycle.

The most common forwarding strategies are store-and-forward, wormhole, and virtual cut-through. These will now be explained. Table II summarizes the latency penalty and storage cost in each node for each of these schemes.

Store-and-forward. Store-and-forward routing is a packet switched protocol in which the node stores the complete packet and forwards it based on the information within its header. Thus the packet may stall if the router in the forwarding path does not have sufficient buffer space. The CLICHE [Kumar et al. 2002] is an example of a store-and-forward NoC.

Wormhole. Wormhole routing combines packet switching with the data streaming quality of circuit switching to attain a minimal packet latency. The node looks at the header of the packet to determine its next hop and immediately forwards it. The subsequent flits are forwarded as they arrive. This causes the packet to *worm* its way through the network, possibly spanning a number of nodes, hence the name. The latency within the router is not that of the whole packet. A stalling packet, however, has the unpleasantly expensive side effect of occupying all the links that the worm spans. In Section 3.2.3, we will see how virtual channels can relieve this side effect at a marginal cost. In Al-Tawil et al. [1997], a well-structured survey of wormhole routing techniques is provided, and a comparison between a number of schemes is made.

Virtual cut-through. Virtual cut-through routing has a forwarding mechanism similar to that of wormhole routing. But before forwarding the first flit of the packet, the node waits for a guarantee that the next node in the path will accept the entire packet. Thus if the packet stalls, it aggregates in the current node without blocking any links.

While macronetworks usually employ store-and-forward routing, the prevailing scheme for NoC is wormhole routing. Advantages are low latency and the avoidance of

area costly buffering queues. A special case of employing single flit packets is explored in Dally and Towles [2001]. Here the data and header bits of the packets are transmitted separately and in parallel across a link, and the data path is quite wide (256 bits). Each flit is thus a packet in its own right, holding information about its destination. Hence, unlike wormhole routing, the stream of flits may be interlaced with other streams and stalling is restricted to the local node. Still single flit latency is achieved. The cost is a higher header-to-payload ratio, resulting in larger bandwidth overhead.

3.2.3. Flow Control. Peh and Dally [2001] have defined flow control as the mechanism that determines the packet movement along the network path. Thus it encompasses both global and local issues. Flow control mainly addresses the issue of ensuring correct operation of the network. In addition, it can be extended to also include issues on utilizing network resources optimally and providing predictable performance of communication services. Flow control primitives thus also form the basis of differentiated communication services. This will be discussed further in Section 3.2.4

In the following, we first discuss the concept of virtual channels and their use in flow control. We then discuss a number of works in the area, and, finally, we address buffering issues.

Virtual channels (VCs). VCs are the sharing of a physical channel by several logically separate channels with individual and independent buffer queues. Generally, between 2 and 16 VCs per physical channel have been proposed for NoC. Their implementation results in an area and possibly also power and latency overhead due to the cost of control and buffer implementation. There are however a number of advantageous uses. Among these are:

- avoiding deadlocks.* Since VCs are not mutually dependent on each other, by adding VCs to links and choosing the routing scheme properly, one may break cycles in the resource dependency graph [Dally and Seitz 1987].
- optimizing wire utilization.* In future technologies, wire costs are projected to dominate over transistor costs [ITRS 2003]. Letting several logical channels share the physical wires, the wire utilization can be greatly increased. Advantages include reduced leakage power and wire routing congestion.
- improving performance.* VCs can generally be used to relax the interresource dependencies in the network, thus minimizing the frequency of stalls. In Dally [1992], it is shown that dividing a fixed buffer size across a number of VCs improve the network performance at high loads. In Duato and Pinkston [2001], the use of VCs to implement adaptive routing protocols is presented. Vaidya et al. [2001] and Cole et al. [2001] discusses the impact and benefit of supporting VCs.
- providing differentiated services.* Quality-of-service (QoS, see Section 3.2.4) can be used as a tool to optimize application performance. VCs can be used to implement such services by allowing high priority data streams to overtake those of lower priority [Felician and Furber 2004; Rostislav et al. 2005; Beigne et al. 2005] or by providing guaranteed service levels on dedicated connections [Bjerregaard and Sparsø 2005a].

To ensure correct operation, the flow control of the network must first and foremost avoid deadlock and livelock. Deadlock occurs when network resources (e.g., link bandwidth or buffer space) are suspended waiting for each other to be released, that is, where one path is blocked leading to other being blocked in a cyclic fashion [Dally and Seitz 1987]. It can be avoided by breaking cyclic dependencies in the resource dependency graph. Figure 13 illustrates how VCs can be used to prevent stalls due to dependencies on shared network resources. It is shown how in a network without VCs, stream B is

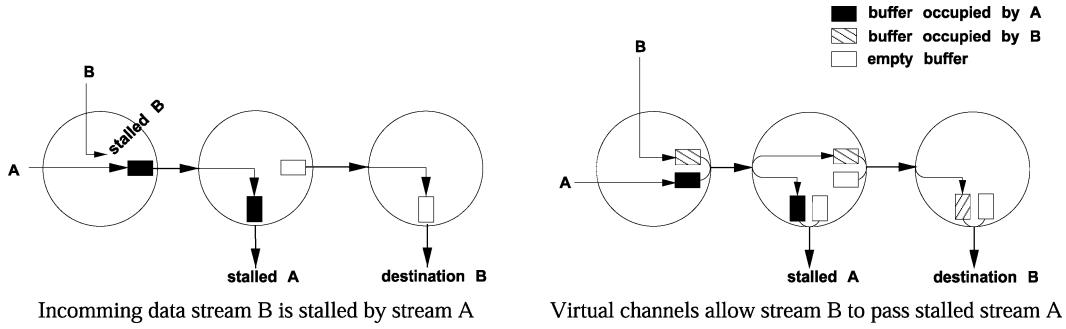


Fig. 13. Using virtual channels, independently buffered logical channels sharing a physical link, to prevent stalls in the network. Streams on different VCs can pass each other, while streams sharing buffer queues may stall.

stalled by stream A. In a network with VCs, however, stream B is assigned to a different VC with a separate buffer queue. Thus even though stream A is stalled stream B is enabled to pass.

Livelock occurs when resources constantly change state waiting for other to finish. Livelock is less common but may be expected in networks where packets are reinjected into the network or where backstepping is allowed, for example, during nonminimal adaptive routing.

Methods to avoid deadlock, and livelock can be applied either locally at the nodes with support from service primitives for example, implemented in hardware, or globally by ensuring logical separation of data streams by applying end-to-end control mechanisms. While local control is most widespread, the latter was presented in Millberg et al. [2004] using the concept of Temporally Disjoint Networks which was described in Section 3.2.2. As mentioned previously, dimension-ordered routing is a popular choice for NoC because it provides freedom from deadlock, without the need to introduce VCs. The *turn model* [Glass and Ni 1994] also does this but allows more flexibility in routing. A related approach is the *odd-even turn model* [Chiu 2000] for designing partially adaptive deadlock-free routing algorithms. Unlike the turn model, which relies on prohibiting certain turns in order to achieve freedom from deadlock, this model restricts the *locations* where some types of turns can be taken. As a result, the degree of routing adaptiveness provided is more even for different source-destination pairs. The ANoC [Beigne et al. 2005] implements this routing scheme.

The work of Jose Duato has addressed the mathematical foundations of routing algorithms. His main interests have been in the area of adaptive routing algorithms for multicomputer networks. Most of the concepts are directly applicable to NoC. In Duato [1993], the theoretical foundation for deadlock-free adaptive routing in wormhole networks is given. This builds on early work by Dally, which showed that by avoiding cyclic dependencies in the channel dependency graph of a network, deadlock-free operation is assured. Duato expands the theory to allow adaptive routing, and furthermore shows that the absence of cyclic dependencies is too restrictive. It is enough to require the existence of a channel subset which defines a connected routing subfunction with no cycles in its *extended* channel dependency graph. The extended channel dependency graph is defined in Duato [1993] as a graph for which the arcs are not only pairs of channels for which there is a direct dependency, but also pairs of channels for which there is an indirect dependency. In Duato [1995] and Duato [1996], this theory is refined and extended to cover also cut-through and store-and-forward routing. In Duato

and Pinkston [2001], a general theory is presented which glues together several of the previously proposed theories into a single theoretical framework.

In Dally and Aoki [1993], the authors investigated a hybrid of adaptive and deterministic routing algorithms using VCs. Packets are routed adaptively until a certain number of hops have been made in a direction away from the destination. Thereafter, the packets are routed deterministically in order to be able to guarantee deadlock-free operation. Thus the benefits of adaptive routing schemes are approached, while keeping the simplicity and predictability of deterministic schemes.

Other research has addressed flow control approaches purely for improving performance. In Peh and Dally [1999] and Kim et al. [2005], look-ahead arbitration schemes are used to allocate link and buffer access ahead of data arrival, thus reducing the end-to-end latency. This results in increased bandwidth utilization as well. Peh and Dally use virtual channels, and their approach is compared with simple virtual-channel flow control, as described in Dally [1992]. It shows an improvement in latency of about 15% across the entire spectrum of background traffic load, and network saturation occurs at a load 20% higher. Kim et al. do not use virtual channels. Their approach is shown to improve latency considerably (by 42%) when network load is low (10%) with much less improvement (13%) when network load is high (50%). In Mullins and Moore [2004], a virtual-channel router architecture for NoC is presented which optimizes routing latency by hiding control overheads, in a single cycle implementation.

Buffering. Buffers are an integral part of any network router. In by far the most NoC architectures, buffers account for the main part of the router area. As such, it is a major concern to minimize the amount of buffering necessary under given performance requirements. There are two main aspects of buffers (i) their size and (ii) their location within the router. In Kumar et al. [2002], it is shown that increasing the buffer size is not a solution towards avoiding congestion. At best, it delays the onset of congestion since the throughput is not increased. The performance improved marginally in relation to the power and area overhead. On the other hand, buffers are useful to absorb bursty traffic, thus leveling the bursts.

Tamir and Frazier [1988] have provided an comprehensive overview of advantages and disadvantages of different buffer configurations (size and location) and additionally proposed a buffering strategy called dynamically allocated multiqueue (DAMQ) buffer. In the argument of input vs. output buffers, for equal performance, the queue length in a system with output port buffering is always found to be shorter than the queue length in an equivalent system with input port buffering. This is so, since in a routing node with input buffers, a packet is blocked if it is queued behind a packet whose output port is busy (head-of-the-line-blocking). With regards to centralized buffer pools shared between multiple input and output ports vs distributed dedicated FIFOs, the centralized buffer implementations are found to be expensive in area due to overhead in control implementation and become bottlenecks during periods of congestion. The DAMQ buffering scheme allows independent access to the packets destined for each output port, while applying its free space to any incoming packet. DAMQ shows better performance than FIFO or statically-allocated shared buffer space per input-output port due to better utilization of the available buffer space especially for nonuniform traffic. In Rijpkema et al. [2001], a somewhat similar concept called virtual output queuing is explored. It combines moderate cost with high performance at the output queues. Here independent queues are designated to the output channels, thus enhancing the link utilization by bypassing blocked packets.

In Hu and Marculescu [2004a], the authors present an algorithm which sizes the (input) buffers in a mesh-type NoC on the basis of the traffic characteristics of a given

application. For three audio/video benchmarks, it was shown how such intelligent buffer allocation resulted in about 85% savings in buffering resources in comparison to uniform buffer sizes without any reduction in performance.

3.2.4. Quality of Service (QoS). QoS is defined as service quantification that is provided by the network to the demanding core. Thus it involves two aspects: (i) defining the services represented by a certain quantification and (ii) negotiating the services. The services could be low latency, high throughput, low power, bounds on jitter, etc. Negotiating implies balancing the service demands of the core with the services available from the network.

In Jantsch and Tenhunen [2003, 61–82], Goossens et al characterize the nature of QoS in relation to NoC. They identify two basic QoS classes, best-effort services (BE) which offer no commitment, and guaranteed services (GS) which do. They also present different levels of commitment, and discuss their effect on predictability of the communication behavior: 1) correctness of the result, 2) completion of the transaction, 3) bounds on the performance. In Rijpkema et al. [2001], argumentation for the necessity of a combination of BE and GS in NoC is provided. Basically, GS incur predictability, a quality which is often desirable, for example, in real-time systems, while BE improves the average resource utilization [Jantsch and Tenhunen 2003, 61–82; Goossens et al. 2002; Rijpkema et al. 2003]. More details of the advantages of GS from a design flow and system verification perspective are given in Goossens et al. [2005] in which a framework for the development of NoC-based SoC, using the *ÆTHEREAL* NoC, is described.

Strictly speaking, BE refers to communication for which no commitment can be given whatsoever. In most NoC-related works, however, BE covers the traffic for which only correctness and completion are guaranteed, while GS is traffic for which additional guarantees are given, that is, on the performance of a transaction. In macronetworks, service guarantees are often of a statistical nature. In tightly bound systems such as SoC, hard guarantees are often preferred. GS allows analytical system verification, and hence a true decoupling of subsystems. In order to give hard guarantees, GS communication must be logically independent of other traffic in the system. This requires connection-oriented routing. Connections are instantiated as virtual circuits which use logically independent resources, thus avoiding contention. The virtual circuits can be implemented by either virtual channels, time-slots, parallel switch fabric, etc. As the complexity of the system increases and as GS requirements grow, so does the number of virtual circuits and resources (buffers, arbitration logic, etc) needed to sustain them.

While hard service guarantees provide an ultimate level of predictability, soft (statistical) GS or GS/BE hybrids have also been the focus of some research. In Bolotin et al. [2004], Felicjan and Furber [2004], Beigne et al. [2005] and Rostislav et al. [2005], NoCs providing prioritized BE traffic classes are presented. SoCBUS [Sathe et al. 2003] provides hard, short-lived GS connections; however, since these are setup using BE routed packets, and torn down once used, this can also be categorized as soft GS.

ÆTHEREAL [Goossens et al. 2005], NOSTRUM [Millberg et al. 2004], MANGO [Bjerregaard and Sparsø 2005a], SONICS [Weber et al. 2005], aSOC [Liang et al. 2004], and also the NoCs presented in Liu et al. [2004], in Leroy et al. [2005], and the static NoC used in the RAW multiprocessor architecture [Taylor et al. 2002], are examples of NoCs implementing hard GS. While most NoCs that implement hard GS use variants of time division multiplexing (TDM) to implement connection-oriented packet routing, thus guaranteeing bandwidth on connections, the clockless NoC MANGO uses sequences of virtual channels to establish virtual end-to-end connections. Hence limitations of TDM, such as bandwidth and latency guarantees which are inversely proportional, can be overcome by appropriate scheduling. In Bjerregaard and Sparsø [2005b], a scheme

for guaranteeing latency, independently of bandwidth, is presented. In Leroy et al. [2005], an approach for allocating individual wires on the link for different connections is proposed. The authors call this *spatial division multiplexing* as opposed to TDM.

For readers interested in exploitation of GS (in terms of throughput) virtual circuits during idle times, in Andreasson and Kumar [2004, 2005] the concept of slack-time aware routing is introduced. A producer manages injection of BE packets during the slacks in time-slots reserved for GS packets, thereby mixing GS and BE traffic at the source which is unlike other schemes discussed so far where it is done in the routers. In Andreasson and Kumar [2005], the impact of variation of output buffer on BE latency is investigated, while in Andreasson and Kumar [2004], the change of injection control mechanism for fixed buffer size is documented. QoS can also be handled by controlling the injection of packets into a BE network. In Tortosa and Nurmi [2004], scheduling schemes for packet injection in a NoC with a ring topology were investigated. While a basic scheduling, which always favors traffic already in the ring, provided the highest total bandwidth, weighted scheduling schemes were much more fair in their serving of different cores in the system.

In addition to the above, QoS may also cover special services such as:

- broadcast, multicast, narrowcast*. These features allow simultaneous communication from one source to all, that is, broadcast, or select destinations as is shown in ÆTHEREAL [Jantsch and Tenhunen 2003, 61–82] where a master can perform read or write operations on an address-space distributed among many slaves. In a connection-oriented environment, the master request is channeled to a single slave for execution in narrowcast, while the master request is replicated for execution at all slaves in multicast. APIs are available within the NA to realize these types of transactions [Radulescu et al. 2004]. An alternate multicast implementation is discussed in Millberg et al. [2004] where a virtual circuit meanders through all the destinations.
- virtual wires*. This refers to the use of network message-passing services to emulate direct pin-to-pin connection. In Bjerregaard et al. [2005], such techniques are used to support a flexible interrupt scheme in which the interrupt of a slave core can be programmed to trigger any master attached to the network by sending a trigger packet.
- complex operations*. Complex functionality such as test-and-set issued by a single command across the network can be used to provide support for, for example, semaphores.

3.3. Link Level

Link-level research regards the node-to-node links. These links consist of one or more channels which can be either virtual or physical. In this section, we present a number of areas of interest for link level research: synchronization, implementation, reliability, and encoding.

3.3.1. Synchronization. For link-level synchronization in a multiclock domain SoC, Chelcea and Nowick [2001] have presented a mixed-time FIFO design. The FIFO employs a ring of storage elements in which tokens are used to indicate full or empty state. This simplifies detection of the state of the FIFO (full or empty) and thus makes synchronization robust. In addition, the definitions of full and empty are extended so that full means that 0 or 1 cell is unused, while empty means only 0 or 1 cells is used. This helps in hiding the synchronization delay introduced between the state detection and the input/output handshaking. The FIFO design introduced can be made arbitrarily robust with regards to metastability as settling time and latency can be traded off.

With the emerging of the GALS concept of *globally asynchronous locally synchronous* systems [Chapiro 1984; Meincke et al. 1999], implementing links using asynchronous circuit techniques [Sparsø and Furber 2001; Hauck 1995] is an obvious possibility. A major advantage of asynchronous design styles relevant for NoC is the fact that, apart from leakage, no power is consumed when the links are idle. Thus, the design style also addresses the problematic issue of increasing power usage by large chips. Another advantage is the potentially very low forward latency in uncongested data paths leading to direct performance benefits. Examples of NoCs based on asynchronous circuit techniques are CHAIN [Bainbridge and Furber 2002; Amde et al. 2005], MANGO [Bjerregaard and Sparsø 2005a], ANoC [Beigne et al. 2005], and QNoC [Rostislav et al. 2005]. Asynchronous logic incorporates some area and dynamic power overhead compared with synchronous logic due to local handshake control. The 1-of-4 encodings discussed in Section 3.3.4, generalized to 1-of-N, is often used in asynchronous links [Bainbridge and Furber 2001].

On the other hand, resynchronization of an incoming asynchronous transmission is also not trivial. It costs both time and power, and bit errors may be introduced. In Dobkin et al. [2004], resynchronization techniques are described, and a method for achieving high throughput across an asynchronous to synchronous boundary is proposed. The work is based on the use of stoppable clocks, a scheme in which the clock of a core is stopped while receiving data on an asynchronous input port. Limitations to this technique are discussed, and the proposed method involves only the clock on the input register being controlled. In Ginosaur [2003], a number of synchronization techniques are reviewed, and the pitfalls of the topic are addressed.

The trade-offs in the choice of synchronization scheme in a globally asynchronous or multiclocked system is sensitive to the latency requirements of the system, the expected network load during normal usage, the node complexity, etc.

3.3.2. Implementation Issues. As chip technologies scale into the DSM domain, the effect of wires on link delays and power consumption increase. Aspects and effects on wires of technology scaling are presented in Ho et al. [2001], Lee [1998], Havemann and Hutchby [2001], and Sylvester and Keutzer [2000]. In Liu et al. [2004], these issues are covered specifically from a NoC point-of-view, projecting the operating frequency and size of IP cores in NoC-based SoC designs for future CMOS technologies down to $0.05\text{ }\mu\text{m}$. In the following, we will discuss a number of physical level issues relevant to the implementation of on-chip links.

Wire segmentation. At the physical level, the challenge lies in designing fast, reliable and low power point-to-point interconnects, ranging across long distances. Since the delay of long on-chip wires is characterized by distributed RC charging, it has been standard procedure for some time to apply segmentation of long wires by inserting *repeater* buffers at regular intervals in order to keep the delay linearly dependent on the length of the wire. In Dobbelaere et al. [1995], an alternative type of repeater is proposed. Rather than splitting and inserting a buffer in the path of the wire, it is based on a scheme of sensing and pulling the wire using a keeper device attached to the wire. The method is shown to improve the delay of global wires by up to 2 times compared with conventional repeaters.

Pipelining. Partitioning long interconnects into pipeline stages as an alternative to wire segmentation is an effective way of increasing throughput. The flow control handshake loop is shorter in a pipelined link, making the critical loop faster. This is at the expense of latency of the link and circuit area since pipeline stages are more complex than repeater buffers. But the forward latency in an asynchronous pipeline handshake cycle can be minimized to a few gate delays so, as wire effects begin to dominate performance

in DSM technologies, the overhead of pipelining as opposed to buffering will dwindle. In Singh and Nowick [2000], several high-throughput clockless pipeline designs were implemented using dynamic logic. Completion detection was employed at each stage to generate acknowledge signals which were then used to control the precharging and evaluation of the dynamic nodes. The result was a very high throughput of up to 1.2 GDI/s (giga data items per second) for single rail designs, in a $0.6\text{ }\mu\text{m}$ CMOS technology. In Mizuno et al. [2001], a hybrid of wire segmentation and pipelining was shown in that a channel was made with segmentation buffers implemented as latches. A congestion signal traveling backwards through the channel compresses the data in the channel, storing it in the latches until the congestion is resolved. Thus a back pressure flow control scheme was employed without the cost of full pipeline elements.

Low swing drivers. In an RC charging system, the power consumption is proportional to the voltage shift squared. One way of lowering the power consumption for long on-chip interconnects is by applying low-swing signaling techniques which are also widely used for off-chip communication lines. Such techniques are presented and analyzed in Zhang et al. [1999]. Basically the power usage is lowered at the cost of the noise margin. However, a differential transmission line (2 wires), on which the voltage swing is half that of a given single-ended transmission line, has differential mode noise characteristics comparable to the single-ended version. This is so because the voltage difference between the two wires is the same as that between the single-ended wire and a mid-point between supply and ground. As an approximation, it uses only half the power, however, since the two wires working at half the swing each consume one-fourth the power. The common mode noise immunity of the differential version is also greatly improved, and it is thus less sensitive to crosstalk and ground bounces, important sources of noise in on-chip environments as discussed in the reliability section that follow. In Ho et al. [2003], the design of a low-swing, differential on-chip interconnect for the Smart Memories [Mai et al. 2000] is presented and validated with a test chip.

In Svensson [2001] the author demonstrated how an optimum voltage swing for minimum power consumption in on- and off-chip interconnects can be found for a given data activity rate. The work takes into account dynamic and static power consumption of driving the wire as well as in the receiver, which needs to amplify the signal back to full logic level. Calculations are presented for a $0.18\text{ }\mu\text{m}$ CMOS technology. Figure 14 displays the power consumption versus voltage swing for a global on-chip wire of 5–10 mm, a power supply of 1.3 V, and a clock frequency of 1 GHz. For a data activity rate of 0.25 (random data), it is shown that there is a minimum at 0.12 V. This minimum occurs for a two-stage receiver amplifier and corresponds to a power saving of 17x. Using a single stage amplifier in the receiver, there is a minimum at 0.26 V, corresponding to a power saving of 14x.

Future issues. In Heiliger et al. [1997], the use of microstrip transmission lines as waveguides for sub-mm wave on-chip interconnects is analyzed. It is shown that using SiO_2 as dielectric exhibits prohibitively large attenuation. However, the use of bisbenzocyclobutene-polymer offers favorable line parameters, with an almost dispersion free behavior at moderate attenuation ($\leq 1\text{ dB/mm}$ at 100 GHz). In Kapur and Saraswat [2003], a comparison between electrical and optical interconnects for on-chip signaling and clock distribution is presented. Figure 15 shows the models used in evaluating optical and electrical communication. The delay vs. power and delay vs. interconnect length trade-offs are analyzed for the two types of signaling. In Figure 16, it is shown that the critical length above which the optical system is faster than the electrical one is approximately 3–5 mm, projected for a 50 nm CMOS fabrication technology with copper wiring. The work also shows that, for long interconnects (defined as 10 mm and above), the optical communication has a great potential for low power

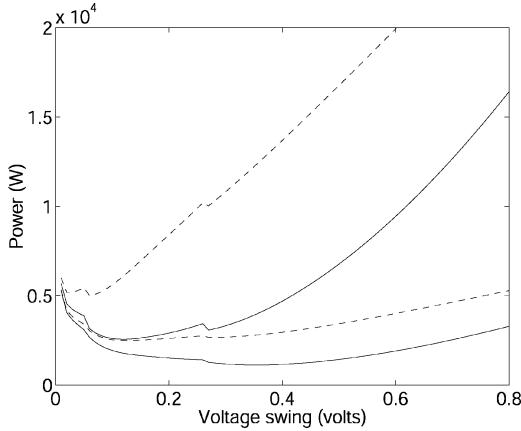


Fig. 14. Total power versus voltage swing for long (5–10 mm) on-chip interconnect. Solid line case 1: power supply generated off-chip by high efficiency DC-DC converter. Dashed line case 2: power supply generated internally on-chip. Upper curves for data activity of 0.25, lower curves 0.05 (reprinted from Svensson [2001] Fig. 2, ©2001 with permission from Christer Svensson).

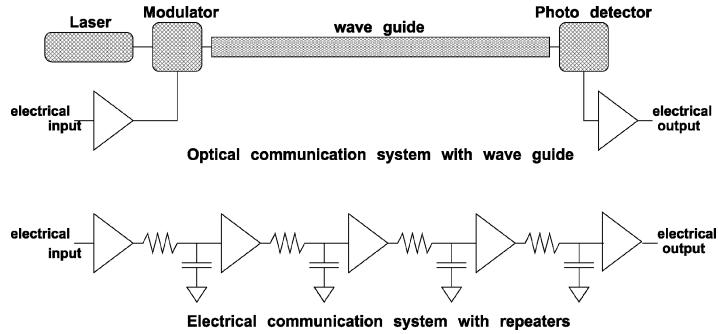


Fig. 15. Model of electrical and optical signaling systems for on-chip communication, showing the basic differences.

operation. Thus it is projected to be of great use in future clock distribution and global signaling.

3.3.3. Reliability. Designing global interconnects in DSM technologies, a number of communication reliability issues become relevant. Noise sources which can have an influence on this are mainly crosstalk, power supply noise such as ground bounce, electromagnetic interference (EMI), and intersymbol interference.

Crosstalk is becoming a serious issue due to decreasing supply voltage, increasing wire to wire capacitance, increasing wire inductance (e.g., in power supply lines), and increasing rise times of signaling wires. The wire length at which the peak crosstalk voltage is 10% of the supply voltage decreases drastically with technology scaling [Jantsch and Tenhunen 2003, chap. 6], and, since the length of global interconnects does not scale with technology scaling, this issue is especially relevant to the implementation of

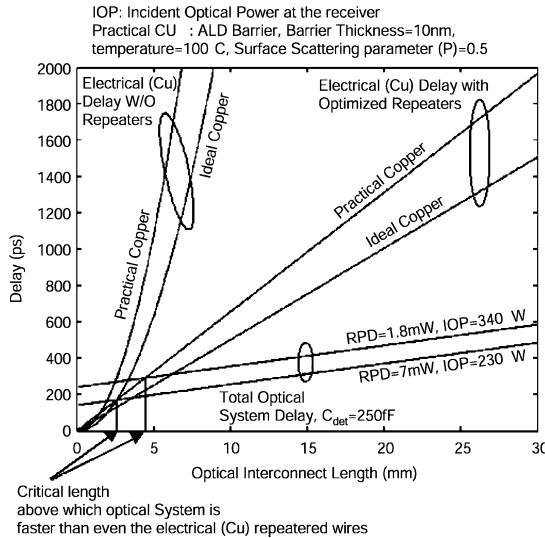


Fig. 16. Delay comparison of optical and electrical interconnect (with and without repeaters) in a projected 50 nm technology (reprinted from Kapur and Saraswat [2003] by Pawan Kapur and Krishna C. Saraswat, Fig. 13, ©2002, with permission from Elsevier).

NoC links. Power supply noise is worsened by the inductance in the package bonding wires, and the insufficient capacitance in the on-chip power grid. The effect of EMI is worsening as the electric charges moved by each operation in the circuit is getting smaller, making it more susceptible to external influence. Intersymbol interference, that is, the interference of one symbol on the following symbol on the same wire, is increasing as circuit speeds go up.

In Jantsch and Tenhunen [2003, chap. 6], Bertozzi and Benini present and analyze a number of error detecting/correcting encoding schemes in relation to NoC link implementation. Error recovery is a very important issue, since an error in, for instance, the header of a packet, may lead to deadlock in the NoC, blocking the operation of the entire chip. This is also recognized in Zimmer and Jantsch [2003] in which a fault model notation is proposed which can represent multiwire and multicycle faults. This is interesting due to the fact that crosstalk in DSM busses can cause errors across a range of adjacent bits. It is shown that, by splitting a wide bus into separate error detection bundles, and interleaving these, the error rate after using single-error correcting and double-error detecting codes can be reduced by several orders of a magnitude. This is because these error-correction schemes function properly when only one or two errors occur in each bundle. When the bundles are interleaved, the probability of multiple errors within the same bundle is greatly reduced.

In Gaughan et al. [1996] the authors deal with dynamically occurring errors in networks with faulty links. Their focus is on routing algorithms that can accommodate such errors, assuming that support for the detection of the errors is implemented. For wormhole routing, they present a scheme in which a data transmission is terminated upon detection of an error. A *kill* flit is transmitted backwards, deleting the worm and telling the sender to retransmit it. This naturally presents an overhead and is not generally representative for excising NoC implementations. It can, however, prove necessary in mission critical systems. The paper provides formal mathematical proofs of deadlock-freedom.

Another issue with new CMOS technologies is the fact that the delay distribution—due to process variations—flattens with each new generation. While typical delay improves, worst case delay barely changes. This presents a major problem in todays design methodologies as these are mostly based on worst case assumptions. Self-calibrating methods, as used in Worm et al. [2005], are a way of dealing with unreliability issues of this character. The paper presents a self-calibrating link, and the problem of adaptively controlling its voltage and frequency. The object is to maintain acceptable design trade-offs between power consumption, performance, and reliability when designing on-chip communication systems using DSM technologies.

Redundant transmission of messages in the network is also a way of dealing with fabrication faults. In Pirretti et al. [2004], two different flooding algorithms and a random walk algorithm are compared. It is shown that the flooding algorithms have an exceedingly large communication overhead, while the random walk offers reduced overhead and still maintains useful levels of fault tolerance.

With the aim of improving fabrication yield, Dally and Towles [2001] propose extra wires between nodes so that defective wires found during postproduction tests or during self-test at start-up can be bypassed. Another potential advantage of a distributed shared communication structure is the possibility of bypassing entire regions of a chip if fabrication faults are found.

Dynamic errors are more likely in long wires and segmenting links into pipeline stages helps to keep the error rate down and the transmission speed up. Since segmentation of the communication infrastructure is one of the core concepts of NoC, it inherently provides solutions to the reliability problems. The segmentation is made possible because NoC-based systems generally imply the use of programming models allowing some degree of latency insensitive communication. Thus it is shown how the issues and solutions at the physical level relate directly to issues and solutions at system level, and vice versa. Another solution towards avoiding dynamic errors is the shielding of signal wires, for example, by ground wires. This helps to minimize crosstalk from locally interfering wires at the expense of wiring area.

3.3.4. Encoding. Using encoding for on-chip communication has been proposed; the most common objective is to reduce power usage per communicated bit, while maintaining high speed and good noise margin. In Bogliolo [2001], the proposed encoding techniques are categorized as speed-enhancing or low-power encodings, and it is shown how different schemes in these two categories can be combined to gain the benefits of both. In Nakamura and Horowitz [1996], a very simple low-weight coding technique was used to reduce dI/dt noise due to simultaneous switching of off-chip I/O drivers. An 8-bit signal was simply converted to a 9-bit signal, the 9th bit indicating whether the other 8 bits should be inverted. The density of 1's was thus reduced, resulting in a reduction of switching noise by 50% and of power consumption by 18%. Similar techniques could prove useful in relation to long on-chip interconnects. The abundant wire resources available on-chip can also be used to implement more complex M-of-N encodings, thus trading wires for power. A widely used technique, especially in asynchronous implementations, is 1-of-4 encoding. This results in a good power/area trade-off and low encoding/decoding overhead [Bainbridge and Furber 2001; Bainbridge and Furber 2002].

Another area of encoding, also discussed in Section 3.3.3, relates to error management. This involves the detection and correction of errors that may occur in the network. The mechanism may be observed at different layers of the network and thus be applicable to either phits, flits, packets, or messages. With regards to NoC, the interesting issues involve errors in the links connecting the nodes since long wires

of deep submicron technologies may exhibit unreliable behavior (see Section 3.3.3). xpipes [Osso et al. 2003] implements a flit-level CRC mechanism, running in parallel with switching (thus masking its delay), to detect errors. Another common technique is parity-checks. The need here is to balance complexity of error-correction circuits to the urgency of such mechanisms.

An interesting result is obtained in Jantsch and Vitkowski [2005] wherein the authors investigate power consumption in the NOSTRUM NoC. Results are based on a $0.18\text{ }\mu\text{m}$ implementation and scaled down to 65 nm. The paper concludes that the major part of the power is spent in the link wires. Power-saving encoding however reduces performance and simply scaling the supply voltage to normalize performance—in nonencoded links—actually results in better power figures than any of the encoding schemes investigated. Subsequently, the authors propose the use of end-to-end data protection through error correction methods which allows voltage scaling, while maintaining the fault probability without lowering the link speed. In effect, this results in better power figures.

In this section, we have discussed issues relevant to the lowest level of the NoC, the link level. This concludes the discussion of network design and implementation topics. In the following section, we discuss NoC from the view of design approach and modeling in relation to SoC.

4. NOC MODELING

NoC, described as a subset of SoC, is an integral part of SoC design methodology and architecture. Given the vast design space and implementation decisions involved in NoC design, modeling and simulation is important to design flow, integration, and verification of NoC concepts. In this section, we first discuss issues related to NoC modeling, and then we explore design methodology used to study the system-level impact of the NoC. Finally, traffic characterization, which bridges system-level dynamics with NoC requirements, is discussed.

4.1. Modeling

Modeling the NoC in abstract software models is the first means to approach and understand the required NoC architecture and the traffic within it. Conceptually the purpose of NoC modeling is (i) to explore the vast design and feature space, and (ii) to evaluate trade-offs between power, area, design-time, etc; while adhering to application requirements on one side and technology constraints on the other side. Modeling NoC has three intertwined aspects: modeling environment, abstraction levels, and result analysis. In the modeling environment section, we present three frameworks to describe NoC. Section 4.1.2 discusses work done across different levels of NoC abstraction. The result analysis deals with a wide range of issues and is hence dealt with separately in Section 5.

4.1.1. Modeling Environment. The NoC models are either analytical or simulation based and can model communication across abstractions.

In a purely abstract framework, a NoC model using allocators, scheduler, and synchronizer is presented in Madsen et al. [2003] and Mahadevan et al. [2005]. The allocator translates the path traversal requirements of the message in terms of its resource requirements such as bandwidth, buffers, etc. It attempts to minimize resource conflicts. The scheduler executes the message transfer according to the particular network service requirements. It attempts to minimize resource occupancy. A synchronizer models the dependencies among communicating messages allowing concurrency. Thus these

three components are well suited to describe a wide variety of NoC architectures and can be simulated in a multiprocessor real-time environment.

OPNET, a commercial network simulator originally developed for macronetworks, is used as a NoC simulator in Bolotin et al. [2004], Xu et al. [2004], and Xu et al. [2005]. OPNET provides a convenient tool for hierarchical modeling of a network, including processes (state machines), network topology description, and simulation of different traffic scenarios. However, as noted in Xu et al. [2004] and Xu et al. [2005], it needs to be adapted for synchronous environments, requiring explicit design of clocking scheme and a distribution network. Bolotin et al. [2004] uses OPNET to model a QoS-based NoC architecture and design with irregular network topology.

A VHDL-based cycle accurate RTL model for evaluating power and performance of NoC architecture is presented in Banerjee et al. [2004]. The power and delay are evaluated for fine-grain components of the routers and links using SPICE simulations for a $0.18\text{ }\mu\text{m}$ technology and incorporated into the architectural-level blocks. Such modeling enables easy evaluation of dynamic vs leakage power at the system level. As expected, at high injection rate (packets/cycle/node), it was found that dynamic power dominates over leakage power. The Orion power performance simulator proposed by Wang et al. [2002] modeled only dynamic power consumption.

Recently, due to the increasing size of applications, NoC emulation [Genko et al. 2005] has been proposed as an alternative to simulation-based NoC models. It has been shown that FPGA-based emulation can take a few seconds compared to simulation-based approaches which can take hours to process through many millions of cycles as would be necessary in any thorough communication coexploration.

4.1.2. Noc Modeling at Different Abstraction Levels. New hardware description languages are emerging, such as SystemC [2002], a library of C++, and SystemVerilog [Fitzpatrick 2004], which make simulations at a broad range of abstraction levels readily available and thus support the full range of abstractions needed in a modular NoC-based SoC design. In Bjerregaard et al. [2004], mixed-mode asynchronous handshake channels were developed in SystemC, and a mixed abstraction-level design flow was used to design two different NoC topologies.

From an architectural point of view, the network topology generally incur the use of a segmented (multihop) communication structure, however, some researchers, working at the highest levels of abstraction, define NoC merely as a multiport blackbox communication structure or core, presenting a number of ports for communication. A message can be transmitted from an arbitrary port to any other, allowing maximum flexibility of system communication. At this level, the actual implementation of the NoC is often not considered. Working at this high abstraction level allows a great degree of freedom from lower level issues. Table III adapted from Gerstlauer [2003] summarizes, in general, the communication primitives at different levels of abstraction.

At system level, transaction-level models (TLM) are typically used for modeling communication behavior. This takes the form of either synchronous or asynchronous *send()* / *receive()* message passing semantics which use unique channels for communication between the source and the destination. One level below this abstraction, for NoCs, additional identifiers such as addressing may be needed to uniquely identify the traversal path or for providing services for end-to-end communication. Control primitives at network and link level, which are representative of actual hardware implementation, model the NoC flow-control mechanisms. In Gerstlauer [2003], a JPEG encoder and voice encoder/decoder running concurrently were modeled for each and for mixed levels of abstraction. The results show that the model complexity generally grows exponentially with a lower level of abstraction. By extrapolating the result from bus to NoC, interestingly, model complexity at NA level can be found to be higher than at other

Table III. Communication Semantics and Abstraction for NoC, Adapted From Gerstlauer [2003]

Layer	Interface semantics	Communication
Application/ Presentation	IP-to-IP messaging <code>sys.send(struct myData)</code> <code>sys.receive(struct myData)</code>	Message passing
Session/ Transport	IP-to-IP port-oriented messaging <code>nwk.read(messagepointer*, unsigned len)</code> <code>nwk.write(int addr, msgptr*, unsigned len)</code>	Message passing or shared memory
Network	NA-to-NA packet streams <code>ctrl.send()</code> , <code>ctrl.receive()</code> <code>link.read(bit[] path, bit[] data_packet)</code> <code>link.write(bit[] path, bit[] data_packet)</code>	Message passing or shared memory
Link	Node-to-Node logical links and shared byte streams <code>ctrl.send()</code> , <code>ctrl.receive()</code> <code>channel.transmit(bit[] link, bit[] data_flit)</code> <code>channel.receive(bit[] link, bit[] data_flit)</code>	Message passing
Physical	Pins and wires <code>A.drive(0)</code> , <code>D.sample()</code> , <code>clk.tick()</code>	Interconnect

levels due to the slicing of message, connection management, buffer management, and others.

Working between a session to network layer, Juurlink and Wijshoff [1998] have made a comparison of three communication models used in parallel computation: (i) asynchronous communication with fixed message size, (ii) synchronous communication which rewards burst-mode message transfers, and (iii) asynchronous with variable message size communication while also accounting for network load. Cost-benefit analysis shows that, though the software-based messaging layers serve a very useful function of delinking computation and communication, it creates anywhere from between 25% to 200% overhead as opposed to optimized hardware implementation.

A similar study of parallel computation applications, but with a more detailed network model, was undertaken by Vaidya et al. [2001]. Here the network was implemented to use adaptive routing with virtual channels. The applications, running on power-of-two number of processors using grid-based network topologies, used shared memory or message passing for communication, thus generating a wide range of traffic patterns. They found that increasing the number of VCs and routing adaptively offers little performance improvement for scalable shared memory applications. Their observation holds true over a range of systems and problem sizes. The results show that the single most important factor for improving performance in such applications is the router speed which is likely to provide lasting payoffs. The benefits of a faster router are visible across all applications in a consistent and predictable fashion.

Ahonen et al. [2004] and Lahiri et al. [2001] have associated high-level modeling aspects with actual design choices, such as selection of an appropriate topology, selection of communication protocols, specification of architectural parameters (such as bus widths, burst transfer size, priorities, etc), and mapping communications onto the architecture, as requirements to optimize the on-chip communication for application-specific needs. Using a tool called OIDIPUS, Ahonen et al. [2004] compare placement of twelve processors in a ring-based topology. They found that OIDIPUS, which uses the physical path taken by the communication channels as the cost function, generated topologies that are only marginally inferior to human design. Without being restricted to any one topology, Lahiri et al. [2001] have evaluated traffic characteristics in a static priority-based shared bus, hierarchical bus, two-level time division multiplexed access (TDMA), and ring-based communication architecture. They found that no single architecture uniformly outperforms other.

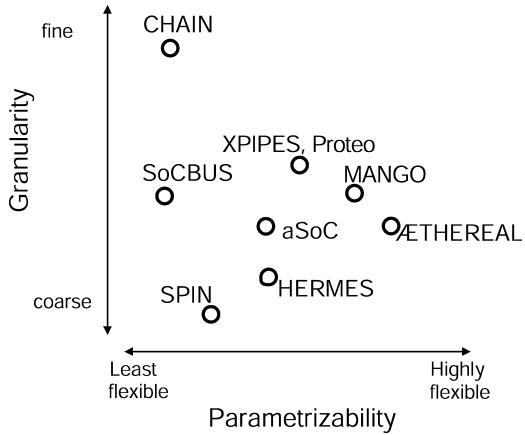


Fig. 17. NoC instantiation space.

Wieferink [2004] have demonstrated a processor/communication coexploration methodology which works cross-abstraction and in a cosimulation platform. Here LISA-based IP core descriptions have been integrated with SystemC-based bus-based transaction-level models. A wide range of APIs are then provided to allow modeling between LISA and SystemC models, to allow instruction accurate models to coexist with cycle accurate models and TLM with RTL models. MPARM [Loghi et al. 2004] is a similar cycle accurate and SystemC coexploration platform used in exploration of AMBA, STBus, and xpipes NoC evaluation.

4.2. Design and Coexploration Methodology

The NoC components, as described in Section 2.1, lends itself to flexible NoC designs such as parameterizable singular IP core or malleable building blocks, customizable at the network layer for design and reuse into application-specific NoC. A SoC design methodology requiring a communication infrastructure can exploit either characteristics to suit the application's needs. Keeping this in mind, different NoC researchers have uniquely tailored their NoC architectures. Figure 17 shows our assessment of instance-specific capability of these NoC architectures. The two axis are explained as follows.

- Parametrizability at system-level.* By this, we mean the ease with which a system-level NoC characteristic can be changed at instantiation time. The NoC description may encompass a wide range of parameters, such as: number of slots in the switch, pipeline stages in the links, number of ports of the network, and others. This is very useful for coexploration directly with IP cores of the SoC.
- Granularity of NoC.* By granularity, we mean at what level the NoC or NoC components is described. At the coarser end, the NoC may be described as a single core, while at the other end of the spectrum, the NoC may be assembled from lower-level blocks.

Consider the example of CHAIN [Bainbridge and Furber 2002]. It provides a library of fine-grained NoC components. Using these components, a NoC designer can use a Lego-brick approach to build the desired NoC topology, though as system-level block such a NoC has low flexibility. Thus it may be disadvantageous when trying to find the optimum SoC communication architecture in a recursive design space exploration

process. The *ÆTHEREAL* [Goossens et al. 2002], SoCBUS [Sathe et al. 2003], and aSoC [Liang et al. 2000] networks describe the NoC as a relatively coarse grain system-level module but with widely different characteristics. The *ÆTHEREAL* is highly flexible in terms of adjusting the available slots, number of ports, etc., which is useful for NoC exploration; whereas aSoC and SoCBUS do not expose many parameters for change (though aSoC supports flexible programming of connections after instantiation). The SPIN NoC [Guerrier and Greiner 2000], designed as a single IP core, is least parameterizable with its fixed topology and protocol. Interestingly, the xpipes [Osso et al. 2003] provides not merely a set of relatively fine-grain soft-macros of switches and pipelined links which the xpipesCompiler [Jalabert et al. 2004] uses to automatically instantiate an application-specific network, but also enables optimization of system-level parameters such as removing redundant buffers from output ports of switches, sharing signals common to objects, etc. This lends itself to both flexibility for coexploration and easy architectural changes when needed. Similarly, conclusions can be drawn of Proteo [Siguenza-Tortosa et al. 2004], HERMES [Moraes et al. 2004] and MANGO [Bjerregaard and Sparsø 2005a] NoCs. A detailed comparison of different features of most of the listed NoCs is tabulated in Moraes et al. [2004].

The impact on SoC design time and coexploration of different NoC design styles listed is considerable. For example, in Jalabert et al. [2004], during design space exploration, to find an optimum NoC for three video applications, that is, video object plane decoder, MPEG4 decoder and multiwindow display, the xpipesCompiler found that irregular networks with large switches may be more advantageous than regular networks. This is easier to realize in a macroblock NoC such as CHAIN or xpipes than it is in NoC designed as a single (system level) IP core such as SPIN. The basis for the compiler's decision is the pattern of traffic generated by the application. This is the focus of the next section. Further explanation of trade-offs in using a flexible instantiation-specific NoC can be found in Pestana et al. [2004] where different NoC topologies and each topology with different router and NA configuration is explored.

4.3. Traffic Characterization

The communication types expected in a NoC range across virtual wires, memory access, audio/video stream, interrupts, and others. Many combinations of topology, protocol, packet sizes, and flow control mechanisms exist for the efficient communication of one or more predominant traffic patterns. For example, in Kumar et al. [2002], packet-switched NoC concepts have been applied to a 2D mesh network topology, whereas in Guerrier and Greiner [2000], such concepts have been applied to a butterfly fat-tree topology. The design decisions were based on the traffic expected in the respective systems. Characterizing the expected traffic is an important first step towards making sound design decisions.

A NoC must accommodate different types of communication. We have realized that, regardless of the system composition, clustering, topology, and protocol, the traffic within a system will fall into one of three categories.

- (1) *Latency Critical.* Latency critical traffic is traffic with stringent latency demands such as for critical interrupts, memory access, etc. These often have low payload.
- (2) *Data Streams.* Data streaming traffic have high payload and demand QoS in terms of bandwidth. Most often it is large, mostly fixed bandwidth, which may be jitter critical. Examples are MPEG data, DMA access, etc.
- (3) *Miscellaneous.* This is traffic with no specific requirements of commitment from the network.

This categorization is a guideline rather than a hard specification and is presented as a superset of possible traffic types. Bolotin et al. [2004] provide a more refined traffic categorization, combining the transactions at the network boundary with service requirements, namely, signaling, real-time, read/write (RD/WR), and block transfer. In relation to the previous categorization, signaling is latency critical, real-time is data streaming, and RD/WR and block transfer are both miscellaneous with the message size as the distinguishing factor. Though one or more of the traffic patterns may be predominant in the SoC, it is important to understand that a realistic NoC design should be optimized for a mix of traffic patterns. The conclusions of a case study of NoC routing mechanism for three traffic conditions with a fixed number of flits per packet as presented in Ost et al. [2005] can thus be enriched by using nonuniform packet size and relating them to the traffic categories presented.

It is important to understand the bandwidth requirements of the listed traffic types for a given application, and accordingly map the IP cores on the chosen NoC topology. Such a study is done in Murali and Micheli [2004a]. NMAP (now called SUNMAP [Murali and Micheli 2004b]), a fast mapping algorithm that minimizes the average communication delay with minimal path and split traffic routing in 2D mesh, is compared with greedy and partial branch-and-bound algorithms. It is shown to produce results of higher merit (reduced packet latency) for DSP benchmarks. Another dimension in the mapping task is that of allocating guaranteed communication resources. In Goossens et al. [2005] and Hansson et al. [2005] approaches to this task are explored for the *AETHEREAL* NoC.

Specific to the data stream type traffic described, Rixner et al. [1998] have identified unique qualities relating to the interdependencies between the media streams and frequency of such streams in the system. It is called the streaming programming model. The basic premises of such programming is static analysis of the application to optimize the mapping effort based on prior knowledge of the traffic pattern so as to minimize communication. The communication architecture tuner (CAT) proposed by Lahiri et al. [2000] is a hardware-based approach that does runtime analysis of traffic and manipulates the underlying NoC protocol. It does this by monitoring the internal state and communication transactions of each core and then predicts the relative importance of each communication event in terms of its potential impact on different system-level performance metrics such as number of deadline misses, average processing time, etc.

The various blocks of NoC can be tuned for optimum performance with regard to a specific traffic characteristic, or the aim can be more general, towards a one-fits-all network, for greater flexibility and versatility.

5. NETWORK ANALYSIS

The most interesting and universally applicable parameters of NoC are *latency*, *bandwidth*, *jitter*, *power consumption*, and *area usage*. Latency, bandwidth and jitter can be classified as performance parameters, while power consumption and area usage are the cost factors. In this section, we will discuss the analysis and presentation of results in relation to these parameters.

5.1. Performance Parameters and Benchmarks

Specifying a single one of the performance parameters previously introduced is not sufficient to confer a properly constrained NoC behavior. The following example illustrates this.

Given a network during normal operation, it is assumed that the network is not overloaded. For such a network, all data is guaranteed to reach its destination when

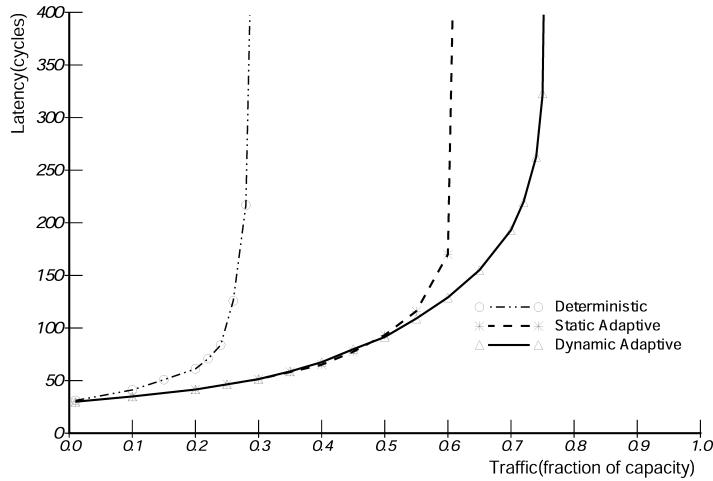


Fig. 18. Latency vs. network load for different routing schemes. The figure shows how the employment of more complex routing schemes move the point at which the network saturates (reprinted from Dally and Aoki [1993] Fig. 5, ©1993, with permission from IEEE).

employing a routing scheme in which no data is dropped (see Section 3.2.2, delay routing model). This means that, as long as the capacity of the network is not exceeded, any transmission is guaranteed to succeed (any required bandwidth is guaranteed). However, nothing is stated concerning the transmission latency which may well be very high in a network operated near full capacity. As shown in Figure 18, the exact meaning of which will be explained later, the latency of packets rise in an exponential manner as the *network load* increases. The exact nature of the network load will be detailed later in this section. It is obvious that such guarantees are not practically usable. We observe that the bandwidth specification is worthless without a bound on the latency as well. This might also be presented in terms of a maximum time window within which the specified bandwidth would always be reached, that is, the jitter of the data stream (the spread of the latencies). Jitter is often a more interesting parameter in relation to bandwidth than latency as it describes the temporal evenness of the data stream.

Likewise, a guaranteed bound on latency might be irrelevant if the bandwidth supported at this latency is insufficient. Thus latency, bandwidth, and jitter are closely related. Strictly speaking, one should not be specified without at least one of the others.

At a higher abstraction level, performance parameters used in evaluating multi-computer networks in general have been adopted by NoC researchers. These include *aggregated bandwidth*, *bisection bandwidth*, *link utilization*, *network load*, etc. The aggregate bandwidth is the accumulated bandwidth of all links, and the bisection bandwidth is the minimum collective bandwidth across links that, when cut, separate the network into two equal set of nodes. Link utilization is the load on the link compared with the total bandwidth available. The network load can be measured as a fraction of the network capacity, as normalized bandwidth. The network capacity is the maximum capacity of the network for a uniform traffic distribution, assuming that the most heavily loaded links are located in the network bisection. These and other aspects of network performance metrics are discussed in detail in Chapter 9 of Duato et al. [2003].

For highly complex systems, such as full-fledged computer systems including processor(s), memory, and peripherals, the individual parameters may say little about the

overall functionality and performance of the system. In such cases, it is customary to make use of benchmarks. NoC-based systems represent such complexity, and benchmarks would be natural to use in its evaluation. Presenting performance in the form of benchmark results would help clarify the effect of implemented features in terms of both performance benefits (latency, jitter, and bandwidth) and implementation and operation costs (area usage and power consumption). Benchmarks would thus provide a uniform plane of reference from which to evaluate different NoC architectures. At present, no benchmark system exists explicitly for NoC, but its development is an exciting prospect. In Vaidya et al. [2001], examples from the NAS benchmarks [Bailey et al. 1994] were used, in particular Class-A NAS-2. This is a set of benchmarks that has been developed for the performance evaluation of highly parallel supercomputers which mimic the computation and data movement characteristics of large scale computational fluid dynamics applications. It is questionable, however, how such parallel computer benchmarks can be used in NoC as the applications in SoCs are very different. In particular, SoC applications are generally highly heterogeneous, and the traffic patterns therein likewise. Another set of benchmarks, used as the basis of NoC evaluation in Hu and Marculescu [2004a], are the embedded system synthesis suite (E3S) [Dick 2002].

5.2. Presenting Results

Generally it is necessary to simplify the multidimensional performance space. One common approach is to adjust a single aspect of the design, while tracking the effect on the performance parameters. An example is tracking the latency of packets, while adjusting the bandwidth capabilities of a certain link within the network, or the amount of background traffic generated by the test environment. In Section 5.2.1, we will give specific examples of simple yet informative ways of communicating results of NoC performance measurements.

Since the NoC is a shared, segmented communication structure wherein many individual data transfer sessions can take place in parallel, the performance measurements there in, not only on the traffic being measured therein, but also on the other traffic in the network, the *background traffic*. The degree of background traffic is often indicated by the network load as described earlier. Though very simple, this definition makes valuable sense in considering a homogeneous, uniformly loaded network. One generally applicable practical method for performance evaluation is thus generating a uniform randomly-distributed background traffic so that the network load reaches a specified point. Test packets can then be sent from one node to another, according to the situation that one desires to investigate, and the latencies of these packets can be recorded (see example (i) in Section 5.2.1).

Evenly distributed traffic, however, may cloud important issues of the network performance. In Dally and Aoki [1993], the degree of symmetry of the traffic distribution in the network was used to illustrate aspects of different types of routing protocols, adaptive and deterministic. The adaptive protocol resulted in a significant improvement of throughput over the deterministic one for nonuniform traffic but had little effect on performance with uniformly distributed traffic. The reason for this is that the effect of adaptive protocols is to even out the load to avoid hotspots, thus making better use of the available network resources. If the bulk load is already evenly distributed, there is no advantage. Also traffic parameters, like number of packets and packet size, can have a great influence on performance, for example, in relation to queueing strategies in nodes.

There are many ways to approach the task of presenting test results. The performance space is a complex, multidimensional one, and there are many pitfalls to be avoided in order to display intelligible and valuable information about the performance of a

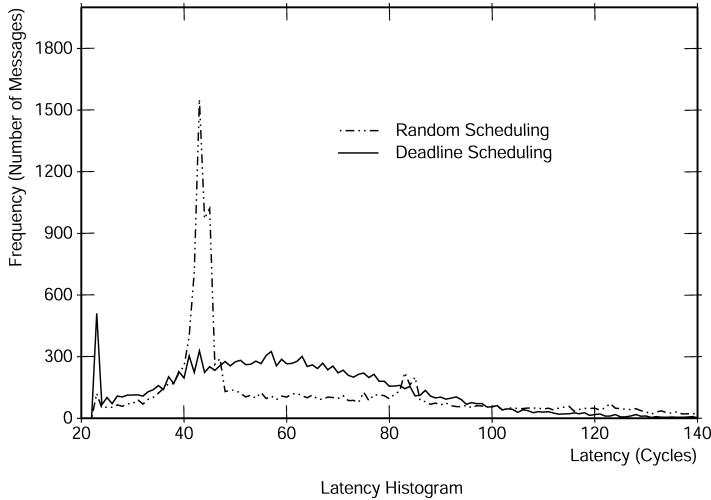


Fig. 19. Number of messages as a function of latency of message (latency distribution), for two scheduling schemes (reprinted from Dally [1992] Fig. 17, ©1992, with permission from IEEE).

network. Often the presented results fail to show the interesting aspects of the network. It is easy to get lost in the multitude of possible combinations of test parameters. This may lead to clouding (or at worst failure to properly communicate), the relevant aspects of the research. Though the basis for performance evaluation may vary greatly, it is important for researchers to be clear about the evaluation conditions, allowing others to readily and intuitively grasp the potential of a newly developed idea and the value of its usage in NoC.

5.2.1. Examples. We will now give some specific examples that we find clearly communicate the performance of the networks being analyzed. What makes these examples good are their simplicity in providing a clear picture of some very fundamental properties of the involved designs.

(i) *Average latency vs. network load.* In Dally and Aoki [1993], this is used to illustrate the effect of different routing schemes. Figure 18 is a figure from the article, showing how the average latency of the test data grows exponentially as the background traffic load of the network is increased. In the presented case, the *throughput saturation point*, the point at which the latency curve bends sharply upwards, is shifted right as more complex routing schemes are applied. This corresponds to a better utilization of available routing resources. The article does not address the question of cost factors of the implementation.

(ii) *Frequency of occurrence vs. latency of packet.* Displaying the average latency of packets in the network may work well for establishing a qualitative notion of network performance. Where more detail is needed, a histogram or similar graph showing the distribution of latencies across the delay spectrum is often used with great effect. This form of presentation is used in Dally [1992] to illustrate the effect of routing prioritization schemes on the latency distribution. Figure 19, taken from the article, shows the effect of *random scheduling* and *deadline scheduling*. Random scheduling schedules the packets for transmission in a random fashion, while deadline scheduling prioritizes packets according to how long they have been waiting (oldest-packet-first). It is shown how the choice of scheduling affects the distribution of latencies of messages. In

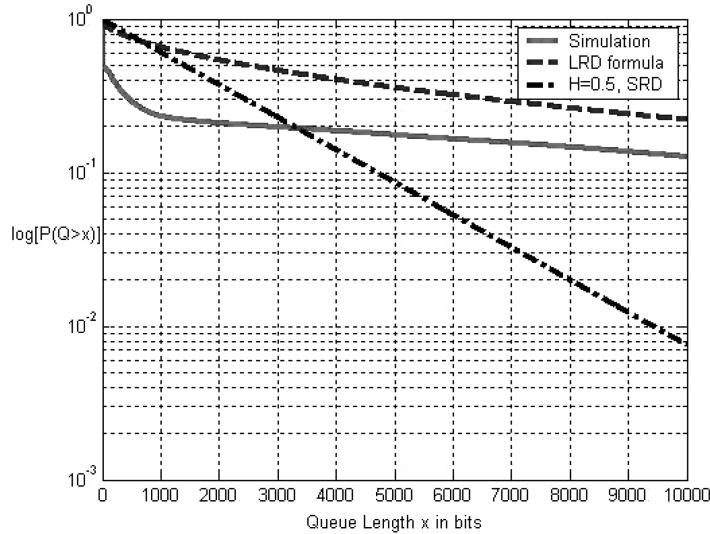


Fig. 20. The probability of queue length exceeding buffer size. The results for two models based on stochastic processes, LRD (Long Range Dependent) and SRD (Short Range Dependent), are plotted along with simulation results for comparison (reprinted from Varatkar and Marculescu [2002] Fig. 6).

Bjerregaard and Sparsø [2005b], such a latency distribution graph is also used to display how a scheduling scheme provides hard latency bounds in that the graph is completely empty beyond a certain latency.

(iii) *Jitter vs. network load.* The jitter of a sequence of packets is important when dimensioning buffers in the network nodes. High jitter (bursty traffic) needs large buffers to compensate in order to avoid congestion resulting in suboptimal utilization of routing resources. This issue is especially relevant in multimedia application systems with large continuous streams of data such as that presented in Varatkar and Marculescu [2002]. In this work, statistical mathematical methods are used to analyze the traffic distribution. Figure 20, taken from the article, explores the use of two different models based on stochastic processes for predicting the probability that the queue length needed to avoid congestion exceeds the actual buffer size in the given situation. The models displayed in the figure are LRD (Long Range Dependent) or *self-similar*, and SRD (Short Range Dependent) or *Markovian* stochastic processes. In the figure, these models are compared with simulation results. The contributions of the paper include showing that LRD processes can be used effectively to model the bursty traffic behavior at chip level, and the figure shows how indeed the predictions of the LRD model comes closer to the simulation results than those of the SRD model.

5.3. Cost Factors

The cost factors are basically power consumption and area usage. A comparative analysis of cost of NoC is difficult to make. As is the case for performance evaluation, no common ground for comparison exists. This would require different NoC being demonstrated for the same application which is most often not the case. Hence a somewhat broad discussion of cost in terms of area and power cost is presented in this section.

The power consumption of the communication structure in large single-chip systems is a major concern, especially for mobile applications. As discussed earlier, the power

used for global communication does not scale with technology scaling, leading to increased power use by communication relative to power use by processing. In calculating the power consumption of a system, there are two main terms: (i) power per communicated bit and (ii) idle power. Depending on the traffic characteristics in the network, different implementation styles will be more beneficial with regards to power usage. In Nielsen and Sparsø [2001], a power analysis of different low-power implementations of on-chip communication structures was made. The effects on power consumption of scaling a design were seen, and a bus design was compared with torus connected grid design (both synchronous and asynchronous implementations). Asynchronous implementation styles (discussed in Section 3.3.1), are beneficial for low network usage since they have very limited power consumption when idle, but use more power per communicated bit due to local control overhead. Technology scaling, however, leads to increased leakage current, resulting in an increasing static power use in transistors. Thus the benefit of low idle power in asynchronous circuits may dwindle.

From a system-level perspective, knowledge of network traffic can be used to control the power use of the cores. Interest has been expressed in investigating centralized versus distributed power management (DPM) schemes. Centralized power managers (PM) are a legacy in bus-based systems. Since NoC is most often characterized by distributed routing control, naturally distributed PMs, such as those proposed in Benini and Micheli [2001] and Simunic and Boyd [2002], would be useful. In both of these studies, conceptually there is a node-centric and network-centric PM. The node-centric PM controls the powering up or down of the core. The network-centric PM is used for overall load-balancing and to provide some estimations to the node-centric PM of incoming requests, thus masking the core's wake-up cost by precognition of traffic. This type of power management is expected to be favored to reduce power consumption in future NoCs. The results, presented in Simunic and Boyd [2002], show that, with only node PM, the power saving range from a factor of 1.5 to 3 compare to no power managers. Combining dynamic voltage scaling with DPM gives overall saving of a factor of 3.6. The combined implementation of node and network-centric management approaches shows energy savings of a factor of 4.1 with the performance penalty reduced by a minimum 15% compared to node-only PM. Unlike these dynamic runtime energy monitors, in Hu and Marculescu [2004b], a system-level energy-aware mapping and scheduling (EAS) algorithm is proposed which statically schedules both communication transactions and computation tasks. For experiments done on 2D mesh with minimal path routing, energy savings of 44% are reported when executing complex multimedia benchmarks.

A design constraint of NoC less applicable to traditional multicomputer networks lies in the area usage. A NoC is generally required to take up less than 5% of the total chip area. For a $0.13\text{ }\mu\text{m}$ SoC with one network node per core and an average core size of $2 \times 2\text{ mm}$ (approximately 100 cores on a large chip), this corresponds to 0.2 mm^2 per node. One must also remember that the NA will use some area, depending on the complexity of the features that it provides. Trade-off decisions which are applicable to chip design in general and not particular to NoC are beyond the scope of this survey. At the network level, many researchers have concluded that buffering accounts for the major portion of the node area, hence wormhole routing has been a very popular choice in NoCs (see Section 3.2.2). As examples of an area issue related to global wires, introducing *fat wires*, that is, the usage of wide and tall top-level metal wires for global routing, may improve the power figures but at the expense of area [Sylvester and Keutzer 2000].

6. NOC EXAMPLES

In this section, we briefly recapitulate a handful of specific NoC examples, describing the design choices of actual implementations and the accompanying work by the research

groups behind them. This is by no means a complete compilation of existing NoCs, there are many more, rather the purpose of this section is to address a representative set: *ÆTHEREAL*, *NOSTRUM*, *SPIN*, *CHAIN*, *MANGO*, and *XPIPES*. In Moraes et al. [2004], a list in tabular form is provided which effectively characterizes many of the NoCs not covered in the following.

- (1) ***ÆTHEREAL***. The *ÆTHEREAL*, developed at Philips, is a NoC that provides guaranteed throughput (GT) alongside best-effort (BE) service [Rijpkema et al. 2001; Goossens et al. 2002; Wielage and Goossens 2002; Dielissen et al. 2003; Jantsch and Tenhunen 2003] (pgs: 61-82) [Rijpkema et al. 2003; Radulescu et al. 2004; Goossens et al. 2005]. In the *ÆTHEREAL* the guaranteed services pervade as a requirement for hardware design and also as a foundation for software programming. The router provides both GT and BE services. All routers in the network have a common sense of time, and the routers forward traffic based on slot allocation. Thus a sequence of slots implement a virtual circuit. GT traffic is connection-oriented, and in early router instantiations, did not have headers as the next hop was determined by a local slot table. In recent versions, the slot tables have been removed to save area, and the information is provided in a GT packet header. The allocation of slots can be setup statically, during an initialization phase, or dynamically, during runtime. BE traffic makes use of non-reserved slots and of any slots reserved but not used. BE packets are used to program the GT slots of the routers. With regard to buffering, input queuing is implemented using custom-made hardware fifos to keep the area costs down. The *ÆTHEREAL* connections support a number of different transaction types, such as read, write, acknowledged write, test and set, and flush, and, as such, it is similar to existing bus protocols. In addition, it offers a number of connection types including narrowcast, multicast, and simple.

In Dielissen et al. [2003], an *ÆTHEREAL* router with 6 bidirectional ports of 32 bits was synthesized in $0.13 \mu\text{m}$ CMOS technology. The router had custom-made BE input queues depth of 24 words per port. The total area was 0.175 mm^2 , and the bandwidth was $500 \text{ MHz} \times 32 \text{ bits} = 16 \text{ Gbit/s}$ per port. A network adapter with 4 standard socket interfaces (either master or slave; OCP, DTL, or AXI based) was also reported with an area of 0.172 mm^2 implemented in the same technology.

In Goossens et al. [2005] and Pestana et al. [2004], an automated design flow for instantiation of application specific *ÆTHEREAL* is described. The flow uses XML to input various parameters such as traffic characteristics, GT and BE requirements, and topology. A case study of MPEG codec SoC is used to validate and verify the optimizations undertaken during the automated flow.

- (2) ***NOSTRUM***. The work of researchers at KTH in Stockholm has evolved from a system-level chip design approach [Kumar et al. 2002; Jantsch and Tenhunen 2003; Zimmer and Jantsch 2003; Millberg et al. 2004]. Their emphasis has been on architecture and platform-based design targeted towards multiple application domains. They have recognized the increasing complexity of working with high-density VLSI technologies and hence highlighted advantages of a grid-based, router-driven communication media for on-chip communication.

Also the implementation of guaranteed services has also been a focus point of this group. In the *NOSTRUM* NoC, guaranteed services are provided by so called *looped containers*. These are implemented by virtual circuits, using an explicit time division multiplexing mechanism which they call *Temporally Disjoint Networks* (TDN) (refer to Sections 3.2.2 and 3.2.3 for more details).

In Jantsch and Vitkowski [2005], the group addressed encoding issues and showed that lowering the voltage swing, then reestablishing reliability using error correction, actually resulted in better power saving than a number of dedicated power saving algorithms used for comparison.

- (3) **SPIN.** The SPIN network (*Scalable Programmable Integrated Network*) [Guerrier and Greiner 2000; Andriahantaina and Greiner 2003] implements a fat-tree topology with two one-way 32-bit datapaths at the link layer. The fat tree is an interesting choice of irregular network claimed in Leiserson [1985] to be “nearly the best routing network for a given amount of hardware.” It is proven that, for any given amount of hardware, a fat tree can simulate any other network built from the same amount of hardware with only a polylogarithmic slowdown in latency. This is in contrast to, for example, two-dimensional arrays or simple trees which exhibit polynomial slowdown when simulating other networks and, as such, do not have any advantage over a sequential computer.

In SPIN, packets are sent via the network as a sequence of flits each of size 4 bytes. Wormhole routing is used with no limit on packet size. The first flit contains the header, with one byte reserved for addressing, and the last byte of the packet contains the payload checksum. There are three types of flits; first, data, and last. Link-level flow control is used to identify the flit type and act accordingly upon its content. The additional bytes in the header can be used for packet tagging for special services and for special routing options. The performance of the network was evaluated primarily based on uniform randomly distributed load (see Section 5). It was noted that random hick-ups can be expected under high load. It was found that the protocol accounts for about 31% of the total throughput, a relatively large overhead. In 2003, a 32-port SPIN network was implemented in a 0.13 μm CMOS process, the total area was 4.6 mm^2 (0.144 mm^2 per port), for an accumulated bandwidth of about 100 Gbits/s.

- (4) **CHAIN.** The CHAIN network (*CHip Area INterconnect*) [Bainbridge and Furber 2002], developed at the University of Manchester, is interesting in that it is implemented entirely using asynchronous, or clockless, circuit techniques. It makes use of delay insensitive 1-of-4 encoding, and source routes BE packets. An easy adaption along a path consisting of links of different bit widths is supported. CHAIN is targeted for heterogeneous low-power systems in which the network is system specific. It has been implemented in a smart card which benefits from the low idle power capabilities of asynchronous circuits. Work from the group involved with CHAIN concerns prioritization in asynchronous networks. In Felicijan et al. [2003], an asynchronous low-latency arbiter was presented, and its use in providing differentiated communication services in SoC was discussed, and in Felicijan and Furber [2004], a router implementing the scheme was described.

- (5) **MANGO.** The MANGO network (*Message-passing Asynchronous Network-on-chip providing Guaranteed services over OCP interfaces*), developed at the Technical University of Denmark, is another clockless NoC, targeted for coarse-grained GALS-type SoC [Bjerregaard 2005]. MANGO provides connectionless BE routing as well as connection-oriented guaranteed services (GS) [Bjerregaard and Sparsø 2005a]. In order to make for a simple design, the routers implement virtual channels (VCs) as separate physical buffers. GS connections are established by allocating a sequence of VCs through the network. While the routers themselves are implemented using area efficient bundled-data circuits, the links implement delay insensitive signal encoding. This makes global timing robust because no timing assumptions are necessary between routers. A scheduling scheme called ALG (*Asynchronous Latency*

Guarantees) [Bjerregaard and Sparsø 2005b] schedules access to the links, allowing latency guarantees to be made which are not inversely dependent on the bandwidth guarantees as is the case in TDM-based scheduling schemes. Network adapters provide OCP-based standard socket interfaces based on the primitive routing services of the network [Bjerregaard et al. 2005]. This includes support for interrupts based on virtual wires. The adapters also synchronize the clocked OCP interfaces to the clockless network.

- (6) **XPIPES.** xpipes [Osso et al. 2003] and the accompanying NetChip compiler (a combination of xpipesCompiler [Jalabert et al. 2004] and SUNMAP [Murali and Micheli 2004b]) were developed by the University of Bologna and Stanford University. xpipes consists of soft macros of switches and links that can be turned into instance-specific network components at instantiation time. It promotes the idea of pipelined links with a flexible number of stages to increase throughput. A go-back-N retransmission strategy is implemented as part of link-level error control which reduces switch complexity, though at considerable delay since each flit is not acknowledged until it has been transmitted across the destination switch. The error is indicated by a CRC block running concurrently with switch operation. Thus the xpipes architecture lends itself to be robust to interconnect errors. Overall, delay for a flit to traverse from across one link and node is $2N + M$ cycles, where N is number of pipeline stages and M the switch stages. The xpipesCompiler is a tool to automatically instantiate an application-specific custom communication infrastructure using xpipes components. It can tune flit size, degree of redundancy of the CRC error-detection, address space of cores, number of bits used for packet sequence count, maximum number of hops between any two network nodes, number of flit size, etc.

In a top-down design methodology, once the SoC floorplan is decided, the required network architecture is fed into the xpipesCompiler. Examples of compiler optimization include removing redundant buffers from missing output ports of switches, sharing signals common to objects, etc. Via case studies presented in Bertozzi et al. [2005], the NetChip compiler has been validated for mesh, torus, hypercube, Clos, and butterfly NoC topologies for four video processing applications. Four routing algorithms, dimension-ordered, minimum-path, traffic splitting across minimum-path, and traffic splitting across all paths, is also part of the case study experiments. The floorplan of switches and links of NoC takes the IP block size into consideration. Results are available for average hop delay, area and power for mapping of each of the video application on the topologies. A lightweight implementation, named xpipes-lite, presented in Stergiou et al. [2005], is similar in to xpipes in concept, but is optimized for link latency, area and power, and provides direct synthesis path from SystemC description.

7. SUMMARY

NoC encompasses a wide spectrum of research, ranging from highly abstract software related issues, across system topology to physical level implementation. In this survey, we have given an overview of activities in the field. We have first stated the motivation for NoC and given an introduction of the basic concepts. In order to avoid the wide range of topics relevant to large scale IC design in general, we have assumed a view of NoC as a subset of SoC.

From a system-level perspective, NoC is motivated by the demand for a well structured design approach in large scale SoCs. A modularized design methodology is needed in order to make efficient use of the ever increasing availability of on-chip resources

in terms of the number of transistors and wiring layers. Likewise, programming these systems necessitates clear programming models and predictable behavior. NoC has the potential to provide modularity through the use of standard sockets such as OCP and predictability through the implementation of guaranteed communication services. From a physical-level perspective, with scaling of technologies into the DSM region, the increasing impact of wires on performance forces a differentiation between local and global communication. In order for global communication structures to exhibit scalability and high performance, segmentation, wire sharing, and distributed control is required.

In structuring our work, we have adopted a layered approach similar to OSI and divided NoC research into four areas: system, network adapter, network and link research. In accordance with the view of NoC as a subset of SoC, we have dealt first with the latter three areas of research which relate directly to the NoC implementation. Thereafter, we have focused on system-level aspects.

The network adapter orthogonalizes communication and computation, enabling communication-centric design. It is thus the entity which enables a modularized design approach. Its main task is to decouple the core from the network with the purpose of providing high-level network-agnostic communication services based on the low-level routing primitives provided by the network hardware. In implementing standard sockets, IP reuse becomes feasible, and the network adapter may, therefore, hold the key to the commercial success of NoC.

At the network level, issues such as network topology, routing protocols, flow control, and quality-of-service are dominant. With regards to topology, NoC is restricted by a 2D layout. This has made grid-type topologies a widespread choice. We have reviewed the most common routing schemes, store-and-forward, wormhole and virtual cut-through routing, and concluded that wormhole routing is by far the most common choice for NoC designs. The use of virtual channels in avoiding deadlocks and providing guaranteed services was illustrated, and the motivation for guaranteed services was discussed. The predictability that such services incur facilitates easy system integration and analytical system verification, particularly relevant for real-time systems.

Unlike macronetworks, in NoC, the network adapter and network functionality is often implemented in hardware rather than in software. This is so because NoC-based systems are more tightly bound, and simple, fast, power-efficient solutions are required.

Link-level research is much more hardware oriented. We have covered topics like synchronization, that is, between clock domains, segmentation, and pipelining of links, in order to increase bandwidth and counteract the physical limitations of DSM technologies, on-chip signaling such as low-swing drivers used to decrease the power usage in links, and future technologies such as on-chip wave guides, and optical interconnects. We have also discussed the reliability of long links, which are susceptible to a number of noise sources: crosstalk, ground bounce, EMI and intersymbol interference. Segmentation helps keep the effect of these at bay since the shorter a wire is, the less influence they will have. Error detection and correction in on-chip interconnects was discussed, but this is not a dominating area of research. Different encoding schemes were discussed in relation to increasing bandwidth as well as reducing power consumption.

NoC facilitates communication-centric design as opposed to traditional computation-centric design. From a system-level perspective, we have addressed topics relating to the role of NoC in SoC design flows. Key issues are modeling, design methodology, and traffic characterization. The purpose of modeling is to evaluate trade-offs with regard to global traffic in terms of power, area, design time, etc., while adhering to application requirements. With regard to design methodology, we identify two important characteristics of NoC, by which we classify a number of existing NoC solutions:

(i) parametrizability of the NoC as a system level block and (ii) granularity of the NoC components by which the NoC is assembled. These characteristics greatly influence the nature of the design flow enabled by the particular NoC. As a tool for identifying general requirements of a NoC, we have identified a set of traffic types, latency-critical, data-streams and miscellaneous traffic, which span the spectrum of possible traffic in a NoC-based system.

The basic performance parameters of NoC are latency, bandwidth and jitter. The basic cost factors are power consumption and area usage. At a higher level of abstraction, terms like aggregate bandwidth, bisection bandwidth, link utilization and network load can be used. These originate in multicomputer network theory and relate to data movement in general. Stepping up yet another abstraction level, benchmarks can be used for performance analysis. Currently no benchmarks exist specifically for NoC, but the use of benchmarks for parallel computers, as well as embedded systems benchmarks, has been reported.

Six case studies are conducted, explaining the design choices of the *ÆTHEREAL*, *NOSTRUM*, *SPIN*, *CHAIN*, *MANGO* and *XPIPES* NoC implementations. *CHAIN* and *XPIPES* target a platform-based design methodology in which a heterogeneous network can be instantiated for a particular application. *ÆTHEREAL*, *NOSTRUM*, and *MANGO* implement more complex features such as guaranteed services, and target a methodology which draws closer to backbone-based design. *SPIN* differs from the others in that it implements a fat tree rather than a grid-type topology. *CHAIN* and *MANGO* also differ in that they are implemented entirely using clockless circuit techniques and, as such, inherently support globally asynchronous and locally synchronous (GALS) systems.

Continued technology scaling enables large scale SoC. NoCs facilitate a modular, scalable design approach that overcomes both system and physical-level issues. The main job of the NoC designer of the future will be to dimension and structure the network according to the communication needs of the SoC. At present, an interesting challenge lies in specifying ways to define these needs.

ACKNOWLEDGMENTS

We would like to thank professors Jens Sparsø and Jan Madsen of the Department for Informatics and Mathematical Modelling (IMM) at the Technical University of Denmark (DTU) for their tireless effort in helping us review, iterate, and structure this survey. Also our grateful thanks to professor Axel Jantsch (KTH—Stockholm, Sweden) and Andrei Radulescu (Phillips—Eindhoven, Netherlands) for their valuable review of the survey as it was closing in on its final form, and to Mihai Budiu (Carnegie Mellon University—Pittsburgh, USA) for comments and suggestions. Finally, the extensive comments of the anonymous reviewers have helped in taking the survey to its final form.

REFERENCES

- AGARWAL, A. 1999. The Oxygen project—Raw computation. *Scientific American*, 44–47.
- AGGARWAL, A. AND FRANKLIN, M. 2002. Hierarchical interconnects for on-chip clustering. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 602–609.
- AHONEN, T., SIGENZA-TORTOSA, D. A., BIN, H., AND NURMI, J. 2004. Topology optimization for application-specific networks-on-chip. In *International Workshop on System Level Interconnect Prediction (SLIP)*. ACM, 53–60.
- AL-TAWIL, K. M., ABD-EL-BARR, M., AND ASHRAF, F. 1997. A survey and comparison of wormhole routing techniques in a mesh networks. *IEEE Network* 11, 38–45.
- AMDE, M., FELICIANI, T., EDWARDS, A. E. D., AND LAVAGNO, L. 2005. Asynchronous on-chip networks. *IEE Proceedings of Computers and Digital Techniques* 152, 273–283.

- ANDREASSON, D. AND KUMAR, S. 2004. On improving best-effort throughput by better utilization of guaranteed-throughput channels in an on-chip communication system. In *Proceeding of 22th IEEE Norchip Conference*.
- ANDREASSON, D. AND KUMAR, S. 2005. Slack-time aware routing in NoC systems. In *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2353–2356.
- ANDRIAHANTENAINA, A. AND GREINER, A. 2003. Micro-network for SoC: Implementation of a 32-port spin network. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 1128–1129.
- ARM. 2004. AMBA Advanced eXtensible Interface (AXI) Protocol Specification, Version 1.0. <http://www.arm.com>.
- ARTERIS. 2005. A comparison of network-on-chip and busses. White paper. http://www.arteris.com/noc_whitepaper.pdf.
- BAILEY, D., BARSZCZ, E., BARTON, J., BROWNING, D., CARTER, R., DAGUM, L., FATOOGHI, R., FINEBERG, S., FREDERICKSON, P., LASINSKI, T., SCHREIBER, R., SIMON, H., VENKATAKRISHNAN, V., AND WEERATUNGA, S. 1994. RNR Tech. rep. RNR-94-007. NASA Ames Research Center.
- BAINBRIDGE, J. AND FURBER, S. 2002. CHAIN: A delay-insensitive chip area interconnect. *IEEE Micro* 22, 5 (Oct.) 16–23.
- BAINBRIDGE, W. AND FURBER, S. 2001. Delay insensitive system-on-chip interconnect using 1-of-4 data encoding. In *Proceedings of the 7th International Symposium on Asynchronous Circuits and Systems (ASYNC)*. 118–126.
- BANERJEE, N., VELLANKI, P., AND CHATHA, K. S. 2004. A power and performance model for network-on-chip architectures. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*. IEEE, 1250–1255.
- BEIGNE, E., CLERMIDY, F., VIVET, P., CLOUARD, A., AND REAUDIN, M. 2005. An asynchronous NOC architecture providing low latency service and its multi-level design framework. In *Proceedings of the 11th International Symposium on Asynchronous Circuits and Systems (ASYNC)*. IEEE, 54–63.
- BENINI, L. AND MICHELI, G. D. 2001. Powering network-on-chips. In *The 14th International Symposium on System Synthesis (ISSS)*. IEEE, 33–38.
- BENINI, L. AND MICHELI, G. D. 2002. Networks on chips: A new SoC paradigm. *IEEE Comput.* 35, 1 (Jan.), 70–78.
- BERTOZZI, D., JALABERT, A., MURALI, S., TAMHANKAR, R., STERGIOU, S., BENINI, L., AND DE MICHELI, G. 2005. NoC synthesis flow for customized domain specific multiprocessor Systems-on-Chip. In *IEEE Trans. Parall. Distrib. Syst.* 113–129.
- BHOJWANI, P. AND MAHAPATRA, R. 2003. Interfacing cores with on-chip packet-switched networks. In *Proceedings of the 16th International Conference on VLSI Design*. 382–387.
- BJERREGAARD, T. 2005. The MANGO clockless network-on-chip: Concepts and implementation. Ph.D. thesis, Informatics and Mathematical Modeling, Technical University of Denmark, Lyngby, Denmark.
- BJERREGAARD, T., MAHADEVAN, S., OLSEN, R. G., AND SPARSØ, J. 2005. An OCP compliant network adapter for gals-based soc design using the MANGO network-on-chip. In *Proceedings of International Symposium on System-on-Chip (ISSoC)*. IEEE.
- BJERREGAARD, T., MAHADEVAN, S., AND SPARSØ, J. 2004. A channel library for asynchronous circuit design supporting mixed-mode modeling. In *Proceedings of the 14th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. Springer, 301–310.
- BJERREGAARD, T. AND SPARSØ, J. 2005a. A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*. IEEE, 1226–1231.
- BJERREGAARD, T. AND SPARSØ, J. 2005b. A scheduling discipline for latency and bandwidth guarantees in asynchronous network-on-chip. In *Proceedings of the 11th International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE, 34–43.
- BOGLIOLO, A. 2001. Encodings for high-performance energy-efficient signaling. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*. 170–175.
- BOLOTIN, E., CIDON, I., GINOSAUR, R., AND KOLODNY, A. 2004. QNoC: QoS architecture and design process for network-on-chip. *J. Syst. Archit.* 50, 2-3, 105–128.
- CATTHOOR, F., CUOMO, A., MARTIN, G., GROENEVELD, P., RUDY, L., MAEX, K., DE STEEG, P. V., AND WILSON, R. 2004. How can system level design solve the interconnect technology scaling problem. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*. IEEE, 332–337.
- CHAPIRO, D. 1984. Globally-asynchronous locally-synchronous systems. Ph.D. thesis (Report No. STAN-CS-84-1026) Stanford University.

- CHELCEA, T. AND NOWICK, S. M. 2001. Robust interfaces for mixed-timing systems with application to latency-insensitive protocols. In *Proceedings of the 38th Design Automation Conference (DAC)*. IEEE, 21–26.
- CHIU, G.-M. 2000. The odd-even turn model for adaptive routing. *IEEE Trans. Parall. Distrib. Syst.* 11, 729–738.
- COLE, R. J., MAGGS, B. M., AND SITARAMAN, R. K. 2001. On the benefit of supporting virtual channels in wormhole routers. *J. Comput. Syst. Sciences* 62, 152–177.
- CULLER, D. E., SINGH, J. P., AND GUPTA, A. 1998. *Parallel Computer Architecture: A Hardware / Software Approach*. 1st Ed. Morgan Kaufmann.
- DALLY, W. J. 1990. Performance analysis of k-ary n-cube interconnection networks. *IEEE Trans. Comput.* 39, 6 (June) 775–785.
- DALLY, W. J. 1992. Virtual-channel flow control. *IEEE Trans. Parall. Distrib. Syst.* 3, 2 (March) 194–205.
- DALLY, W. J. AND AOKI, H. 1993. Deadlock-free adaptive routing in multicomputer networks using virtual channels. *IEEE Trans. Parall. Distrib. Syst.* 4, 4 (April) 466–475.
- DALLY, W. J. AND SEITZ, C. L. 1987. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. Comput.* 36, 5 (May) 547–553.
- DALLY, W. J. AND TOWLES, B. 2001. Route packets, not wires: On-chip interconnection networks. In *Proceedings of the 38th Design Automation Conference (DAC)*. IEEE, 684–689.
- DE MELLO, A. V., OST, L. C., MORAES, F. G., AND CALAZANS, N. L. V. 2004. Evaluation of routing algorithms on mesh based noc's. Tech. rep., Faculdade de Informatica PUCRS—Brazil.
- DICK, R. 2002. Embedded system synthesis benchmarks suite. <http://www.ece.northwestern.edu/dickrp/e3s/>.
- DIELISSEN, J., RADULESCU, A., GOOSSENS, K., AND RUIJPKEMA, E. 2003. Concepts and implementation of the phillips network-on-chip. In *Proceedings of the IP based SOC (IPSOC)*. IFIP.
- DOBBELAERE, I., HOROWITZ, M., AND GAMAL, A. E. 1995. Regenerative feedback repeaters for programmable interconnections. *IEEE J. Solid-State Circuits* 30, 11 (Nov.) 1246–1253.
- DOBKIN, R., GINOSAUR, R., AND SOTIRIOU, C. P. 2004. Data synchronization issues in GALS SoCs. In *Proceedings of the 10th IEEE International Symposium on Asynchronous Circuits and Systems*. IEEE, 170–179.
- DUATO, J. 1993. A new theory of deadlock-free adaptive routing in wormhole networks. *IEEE Trans. Parall. Distrib. Syst.* 4, 12 (Dec.) 1320–1331.
- DUATO, J. 1995. A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks. *IEEE Trans. Parall. Distrib. Syst.* 6, 10 (Oct.) 1055–1067.
- DUATO, J. 1996. A necessary and sufficient condition for deadlock-free routing in cut-through and store-and-forward networks. *IEEE Trans. Parall. Distrib. Syst.* 7, 8 (Aug.) 841–854.
- DUATO, J. AND PINKSTON, T. M. 2001. A general theory for deadlock-free adaptive routing using a mixed set of resources. *IEEE Trans. Parall. Distrib. Syst.* 12, 12 (Dec.) 1219–1235.
- DUATO, J., YALAMANCHILI, S., AND NI, L. 2003. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann.
- FELICJAN, T., BAINBRIDGE, J., AND FURBER, S. 2003. An asynchronous low latency arbiter for quality of service (QoS) applications. In *Proceedings of the 15th International Conference on Microelectronics (ICM)*. IEEE, 123–126.
- FELICJAN, T. AND FURBER, S. B. 2004. An asynchronous on-chip network router with quality-of-service (QoS) support. In *Proceedings IEEE International SOC Conference*. IEEE, 274–277.
- FITZPATRICK, T. 2004. System verilog for VHDL users. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*. IEEE Computer Society, 21334.
- FORSELL, M. 2002. A scalable high-performance computing solution for networks on chips. *IEEE Micro* 22, 5, 46–55.
- GAUGHAN, P. T., DAO, B. V., YALAMANCHILI, S., AND SCHIMMEL, D. E. 1996. Distributed, deadlock-free routing in faulty, pipelined, direct interconnection networks. *IEEE Trans. Comput.* 45, 6 (June) 651–665.
- GENKO, N., ATIENZA, D., DE MICHELI, G., BENINI, L., MENDIAS, J., HERMIDA, R., AND CATTHOOR, F. 2005. A novel approach for network on chip emulation. In *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2365–2368.
- GERSTLAUER, A. 2003. Communication abstractions for system-level design and synthesis. Tech. Rep. TR-03-30, Center for Embedded Computer Systems, University of California, Irvine, CA.
- GINOSAUR, R. 2003. Fourteen ways to fool your synchronizer. In *Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems*. IEEE, 89–96.
- GLASS, C. J. AND NI, L. M. 1994. The turn model for adaptive routing. *J. ACM* 41, 874–902.

- GOOSSENS, K., DIELISSEN, J., GANGWAL, O. P., PESTANA, S. G., RADULESCU, A., AND RIJPKEMA, E. 2005. A design flow for application-specific networks on chip with guaranteed performance to accelerate SOC design and verification. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*. IEEE, 1182–1187.
- GOOSSENS, K., DIELISSEN, J., AND RADULESCU, A. 2005. λ Ethereal network on chip: Concepts, architectures and implementations. *IEEE Design Test Comput.* 22, 5, 414–421.
- GOOSSENS, K., MEERBERGEN, J. V., PEETERS, A., AND WIELAGE, P. 2002. Networks on silicon: Combining best-effort and guaranteed services. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*. IEEE, 196–200.
- GUERRIER, P. AND GREINER, A. 2000. A generic architecture for on-chip packet-switched interconnections. In *Proceedings of the Design Automation and Test in Europe (DATE)*. IEEE, 250–256.
- GUO, M., NAKATA, I., AND YAMASHITA, Y. 2000. Contention-free communication scheduling for array redistribution. *Parall. Comput.* 26, 1325–1343.
- HANSSON, A., GOOSSENS, K., AND RADULESCU, A. 2005. A unified approach to constrained mapping and routing on networks-on-chip architectures. In *CODES/ISSS*. ACM/IEEE, 75–80.
- HARMANCI, M., ESCUDERO, N., LEBLEBICI, Y., AND IENNE, P. 2005. Quantitative modeling and comparison of communication schemes to guarantee quality-of-service in networks-on-chip. In *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1782–1785.
- HAUCK, S. 1995. Asynchronous design methodologies: an overview. *Proceedings of the IEEE* 83, 1 (Jan.) 69–93.
- HAVEMANN, R. H. AND HUTCHBY, J. A. 2001. High-performance interconnects: An integration overview. *Proceedings of the IEEE* 89, 5 (May) 586–601.
- HAVERINEN, A., LECLERCQ, M., WEYRICH, N., AND WINGARD, D. 2002. SystemC based SoC communication modeling for the OCP protocol. White paper. <http://www.ocpi.org>.
- HEILIGER, H.-M., NAGEL, M., ROSKOS, H. G., AND KURZ, H. 1997. Thin-film microstrip lines for mm and sub-mm-wave on-chip interconnects. In *IEEE MTT-S Int. Microwave Symp. Digest*. Vol. 2. 421–424.
- HO, R., MAI, K., AND HOROWITZ, M. 2003. Efficient on-chip global interconnects. In *Symposium on VLSI Circuits. Digest of Technical Papers*. IEEE, 271–274.
- HO, R., MAI, K. W., AND HOROWITZ, M. A. 2001. The future of wires. *Proceedings of the IEEE* 89, 4 (April) 490–504.
- HU, J. AND MARCULESCU, R. 2004a. Application-specific buffer space allocation for networks-on-chip router design. In *ICCAD*. IEEE/ACM, 354–361.
- HU, J. AND MARCULESCU, R. 2004b. Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*. IEEE, 10234–10240.
- ITRS. 2001. International technology roadmap for semiconductors. Tech. rep., International Technology Roadmap for Semiconductors.
- ITRS. 2003. International technology roadmap for semiconductors. Tech. rep., International Technology Roadmap for Semiconductors.
- JALABERT, A., MURALI, S., BENINI, L., AND MICHELI, G. D. 2004. xpipesCompiler: A tool for instantiating application specific networks-on-chip. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*. IEEE, 884–889.
- JANTSCH, A. 2003. Communication performance in networks-on-chip. <http://www.ele.kth.se/axel/presentations/2003/Stringent.pdf>.
- JANTSCH, A. AND TENHUNEN, H. 2003. *Networks on Chip*. Kluwer Academic Publishers.
- JANTSCH, A. AND VITKOWSKI, R. L. A. 2005. Power analysis of link level and end-to-end data protection in networks-on-chip. In *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1770–1773.
- JUURLINK, B. H. H. AND WIJSHOFF, H. A. G. 1998. A quantitative comparison of parallel computation models. *ACM Trans. Comput. Syst.* 16, 3 (Aug.) 271–318.
- KAPUR, P. AND SARASWAT, K. C. 2003. Optical interconnects for future high performance integrated circuits. *Physica E* 16, 3–4, 620–627.
- KARIM, F., NGUYEN, A., AND DEY, S. 2002. An interconnect architecture for networking systems on chips. *IEEE Micro* 22, 36–45.
- KARIM, F., NGUYEN, A., DEY, S., AND RAO, R. 2001. On-chip communication architecture for OC-768 network processors. In *Proceedings of the 38th Design Automation Conference (DAC)*. ACM, 678–683.
- KIM, D., LEE, K., JOONG LEE, S., AND YOO, H.-J. 2005. A reconfigurable crossbar switch with adaptive bandwidth control for networks-on-chip. In *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2369–2372.

- KIM, K., LEE, S.-J., LEE, K., AND YOO, H.-J. 2005. An arbitration look-ahead scheme for reducing end-to-end latency in networks-on-chip. In *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2357–2360.
- KUMAR, S., JANTSCH, A., SOININEN, J.-P., FORSELL, M., MILLBERG, M., OBERG, J., TIENSYRJÄ, K., AND HEMANI, A. 2002. A network-on-chip architecture and design methodology. In *Proceedings of the Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE Computer Society, 117–124.
- KURD, N., BARKATULLAH, J., DIZON, R., FLETCHER, T., AND MADLAND, P. 2001. Multi-GHz clocking scheme for Intel pentium 4 microprocessor. In *Digest of Technical Papers. International Solid-State Circuits Conference (ISSCC)*. IEEE, 404–405.
- LAHIRI, K., RAGHUNATHAN, A., AND DEY, S. 2001. Evaluation of the traffic-performance characteristics of system-on-chip communication architectures. In *Proceedings of the 14th International Conference on VLSI Design*. IEEE, 29–35.
- LAHIRI, K., RAGHUNATHAN, A., LAKSHMINARAYANA, G., AND DEY, S. 2000. Communication architecture tuners: A methodology for the design of high-performance communication architectures for system-on-chips. In *Proceedings of the Design Automation Conference, DAC*. IEEE, 513–518.
- LEE, K. 1998. On-chip interconnects—gigahertz and beyond. *Solid State Technol.* 41, 9 (Sept.) 85–89.
- LEISERSON, C. E. 1985. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.* c-34, 10, 892–901.
- LEROY, A., MARCHAL, P., SHICKOVA, A., CATTHOOR, F., ROBERT, F., AND VERKEST, D. 2005. Spatial division multiplexing: a novel approach for guaranteed throughput on nocs. In *CODES/ISSS*. ACM/IEEE, 81–86.
- LIANG, J., LAFFELY, A., SRINIVASAN, S., AND TESSIER, R. 2004. An architecture and compiler for scalable on-chip communication. *IEEE Trans. VLSI Syst.* 12, 7, 711–726.
- LIANG, J., SWAMINATHAN, S., AND TESSIER, R. 2000. ASOC: A scalable, single-chip communications architecture. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 37–46.
- LIU, J., ZHENG, L.-R., AND TENHUNEN, H. 2004. Interconnect intellectual property for network-on-chip (NoC). *J. Syst. Archite.* 50, 65–79.
- LOGHI, M., ANGIOLINI, F., BERTOZZI, D., BENINI, L., AND ZAFALON, R. 2004. Analyzing on-chip communication in a MPSoC environment. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*. IEEE, 752–757.
- MADSEN, J., MAHADEVAN, S., VIRK, K., AND GONZALEZ, M. 2003. Network-on-chip modeling for system-level multiprocessor simulation. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS)*. IEEE, 82–92.
- MAHADEVAN, S., STORGAARD, M., MADSEN, J., AND VIRK, K. 2005. ARTS: A system-level framework for modeling MPSoC components and analysis of their causality. In *The 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE Computer Society.
- MAI, K., PAASKE, T., JAYASENA, N., HO, R., DALLY, W. J., AND HOROWITZ, M. 2000. Smart memories: A modular reconfigurable architecture. In *Proceedings of 27th International Symposium on Computer Architecture*. 161–171.
- MEINCKE, T., HEMANI, A., KUMAR, S., ELLERVEE, P., OBERG, J., OLSSON, T., NILSSON, P., LINDQVIST, D., AND TENHUNEN, H. 1999. Globally asynchronous locally synchronous architecture for large high-performance ASICs. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*. Vol. 2. 512–515.
- MILLBERG, M., NILSSON, E., THID, R., AND JANTSCH, A. 2004. Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network-on-chip. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*. IEEE, 890–895.
- MIZUNO, M., DALLY, W. J., AND ONISHI, H. 2001. Elastic interconnects: Repeater-inserted long wiring capable of compressing and decompressing data. In *Proceedings of the International Solid-State Circuits Conference*. IEEE, 346–347, 464.
- MORAES, F., CALAZANS, N., MELLO, A., MÖLLER, L., AND OST, L. 2004. HERMES: An infrastructure for low area overhead packet-switching networks on chip. *The VLSI Integration* 38, 69–93.
- MULLINS, R. AND MOORE, A. W. S. 2004. Low-latency virtual-channel routers for on-chip networks. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*. IEEE, 188–197.
- MURALI, S. AND MICHELI, G. D. 2004a. Bandwidth-constrained mapping of cores onto noc architectures. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*. IEEE, 20896–20902.
- MURALI, S. AND MICHELI, G. D. 2004b. SUNMAP: A tool for automatic topology selection and generation for NoCs. In *In Proceedings of the 41st Design Automation Conference (DAC)*. IEEE, 914–919.

- MUTTERSBACH, J., VILLIGER, T., AND FICHTNER, W. 2000. Practical design of globally-asynchronous locally-synchronous systems. In *Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*. IEEE Computer Society, 52–59.
- NAKAMURA, K. AND HOROWITZ, M. A. 1996. A 50% noise reduction interface using low-weight coding. In *Symposium on VLSI Circuits Digest of Technical Papers*. IEEE, 144–145.
- NEDOVIC, N., OKLOBDZIJA, V. G., AND WALKER, W. W. 2003. A clock skew absorbing flip-flop. In *Proceedings of the International Solid-State Circuits Conference*. IEEE, 342–497.
- NEEB, C., THUL, M., WEHN, N., NEEB, C., THUL, M., AND WEHN, N. 2005. Network-on-chip-centric approach to interleaving in high throughput channel decoders. In *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1766–1769.
- NIELSEN, S. F. AND SPARSØ, J. 2001. Analysis of low-power SoC interconnection networks. In *Proceedings of Nordchip 2001*. 77–86.
- OBERG, J. 2003. *Clocking Strategies for Networks-on-Chip*. Kluwer Academic Publishers, 153–172.
- OCPIP. 2003a. The importance of sockets in SoC design. White paper. <http://www.ocpip.org>.
- OCPIP. 2003b. Open Core Protocol (OCP) Specification, Release 2.0. <http://www.ocpip.org>.
- OKLOBDZIJA, V. G. AND SPARSØ, J. 2002. Future directions in clocking multi-GHz systems. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design, 2002 (ISLPED '02)*. ACM, 219.
- OSO, M. D., BICCARI, G., GIOVANNINI, L., BERTOZZI, D., AND BENINI, L. 2003. Xpipes: a latency insensitive parameterized network-on-chip architecture for multi-processor SoCs. In *Proceedings of 21st International Conference on Computer Design (ICCD)*. IEEE Computer Society, 536–539.
- OST, L., MELLO, A., PALMA, J., MORAES, F., AND CALAZANS, N. 2005. MAIA—a framework for networks on chip generation and verification. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE.
- PANDE, P., GRECU, C., JONES, M., IVANOV, A., AND SALEH, R. 2005. Effect of traffic localization on energy dissipation in NoC-based interconnect. In *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1774–1777.
- PANDE, P. P., GRECU, C., IVANOV, A., AND SALEH, R. 2003. Design of a switch for network-on-chip applications. In *IEEE International Symposium on Circuits and Systems (ISCAS) 5*, 217–220.
- PEH, L.-S. AND DALLY, W. J. 1999. Flit-reservation flow control. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, 73–84.
- PEH, L.-S. AND DALLY, W. J. 2001. A delay model for router microarchitectures. *IEEE Micro 21*, 26–34.
- PESTANA, S., RIJPKEMA, E., RADULESCU, A., GOOSSENS, K., AND GANGWAL, O. 2004. Cost-performance trade-offs in networks on chip: a simulation-based approach. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*. IEEE, 764–769.
- PHILIPS SEMICONDUCTORS. 2002. Device Transaction Level (DTL) Protocol Specification, Version 2.2.
- PIGUET, C., JACQUES, HEER, C., O'CONNOR, I., AND SCHLICHTMANN, U. 2004. Extremely low-power logic. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*, C. Piguet, Ed. IEEE, 1530–1591.
- PIRRETTI, M., LINK, G., BROOKS, R. R., VIJAYKRISHNAN, N., KANDEMIR, M., AND IRWIN, M. 2004. Fault tolerant algorithms for network-on-chip interconnect. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*. 46–51.
- RADULESCU, A., DIELISSEN, J., GOOSSENS, K., RIJPKEMA, E., AND WIELAGE, P. 2004. An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network configuration. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*. IEEE, 878–883.
- RIJPKEMA, E., GOOSSENS, K., AND WIELAGE, P. 2001. A router architecture for networks on silicon. In *Proceeding of the 2nd Workshop on Embedded Systems*. 181–188.
- RIJPKEMA, E., GOOSSENS, K. G. W., RADULESCU, A., DIELISSEN, J., MEERBERGEN, J. V., WIELAGE, P., AND WATERLANDER, E. 2003. Trade-offs in the design of a router with both guaranteed and best-effort services for networks-on-chip. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*. IEEE, 350–355.
- RIXNER, S., DALLY, W. J., KAPASI, U. J., KHAILANY, B., LUPEZ-LAGUNAS, A., MATTISON, P. R., AND OWENS, J. D. 1998. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*. 3–13.
- ROSTISLAV, D., VISHNYAKOV, V., FRIEDMAN, E., AND GINOSAUR, R. 2005. An asynchronous router for multiple service levels networks on chip. In *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*. IEEE, 44–53.
- SATHE, S., WIKLUND, D., AND LIU, D. 2003. Design of a switching node (router) for on-chip networks. In *Proceedings of the 5th International Conference on ASIC*. IEEE, 75–78.

- SIA. 1997. National technology roadmap for semiconductors 1997. Tech. rep., Semiconductor Industry Association.
- SIGUENZA-TORTOSA, D., AHONEN, T., AND NURMI, J. 2004. Issues in the development of a practical NoC: The Proteo concept. *Integrat. VLSI J.* Elsevier, 95–105.
- SIMUNIC, T. AND BOYD, S. 2002. Managing power consumption in networks-on-chips. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*. IEEE Computer Society, 110–116.
- SINGH, M. AND NOWICK, S. 2000. High-throughput asynchronous pipelines for fine-grain dynamic datapaths. In *Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*. IEEE Computer Society, 198–209.
- SPARSO, J. AND FURBER, S. 2001. *Principles of Asynchronous Circuit Design*. Kluwer Academic Publishers, Boston, MA.
- STERGIOU, S., ANGIOLINI, F., CARTA, S., RAFFO, L., BERTOZZI, D., AND MICHELI, G. D. 2005. xpipes lite: A synthesis oriented design library for networks on chips. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*. IEEE.
- SVENSSON, C. 2001. Optimum voltage swing on on-chip and off-chip interconnect. Manuscript available at <http://www.elk.isy.liu.se/~christer/ManuscriptSwing.pdf>.
- SYLVESTER, D. AND KEUTZER, K. 2000. A global wiring paradigm for deep submicron design. *IEEE Trans. Comput. Aided Design Integrat. Circuits Syst.* 19, 242–252.
- SYSTEMC. 2002. The SystemC Version 2.0.1. Web Forum (www.systemc.org).
- TAMIR, Y. AND FRAZIER, G. L. 1988. High-performance multiqueue buffers for VLSI communication switches. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*. IEEE Computer Society, 343–354.
- TAYLOR, M. B., KIM, J., MILLER, J., WENTZLAFF, D., GHODRAT, F., GREENWALD, B., HOFFMAN, H., JOHNSON, P., LEE, J.-W., LEE, W., MA, A., SARAF, A., SENESKI, M., SHNIDMAN, N., STRUMPFEN, V., FRANK, M., AMARASINGHE, S., AND AGARWAL, A. 2002. The RAW microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE MICRO* 12, 2, 25–35.
- TORTOSA, D. A. AND NURMI, J. 2004. Packet scheduling in proteo network-on-chip. *Parall. Distrib. Comput. Netw.* IASTED/ACTA Press, 116–121.
- VAIDYA, R. S., SIVASUBRAMANIAM, A., AND DAS, C. R. 2001. Impact of virtual channels and adaptive routing on application performance. *IEEE Trans. Parall. Distrib. Syst.* 12, 2 (Feb.) 223–237.
- VARATKAR, G. AND MARCULESCU, R. 2002. Traffic analysis for on-chip networks design of multimedia applications. In *Proceedings of the 39th Design Automation Conference (DAC)*. ACM, 795–800.
- VSI ALLIANCE. 2000. Virtual component interface standard Version 2. VSI Alliance www.vsi.org.
- WANG, H.-S., ZHU, X., PEH, L.-S., AND MALIK, S. 2002. Orion: A power-performance simulator for interconnection networks. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society Press, 294–305.
- WEBER, W.-D., CHOU, J., SWARBRICK, I., AND WINGARD, D. 2005. A quality-of-service mechanism for interconnection networks in system-on-chips. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*. IEEE, 1232–1237.
- WIEFERINK, A., KOGEL, T., LEUPERS, R., ASCHIED, G., MEYR, H., BRAUN, G., AND NOHL, A. 2004. A system level processor/communication co-exploration methodology for multi-processor system-on-chip platforms. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*. IEEE Computer Society, 1256–1261.
- WIELAGE, P. AND GOOSSENS, K. 2002. Networks on silicon: Blessing or nightmare? In *Proceedings of the Euromicro Symposium on Digital System Design (DSD)*. IEEE, 196–200.
- WORM, F., THIRAN, P., MICHELI, G. D., AND LENNE, P. 2005. Self-calibrating networks-on-chip. In *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2361–2364.
- XANTHOPOULOS, T., BAILEY, D., GANGWAR, A., GOWAN, M., JAIN, A., AND PREWITT, B. 2001. The design and analysis of the clock distribution network for a 1.2 GHz alpha microprocessor. In *Digest of Technical Papers, IEEE International Solid-State Circuits Conference, ISSCC*. IEEE, 402–403.
- XU, J., WOLF, W., HENKEL, J., AND CHAKRADHAR, S. 2005. A methodology for design, modeling, and analysis of networks-on-chip. In *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1778–1781.
- XU, J., WOLF, W., HENKEL, J., CHAKRADHAR, S., AND LV, T. 2004. A case study in networks-on-chip design for embedded video. In *Proceedings of Design, Automation and Testing in Europe Conference (DATE)*. IEEE, 770–775.
- ZHANG, H., GEORGE, V., AND RABAETY, J. M. 1999. Low-swing on chip signaling techniques: Effectiveness and robustness. *IEEE Trans. VLSI Syst.* 8, 3 (Aug.) 264–272.

- ZHANG, H., PRABHU, V., GEORGE, V., WAN, M., BENES, M., ABNOUS, A., AND RABAETY, J. M. 2000. A 1 V heterogeneous reconfigurable processor IC for baseband wireless applications. In *International Solid-State Circuits Conference. Digest of Technical Papers (ISSCC)*. IEEE, 68–69.
- ZIMMER, H. AND JANTSCH, A. 2003. A fault tolerant notation and error-control scheme for switch-to-switch busses in a network-on-chip. In *Proceedings of Conference on Hardware/Software Codesign and System Synthesis Conference CODES ISSS*. ACM, 188–193.

Received September 2004; revised September 2005; accepted January 2006

Design and Implementation of the Blue Gene/P Snoop Filter

Valentina Salapura, Matthias Blumrich, and Alan Gara

IBM Thomas J. Watson Research Center
Yorktown Heights, NY

Abstract

As multi-core processors evolve, coherence traffic between cores is becoming problematic, both in terms of performance and power. The negative effects of coherence (snoop) traffic can be significantly mitigated through snoop filtering. Shielding each cache with a device that can squash snoop requests for addresses known not to be in cache improves performance significantly for caches that cannot perform normal load and snoop lookups simultaneously. In addition, reducing snoop lookups yields power savings.

This paper describes the design of the Blue Gene/P snoop filters, and presents hardware measurements to demonstrate their effectiveness. The Blue Gene/P snoop filters combine stream registers and snoop caches to capture both the locality of snoop addresses and their streaming behavior. Simulations of SPLASH-2 benchmarks illustrate tradeoffs and strengths of these two techniques. Their combination is shown to be most effective, eliminating 94-99% of all snoop requests using very few stream registers and snoop cache lines. This translates into an average performance improvement of almost 20% for the NAS benchmarks running on an actual Blue Gene/P system.

1. Introduction

Over the past 20 years, ever higher operating frequencies have been the main factor driving the performance growth of single-core processors. However, further increases in operating frequencies are increasingly hard to obtain with newer generations of technology [26]. One of the main reasons is the impact of wire delays as feature sizes continue to shrink [11]. To compensate, processors have adapted increasingly sophisticated microarchitectures [13], often at the cost of inefficiencies in power/performance.

While faster transistors and faster wires are increasingly hard to obtain, the application of Dennard's CMOS scaling theory [8] is continuing to deliver improvements

in density. In response, several multi-core processors have been introduced over the past few years, such as the IBM POWER4 [1] and POWER5 [25] servers, the Intel Core Duo processors [10], the AMD Opteron quad-core processors [3], and the Cell Broadband Architecture [12]. On such multi-core processors, suitably scalable, multithreaded, parallel workloads show significant increases in performance with little or no degradation in the power/performance ratio [22].

IBM recently announced the Blue Gene/P supercomputer [14], successor to the highly successful Blue Gene/L machine [2, 6]. The Blue Gene family of supercomputers is based on multi-core nodes organized within a classic distributed-memory system architecture. The Blue Gene/P node builds upon its predecessor by doubling the number of cores to four, providing a completely coherent shared memory system, and more than doubling the network performance.

Like many embedded cores, the PowerPC 450 used in Blue Gene/P has an integrated first-level cache, and is designed to be used in conjunction with a multi-level cache, generally. Therefore, it does not provide a sophisticated hardware coherence protocol (such as MESI), but provides support for inclusion, consisting of a write-through mode and an input port for specifying addresses to be invalidated. In the case of the PowerPC 450, the first-level cache is single-ported, so lookups and externally-generated invalidations compete for access.

The Blue Gene/P node architecture implements small, dedicated second-level caches (one per core) and a shared third-level cache, so the hardware must maintain memory consistency between all of the first- and second-level caches. This is done by writing all stores through to the third-level cache, and invalidating all remote copies of every store in the first- and second-level caches. Thanks to the high bandwidth available on chip, the store addresses from any core can be broadcast in a point-to-point manner to the other three cores at full speed. However, the large number of invalidations received by each core could degrade performance for two reasons. First, there is

a physical bottleneck at the first-level cache invalidation port because it must be shared by all of the invalidations coming from three other cores and a network DMA engine. Second, the invalidations disrupt the normal cache behavior because the first-level cache is single-ported. The scientific applications that Blue Gene/P favors are generally written to avoid sharing between the cores. As a result, most of the invalidations applied to the caches are useless and could be eliminated.

Our solution was to introduce snoop filters to eliminate the vast majority of useless invalidations. There are two common classes of snoop filters: source-based and destination-based. Source based filters eliminate snoops (invalidations, in our case) before they are even sent to remote caches, while destination-based filters eliminate snoops at the remote destinations. Source-based filters are more appropriate to implement together with directory-based coherence because the directory tracks remote copies of cache lines. Conversely, destination-based filters are more appropriate to implement together with snooping coherence, where cache state is only kept local to each core, as is the case for Blue Gene/P. Therefore, we chose to implement a destination-based snoop filter for every core.

The remainder of this paper describes the design and implementation of the Blue Gene/P snoop filter, which utilizes our novel stream register technique [21]. Section 2 gives a brief overview of the Blue Gene/P system architecture, while Section 3 goes into the details of the snoop filter. Section 4 describes the simulations we performed in order to arrive at our design point, and Section 5 presents some preliminary performance measurements from an actual Blue Gene/P system. We comment on related work in Section 6 and conclude with Section 7. The contributions of this paper are to present our design methodology and to demonstrate a working implementation of a destination-based snoop filter.

2. Blue Gene/P System Overview

The Blue Gene/P supercomputer is a scalable, distributed-memory system consisting of up to 262,144 nodes. Each node is built around a single compute ASIC with 2 GB or 4 GB of external DDR2 DRAM. The compute ASIC is a highly integrated System-on-a-Chip (SoC) chip multiprocessor (CMP). It contains four PowerPC 450 embedded processor cores [15], each with private, highly-associative, 32 KB first-level instruction and data caches. Each core is coupled to a dual-pipeline SIMD floating-point unit and to a small, private, second-level cache whose principal responsibility is to prefetch streams of data. In addition, the chip integrates an 8 MB, shared third-level cache, two memory controllers, five network controllers, and a performance monitor, as illustrated in Figure 1.

The PowerPC 450 microprocessor is a high-performance, out-of-order industry-standard PowerPC processor originally targeted at high-end embedded systems. The processor supports 2-way superscalar instruction execution with a seven stage pipelined microarchitecture. The processor cores include 32KB first-level instruction and data caches organized as 16 associative sets with 64 ways per set.

A dual-pipeline, SIMD floating point unit is attached to each processor core. The floating point unit can execute two fused multiply-add instructions per cycle for a peak floating point performance of 13.6 GFLOPS/node. The floating point unit pairs two floating-point register files and two execution pipes. The primary and secondary register files are independently addressable, but they can be jointly accessed by SIMD instructions. SIMD execution exploits the data-level parallelism often present in high-performance computing workloads to reduce the number of instructions that must be executed, while increasing the number of operations completed.

Like its predecessor, Blue Gene/P provides five dedicated communication networks: the torus network, the collective network, the barrier network, 10Gb/s Ethernet, and IEEE1149.1 (JTAG). The network interfaces are integrated on the same chip as the processing units. The main network is the torus, which provides high performance data communication to nearest neighbor nodes in a 3D configuration with low latency and high throughput. The collective network supports efficient collective operations, such as broadcast and reduction, and serves as the I/O interconnect. A more detailed description of Blue Gene/P can be found in [14].

3. Snoop Filter Architecture

In symmetric multiprocessor (SMP) architectures, snoop requests represent a significant fraction of all cache accesses, but only a small fraction of snoop requests are actually found in any of the remote caches [23][19]. This is particularly true of supercomputing applications where data partitioning and data blocking is performed to increase locality of reference and optimize overall compute performance. This motivated us to design a hardware device that filters out incoming snoop requests, reducing the number of actual snoop requests presented to the cache. In theory, a completely accurate filter can be created by duplicating the cache tag array and filtering out all snoops that miss. However, there are very significant technical realities that make this solution infeasible. For example, the SOC design flow makes it virtually impossible to modify a macro, such as the PowerPC core, or extract a piece of it, such as the cache tags. Furthermore, designing our own duplicate tag array with memory macros and gates

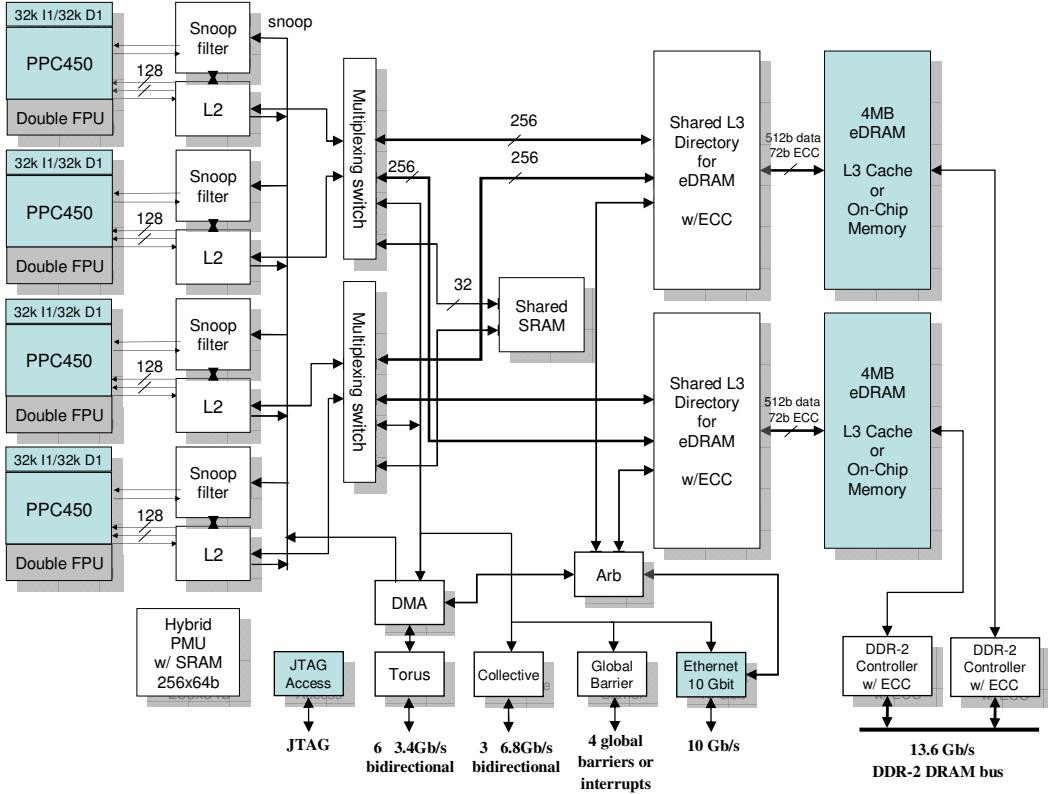


Figure 1. Blue Gene/P node architecture.

would never achieve the performance of the hand-placed L1 cache tags, so it would fall behind. Fortunately, it has been shown that very accurate filtering can be achieved with small designs that conservatively approximate the cache contents and operate at a reasonable frequency.

As mentioned in Section 1, data integrity between the Blue Gene/P cores is maintained with a cache coherence protocol based on write-invalidates, with all L1-caches operating in write-through mode. Every store not only updates the L1-cache of the issuing core, but also sends the write data via the L2 write buffer to the shared L3 cache. The L2s broadcast an invalidate request for the write address to ensure that no “stale” copy of the same datum will remain in the other L1s and L2s. We introduce our snoop filter at each of the four processors, located outside the L1 caches, as shown in Figure 2.

Each snoop filter receives invalidation requests from three remote cores and the network DMA by way of a point-to-point interconnect, so it must process requests from four memory writers concurrently (Figure 2). To handle these

simultaneous requests, we implement a separate snoop filter block, or “port filter”, for each interconnect port. Thus, coherency requests on all ports are processed concurrently, and a small fraction of all requests are forwarded to the processor. For example, each snoop filter in Figure 2 has four separate port filters (as shown in Figure 3), each of which handles requests from one remote processor or the network DMA unit.

Early on, we decided to include multiple filter units which implement various filtering algorithms in each port filter in order to capture various characteristics of the memory references. Some filtering units best capture time locality of memory references, whereas others capture reference streams. Through extensive simulation, we confirmed that the combination of various filtering algorithms achieves the highest filtering rate (reducing the number of snoop requests up to 99%, as shown in Section 4). We explored a number of snoop filter variants, and selected the combination of a snoop cache, a stream register filter, and a range filter.

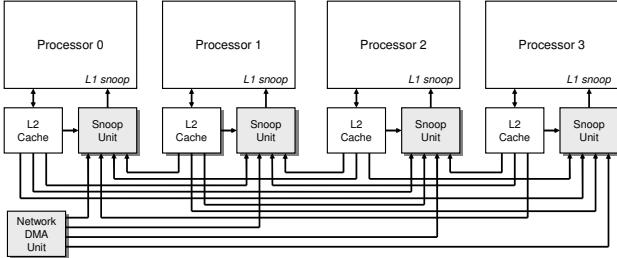


Figure 2. Blue Gene/P point-to-point snoop interconnect. All stores are sent by each L2 cache to all remote caches for invalidation. In addition, all network DMA stores are sent to all processor caches for invalidation.

The snoop cache is essentially a Vector-Exclusive-JETTY [19], which exploits the temporal locality property of some snoop requests. It records blocks that have been snooped recently (thus invalidated in the cache). It consists of a small, direct-mapped array, where an entry is created for each snoop request. A subsequent snoop request for the same block will match in the snoop cache and be filtered. If the block is loaded in the processor's L1 cache, the corresponding entry is removed from the snoop cache, and any new snoop request to the same block will miss in the snoop cache and be forwarded to the L1 cache. There is one dedicated snoop cache filter unit for each memory writer (three processors and the DMA) to allow for concurrent filtering of multiple coherency requests, thus increasing system performance.

Because the snoop cache is intended to capture spatial (as well as temporal) locality, storage efficiency is dramatically increased by using each entry to cache consecutive addresses in an aligned block. That is, each entry stores a base address together with a bit vector that indicates the presence of individual addresses offset from the base. The base address is essentially the address tag of the L1 data cache reduced by five bits that are used for encoding the presence vector. The presence vector encodes a group of 32 consecutive, aligned cache lines of the L1 data cache. Further details can be found in [21].

Unlike the snoop cache that keeps track of what *is not* in the cache, the stream register filter keeps track of what *is* in the cache. More precisely, the stream registers keep track of the lines that are in the cache, but may assume that some lines are cached which are not actually there. The stream registers capture address streams, so they are advantageous for applications where too many spatially-distributed references overflow the snoop caches.

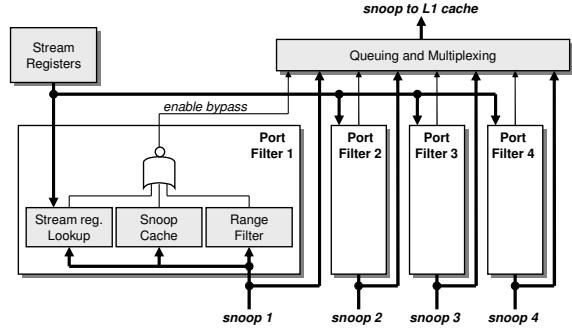


Figure 3. Architecture of a snoop unit including stream registers, four port filters, and queuing logic. Each port filter contains three separate filters, any of which can assert its output to filter the snoop. Otherwise, the snoop address bypasses the filter and is enqueued for invalidation.

The stream register filter has been described in detail in [21], and we provide a summary here. The heart of the filter is the stream registers themselves, of which there are a small number. One of these registers is updated with the line address every time the cache loads a new line. A particular register is chosen for update based upon the current stream register state and the address of the new line being loaded into the cache.

Every remote snoop is checked against the stream registers to see if it might be in the cache or not. This check can be performed in parallel because stream register lookups never change the state of the registers. Therefore, each port filter includes logic to compare the shared stream register state to its unique incoming snoop addresses.

A stream register actually consists of a pair of registers (the base and the mask) and a valid bit. The base register keeps track of address bits that are common to all of the cache lines represented by the stream register, while the corresponding mask register keeps track of which bits these are. More precisely, the mask register indicates which bits are “don’t-care” and which are not. An address matches a stream register if all of the bits which are *not* don’t-care match. For example, a stream register with base 0x12345678 and mask 0xFFFFFFF7F (where 0 means don’t-care) matches addresses 0x12345678 and 0x123456F8.

Every load address (resulting from an L1 miss) is merged with one of the stream registers. Bits that differ between the register base address and the load address cause the corresponding mask bits to be changed to don’t-care (if they are not already in that state). Therefore, two primary design issues emerge: how to choose which register to merge with,

Benchmark	Input parameters	Accesses to memory	Local cache hit rate	Remote cache hit rate	Total coherency accesses
Barnes	16K particles	1,602,120,476	99.73%	0.00047%	1,968,916,971
FFT	256K points	58,481,113	97.12%	0.0000057%	52,627,671
LU	512 matrix	202,643,933	99.24%	0.0000088%	204,434,958
Ocean	258 x 258 ocean	310,234,016	93.36%	0.03%	143,647,839
Cholesky	tk15.O	678,266,460	99.43%	0.00043%	614,572,560
FMM	16K particles	2,084,764,684	99.76%	0.00016%	2,976,937,884
Radix	10M keys	2,716,061,135	99.48%	0.00068%	3,491,931,132
Raytrace	car	404,977,091	98.43%	0.018%	358,731,051

Table 1. SPLASH-2 benchmark characteristics. The low remote cache hit rate shows that almost all invalidation snoops are useless and can be eliminated.

and how to deal with the loss in accuracy (which becomes worse over time).

Our first impulse for deciding which register to merge with was to calculate the Hamming distance between the load address and each of the base registers (taking the mask into account), and then choose the minimum, thereby causing the smallest number of mask bits to be changed to don’t-care. After careful consideration, we decided that the upper address bits should be favored in order to capture streams, where lower address bits would be expected to vary frequently. Therefore, we devised the “Most Matching Upper Bits” scheme which favors registers where the upper mask bits do not change. The basic idea is to choose the register with the longest matching string of consecutive bits, starting with the high-order bit. As shown in Section 4, this scheme was found to be superior and it is what we implemented. A related issue is when to choose a new register instead of one that already contains a stream. We do this by assigning a default distance (which we called the “empty affinity”) to unused registers and then including that in the update selection process.

As cache line load addresses are added to the stream registers, they become less and less accurate in terms of their knowledge of what is actually in the cache. In the limit, some mask register becomes all don’t-care and every possible address is considered to be in the cache and cannot be filtered. To overcome this, the stream register snoop filter includes a mechanism for resetting the registers back to their initial condition. As there is no efficient way to remove an address from the stream registers and guarantee correctness, the registers are cleared whenever the L1 cache has been completely replaced and they begin accumulating addresses anew. We call this complete replacement (relative to some initial state) a “cache wrap”.

The snoop filter contains logic that tracks cache wrapping based on notifications from the L1 cache every time a line is replaced. Because the PowerPC 450 cache uses round-robin replacement within sets, this logic basically consists of sixteen counters, one per set. The stream registers cannot simply be reset when the cache

wraps because they contain all the addresses that caused the wrap. Therefore, they are copied into a duplicate “history” set that is never updated, but participates in lookups. Once the cache wraps a second time, it is safe to discard the history set, so it is overwritten on every wrap in a pipelined manner.

We added a third filter, called the range filter, which unconditionally filters all snoops within a specified address range (or optionally, outside the range). This filter is useful when the four processors are utilizing completely distinct and contiguous sections of physical memory because it insures complete filtering.

Results of all three filter units are considered in a combined filtering decision. If any one of the filtering units decides that a snoop request should be filtered, then it is discarded. Otherwise, the snoop request is queued and forwarded to the L1 cache, as shown in Figure 3.

4. Design Space Exploration

The experiments in this paper represent our top-down approach for finding the best snoop filter design point. For our experiments, we used several codes from the publicly-available SPLASH-2 benchmark suite [24, 27]. We chose to use these codes because they are good representatives for a wide range of scientific applications, which is where we expect to see the most significant impact of Blue Gene/P. We have run the kernels (LU, FFT, Cholesky, and Radix), and some of the applications (Barnes, Ocean, Raytrace, and FMM). Table 1 shows the benchmarks used, the total number of memory accesses for all four processors, and the average percentage of misses in the L1 cache.

The analysis of the cache miss traces collected showed that the hit rate in the local cache of a processor is high, but the percentage of hits in the L1 data caches of all other processors (a.k.a. “remote” processors) is very low. Virtually all snoop requests will miss in the remote caches, representing the total snoop filter opportunity. Such small hit rates are due to the relatively small (32KB) first-level caches, and highlight the importance of snoop filtering for

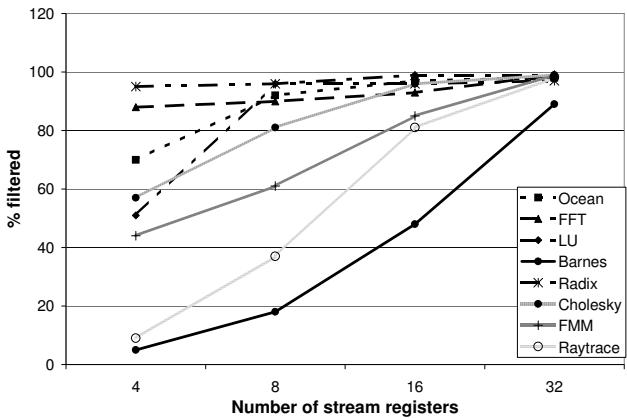


Figure 4. Percentage of snoops filtered as the number of stream registers is increased.

our architecture.

To collect the memory access traces, we used a custom simulator written with Augmint [20], a public-domain, execution-driven multiprocessor simulation environment. Augmint does not include a memory backend, thus requiring us to develop one from scratch. We modeled the L1 data caches of our four PowerPC 450 processors and an ideal memory system below that (since we were only concerned with the order of accesses and their effect on snoop filtering rates).

We developed a custom back-end simulator to process the traces and produce the results in this section. Because we wanted to measure the relative effectiveness of snoop filters over very long traces, we were not concerned with cycle accuracy, but only with the order of accesses and their effect upon the snoop filters and caches. Therefore, the trace entries are processed in order, and they have an instant, atomic effect upon the simulated caches and snoop filters. This simplification allowed us to compare many different alternative architectures, while exposing the significant trends. As a result, however, we could not measure actual execution times.

4.1 Stream Register Analysis

In order to determine the optimal number of stream registers, we have varied their number exponentially from 4 to 32, as shown in Figure 4. Not surprisingly, more stream registers filter a higher percentage of coherence snoop requests. But even when using only eight stream registers, we filter more than 90% of all snoop requests for three benchmark applications.

We observed that the effect of increasing the number of stream registers is not linear with respect to the snoop filtering rate. For the SPLASH-2 benchmarks, choosing

only four stream registers is clearly a bad design point. Selecting 8 or 16 stream registers seems to be the best compromise, whereas 32 stream registers (which doubles the area compared to 16 stream registers) only increases the snoop filtering rate significantly for one benchmark.

We have evaluated two different selection policies to choose the stream register for update, as described in Section 3: minimal Hamming distance, and most matching upper bits (MMUB). Figure 5(a) shows the effect of varying the empty affinity for various stream register sizes using the MMUB update policy for the Ocean application. We illustrate only one application here due to space constraints, but the results for other benchmarks are similar.

If the empty affinity is set too low, empty stream registers are used to establish new streams even for memory accesses belonging to the same stream, resulting in a low filtering rate because few streams are captured. Similarly, setting the affinity value too high causes streams to share registers and obliterate each other's mask bits, resulting in a low filtering rate. When the empty affinity is increased to more than 13, it starts to play a role in the filtering rate, depending on the number of stream registers. For filters having a higher number of stream registers, a higher affinity value is advantageous because it allows for more sensitive stream determination. For configurations with a smaller number of stream registers, a lower affinity allows for the most effective stream discrimination. For example, the optimal empty affinity value is 19 for eight stream registers, and 23 for 32 stream registers.

Figure 5(b) shows the effect of varying the empty affinity for various stream register sizes using the minimum Hamming distance update policy for the Ocean application. The results for the other benchmarks have similar trends, although the maximal filtering rate achieved by Raytrace and FMM was substantially lower than the others (33% for Raytrace; 43% for FMM). Similar to the MMUB update policy, setting the empty affinity too low or too high causes the filtering rate of the stream registers to be low. Although the optimal minimum Hamming distance empty affinity value is fixed at 25 for the Ocean application, the general trend is the same as for the MMUB policy over all the codes we studied.

Across all benchmarks, the sensitivity of the filtering rate to the empty affinity value was less for the MMUB update policy. In addition, for Raytrace and FMM, the MMUB update policy achieves almost 100% filtering, while the Hamming distance update policy reaches less than 50%, even for the largest configurations. The MMUB policy has the advantage of ignoring low-order address bits when establishing streams in the stream registers. The minimum Hamming distance policy results in well-correlated addresses that differ in their low-order address bits being mapped to different stream registers, thereby

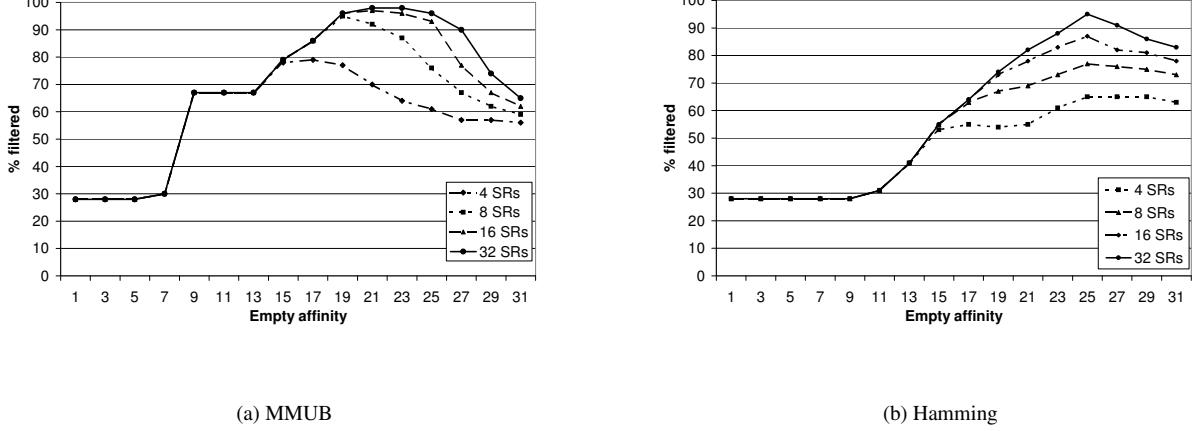


Figure 5. Stream register filter behavior. Percentage of snoops filtered as the empty affinity and number of stream registers is varied for the Ocean application.

causing a kind of pollution. In this case, the effectiveness of stream registers is limited.

4.2 Snoop Cache Analysis

In order to determine the optimal sizing for a snoop cache-based filter, we have varied two parameters: the number of entries, ranging from 4 to 32, and the number of consecutive lines tracked by each entry, ranging from 1 to 64 (as determined by the length of the presence vector). The results for FFT, Ocean and Raytrace are shown in Figure 6. Results for the other benchmarks were similar.

Our experiments show that filters with a greater number of snoop cache entries and/or a longer presence vector are more effective at filtering snoop requests, primarily because of their larger capacity. The filter limit varies for various applications from 83% for Ocean to 99% for Raytrace. For each application, the shape of the cache size vs. presence vector size surface differs, depending on its memory access pattern.

FFT reaches its maximum filtering rate only for bigger configurations, while Ocean never exceeds a filtering rate of 83%. Raytrace is characteristic of several of the benchmarks that do very well with most snoop cache configurations.

4.3 Combining Both Filters

We have discussed and analyzed two snoop filters separately. As both filters cover different memory access patterns, the most effective filtering is achieved when

putting the two filters together. We will show that using the combination of two filters, we can achieve high filtering rates even though each filter unit is quite small.

In order to determine the optimal sizing for our snoop filter, we have varied three parameters: the number of stream registers (4, 8, or 16), the number of snoop cache lines (4 or 8), and the empty affinity (in the most effective range from 19 to 25). We keep the snoop cache presence vector at 32 bits in length. Figure 7 shows results for the same benchmarks shown in the previous figures. Results for the other benchmarks are similar, showing a very high filtering rate for all configurations. Obviously, the two filtering techniques complement each other to obtain near-perfect filtering, even for filter configurations with a modest latch count.

5. Hardware Measurements

Based on our design space exploration, the Blue Gene/P system architecture implements the snoop filters in a point-to-point connection with four port filters, each having stream register lookup logic, a snoop cache, and a range filter. Based on our analysis, each port filter implements eight stream registers and eight snoop cache lines, each with a 32-bit valid line vector.

We recently brought up the first systems built with the Blue Gene/P compute ASICs, and we were able to make some preliminary hardware measurements. We measured runtimes for the NAS benchmarks both with and without the

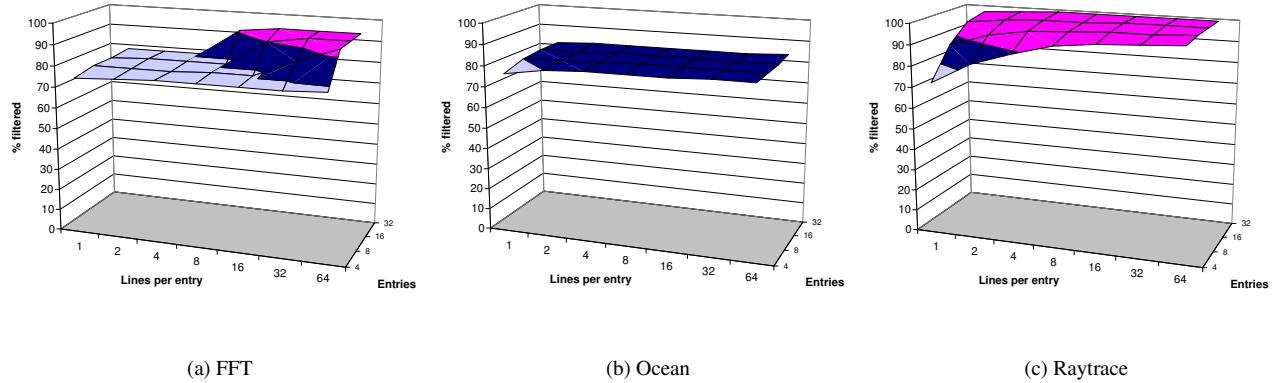


Figure 6. Snoop cache filter behavior. Percentage of snoops filtered as the number of snoop cache sets and the number of lines per set is varied.

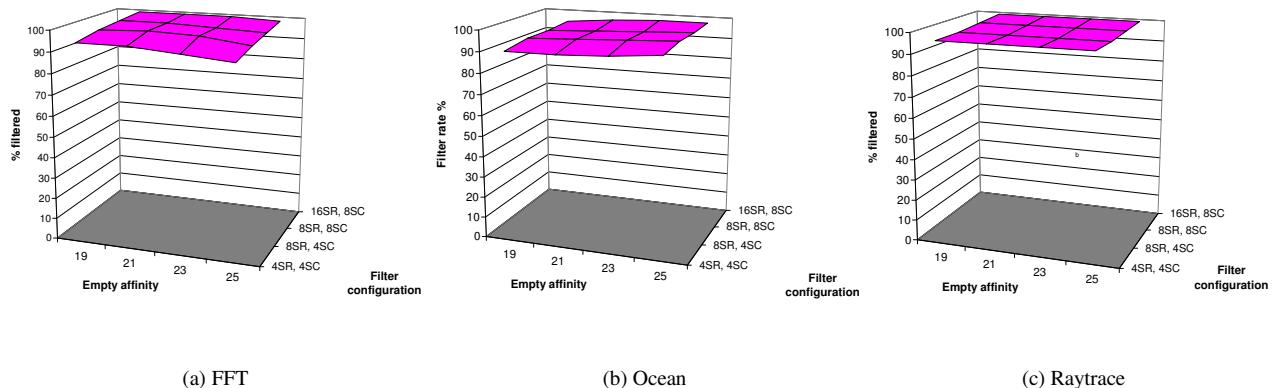


Figure 7. Combined filter behavior. Percentage of snoops filtered for several stream register and snoop cache configurations as the empty affinity is varied.

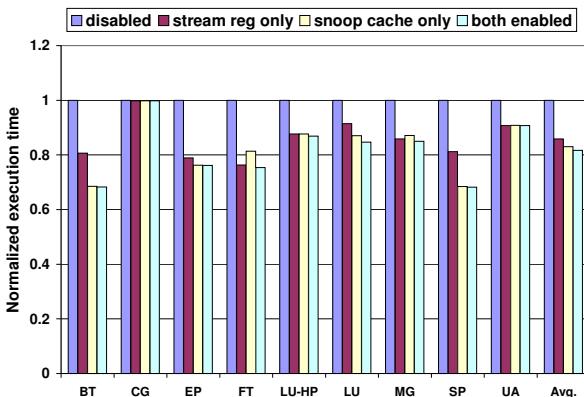


Figure 8. Normalized execution times of the NAS benchmarks on a Blue Gene/P node for various snoop filter configurations.

snoop filters enabled to see the performance effect. Figure 8 shows the normalized execution times for four snoop filter configurations: all snoop filters disabled, stream registers only, snoop caches only, and both snoop filters enabled.

Across all benchmarks, snoop filters reduce execution time. Most benchmarks benefit more from snoop caches (like BT and SP), while some get better performance with stream registers (like FT). The only exception is CG, whose execution time stays the same independent of the snoop filtering. For all benchmarks, the combination of the two snoop filters yields the best result, and reduces the execution time over 30% for BT and SP. On average, execution time is reduced by about 20%.

The hardware measurements confirm the effect of the significant reduction in coherence traffic shown by the simulations. The coherence traffic reduction translates into significant improvement in performance due to the use of the snoop filter. This confirms our simulation strategy to select an effective design point.

6. Related Work

Moshovos et al. [19] describes a snoop filter called JETTY that combines two complementary filtering methods. JETTY defines a characterization of filters as “include” or “exclude”. An include filter tracks what is contained in a cache (or caches) while an exclude filter tracks what is not. The exclude filter consists of a cache of recently invalidated lines. A snoop that hits in the exclude filter is guaranteed not to be in cache, so it can be filtered. The include filter consists of several scoreboard arrays that track disjoint subsets of cache lines present in cache.

Moshovos et al. argue for snoop filtering as a means for power savings. However, our work is primarily motivated by the need to filter useless snoops that reduce performance.

We also consider chip area and power consumption to be significant constraints, causing us to look beyond the simple and accurate method of duplicating the cache tags as a filter.

Several coherent network switches contain source-based snoop filters that block unnecessary coherence requests from ever leaving a node. One such example is the Scalability Port Switch of the Intel E8870 chipset [5]. In this case, the snoop filter tracks the state of all cache lines within a 4-processor node for a system with up to 4 such nodes. Kant [16] modeled a similar system architecture with such a snoop filter. This architecture is also described in the Azusa system [4], which is based on Intel Itanium processors and may use an Intel chipset.

In [17], a HyperTransport network switch for use with AMD Opteron processors is described. The snoop filtering technique is basically the same as that of the E8870, including the fact that 4-processor nodes are supported.

A similar but more tightly-coupled architecture is evaluated in [7], where a single memory controller switch connects multiple multi-processor nodes and contains a snoop filter. The filter prevents unnecessary snoop requests between the nodes, and several variants are studied.

Snoop filters in tightly-coupled multiprocessors, such as CMPs, can be located at each processor in order to shield each from unnecessary snoops without changing the overall coherence scheme. Ekman et al. [9] describe a CMP architecture with Page Sharing Tables that are exclude filters at the granularity of memory pages, rather than cache lines. This architecture is more complicated in that the Page Sharing Tables coordinate to track sharing rather than just presence.

The idea of preventing remote snoop requests from being broadcast can also be applied at the chip level [18]. In this work, snoop filters keep track of memory regions, which can be quite large, and block remote snoops for memory that is known not to be shared.

7. Conclusion

With the emergence of commodity multi-core processors and CMPs, we have entered the era of the SMP-on-a-chip. These high-performance systems will generate an enormous amount of shared memory traffic, so it will be important to eliminate as much of the useless inter-processor snooping as possible. In addition, power dissipation has become a major factor with increased chip density, so mechanisms to eliminate useless coherence actions will be important.

In this paper, we have described and evaluated a snoop filtering architecture for the Blue Gene/P supercomputer, and presented some preliminary performance measurements. Our snoop filter uses multiple, complementary filtering techniques, and parallelizes the filters so that they can handle snoop requests from all remote processors

simultaneously. We explored the design space using the SPLASH-2 benchmarks together with a custom trace generator and simulator. Our Blue Gene/P measurements confirm the direct positive effect that the snoop filters have on performance.

Acknowledgements

The Blue Gene/P project has been supported and partially funded by Argonne National Laboratory and Lawrence Livermore National Laboratory on behalf of the United States Department of Energy under Subcontract No. B554331. The authors would like to thank Ruud Haring, and Michael Gschwind for their valuable contributions to this paper.

References

- [1] Special issue on the IBM POWER4 system. *IBM Journal of Research and Development*, 46(1), January 2002.
- [2] Special issue on the IBM Blue Gene/L system. *IBM Journal of Research and Development*, 49(2/3), March/May 2005.
- [3] Advanced Micro Devices. AMD multi-core technology. <http://multicore.amd.com>, 2007.
- [4] F. Aono and M. Kimura. The Azusa 16-way Itanium server. *IEEE Micro*, 20(5):54–60, September/October 2000.
- [5] F. Briggs, S. Chittor, and K. Cheng. Micro-architecture techniques in the Intel E8870 scalable memory controller. In *Proceedings of the 3rd Workshop on Memory Performance Issues*, pages 30–36, June 2004.
- [6] A. A. Bright, M. R. Ellavsky, A. Gara, R. A. Haring, G. V. Kopcsay, R. F. Lembach, J. A. Marcella, M. Ohmacht, and V. Salapura. Creating the Blue Gene/L supercomputer from low power SoC ASICs. In *Digest of Technical Papers, 2005 IEEE International Solid-State Circuits Conference*, pages 188–189, 2005.
- [7] S. Chinthamani and R. Iyer. Design and evaluation of snoop filters for web servers. In *Proceedings of the 2004 Symposium on Performance Evaluation of Computer Telecommunication Systems*, July 2004.
- [8] R. Dennard, F. Gaenslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted MOSFETs with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, pages 256–268, 1974.
- [9] S. Ekman, F. Dahlgren, and P. Stenstrom. TLB and snoop energy-reduction using virtual caches in low-power chip-multiprocessors. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design*, pages 243–246, August 2002.
- [10] S. Gochman, A. Mendelson, A. Naveh, and E. Rotem. Introduction to Intel Core Duo processor architecture. *Intel Technology Journal*, May 2006.
- [11] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid State Circuits*, 31(9):1277–1284, September 1996.
- [12] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell’s multicore architecture. *IEEE Micro*, 26(2):10–24, March 2006.
- [13] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. A novel SIMD architecture for the CELL heterogeneous chip-multiprocessor. In *Hot Chips 17*, Palo Alto, CA, August 2005.
- [14] IBM Blue Gene team. Overview of the IBM Blue Gene/P project. *IBM Journal of Research and Development*, 52(1/2), January 2008.
- [15] International Business Machines. PPC450Ax6 embedded processor core. *Users Manual, SA14-2754-04*, September 2006.
- [16] K. Kant. Estimation of invalidation and writeback rates in multiple processor systems. <http://kkant.gamerspace.net/papers/invalid.pdf>.
- [17] C. Kelcher, K. McGrath, A. Ahmed, and P. Conway. The AMD opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66–76, March/April 2003.
- [18] A. Moshovos. Regionscout: Exploiting coarse grain sharing in snoop-based coherence. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 234–245, June 2005.
- [19] A. Moshovos, G. Memik, B. Falsafi, and A. N. Choudhary. JETTY: Filtering snoops for reduced energy consumption in SMP servers. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 85–96, 2001.
- [20] A.-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The Augmint multiprocessor simulation toolkit for Intel x86 architectures. In *Proceedings of 1996 International Conference on Computer Design*, pages 486–490, October 1996.
- [21] V. Salapura, M. Blumrich, and A. Gara. Improving the accuracy of snoop filtering using stream registers. In *Proceedings of the 8th MEDEA Workshop*, pages 25–32, September 2007.
- [22] V. Salapura et al. Power and performance optimization at the system level. In *Proceedings of the 2nd International Conference on Computing Frontiers*, pages 125–132, Ischia, Italy, May 2005. ACM.
- [23] C. Saldanha and M. Lipasti. Power efficient cache coherence. In *Proceedings of the Workshop on Memory Performance Issues*, June 2001.
- [24] J. Singh, W.-D. Weber, and A. Gupta. Splash: Stanford parallel applications for shared memory. *Computer Architecture News*, pages 5–44, March 1992.
- [25] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. Power5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5), July 2005.
- [26] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P. Strenski, and P. Emma. Optimizing pipelines for power and performance. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 333–344, Istanbul, Turkey, November 2002. ACM/IEEE.
- [27] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36. ACM, June 1995.

A VLIW Architecture for a Trace Scheduling Compiler

Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth, Paul K. Rodman

Multiflow Computer
175 North Main Street
Branford, CT. 06405
(203) 488-6090

1. Abstract

Very Long Instruction Word (VLIW) architectures were promised to deliver far more than the factor of two or three that current architectures achieve from overlapped execution. Using a new type of compiler which compacts ordinary sequential code into long instruction words, a VLIW machine was expected to provide from ten to thirty times the performance of a more conventional machine built of the same implementation technology.

Multiflow Computer, Inc., has now built a VLIW called the TRACE™ along with its companion Trace Scheduling™ compacting compiler. This new machine has fulfilled the performance promises that were made. Using many fast functional units in parallel, this machine extends some of the basic Reduced-Instruction-Set precepts: the architecture is load/store, the microarchitecture is exposed to the compiler, there is no microcode, and there is almost no hardware devoted to synchronization, arbitration, or interlocking of any kind (the compiler has sole responsibility for runtime resource usage).

This paper discusses the design of this machine and presents some initial performance results.

2. Background for VLIWs

The search for usable parallelism in code has been in progress for as long as there has been hardware to make use of it. But the common wisdom has always been that there is too little low-level fine-grained parallelism to worry about. In his study of the RISC-II processor, Katevenis reported^{Kate85} "...We found low-level parallelism, although usually in small amounts, mainly between address and data computations. The frequent occurrence of conditional-branch instructions greatly limits its exploitation."

This result has been reported before Tjad70, Fost72 and judging from the lack of counterexamples, seems to have been interpreted by all architects and system designers to date as a hint from Mother Nature to look elsewhere for substantial speedups from parallelism.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Researchers at Yale, however, Fish83, Elli86 found that fine-grained parallelism could be exploited by a sufficiently clever compiler to greatly increase the execution throughput of a suitably constructed computer. The compiler exploited statistical information about program branching to allow searching beyond the obvious basic blocks in a program (e.g., past conditional branches) for operations that could be performed in parallel with other possibly-unrelated operations. Fish79 Logical inconsistencies that were created by these motions were corrected by special compensation code inserted by the compiler.

These researchers labelled their proposed architecture "Very-Long-Instruction-Word", and suggested that a single-instruction-stream machine, using many functional units in parallel (controlled by an appropriately large number of instruction bits) would be optimal as an execution vehicle for the compiler. It was proposed that the most suitable VLIW should exhibit four basic features.

- One central controller issues a single long instruction word per cycle.
- Each long instruction simultaneously initiates many small independent operations.
- Each operation requires a small, statically predictable number of cycles to execute.
- Each operation can be pipelined.

In the same spirit as RISC efforts such as MIPS Henn81 and the IBM 801 Radi82 the microarchitecture is exposed to the compiler so that the compiler can make better decisions about resource usage. However, unlike those efforts, a VLIW provides many more functional units that can be used in parallel; Multiflow's Trace Scheduling compiler finds parallelism across basic blocks to keep them busy.

Multiflow Computer, Inc., has now demonstrated the fundamental soundness of both the compiler and the architecture, announcing a product based on these concepts. This paper will discuss Multiflow's TRACE architecture. Some initial experience with programming a VLIW (bringing up UNIX on the TRACE machine) will be recounted.

TRACE, Trace Scheduling, and Multiflow are trademarks of Multiflow Computer, Inc. UNIX is a registered trademark of AT&T Technologies. VAX and VMS are registered trademarks of Digital Equipment Corporation. IBM is a registered trademark of International Business Machines Inc.

3. Introduction to VLIW Computer Architecture

VLIW computers are a fundamentally new class of machine characterized by

- A single stream of execution (one program counter, and one control unit).

- A very long instruction format, providing enough control bits to directly and independently control the action of every functional unit in every cycle.

- Large numbers of datapaths and functional units, the control of which is planned at compile time. There are no bus arbiters, queues, or other hardware synchronization mechanisms in the CPU.

Unlike a vector processor, no high level regularity in the user's code is required to make effective use of the hardware. And unlike a multiprocessor, there is no penalty for synchronization or communication. All functional units run completely synchronized, directly controlled in each clock cycle by the compacting compiler.

The true cost of every operation is exposed at the instruction set level, so that the compiler can optimize instruction scheduling. Pipelining allows new operations to begin on every functional unit in every instruction. This exposed concurrency in the hardware allows the hardware to always proceed at full speed, since the functional units never have to wait for each other to complete. Pipelining also speeds up the system clock rate. Given that we can find and exploit scalar parallelism, there is less temptation to try to do "too much" in a single clock period in one part of the design, and hence slow the clock rate for the entire machine. Judiciously used small scale pipelining of operations like register-to-register moves and integer multiplies, as well as the more obvious floating point calculation and memory reference pipelines, helped substantially in achieving a fast clock rate.

The absence of pipeline interlock or conflict management hardware makes the machine simple and fast. Hennessy ^{Henn81} has estimated that building the MIPS chip without interlocked pipeline stages allowed that machine to go 15% faster. In our VLIW, given our large number of pipelined functional units, pipeline interlocking and conflict resolution would have been almost unimplementable, and the performance degradation would have been far greater than 15%.

VLIWs exploit the same low-level, scalar parallelism that high-end scalar machines have used for decades. Execute-unit schedulers which look ahead in a conventional instruction stream and attempt to dynamically overlap execution of multiple functional units were incorporated in systems beginning with the IBM 360/91 ^{Toma82} and the Control Data 6600. ^{Thor70} These "scoreboards" perform the same scheduling task at runtime that Multiflow's Trace Scheduling compacting compiler performs at compile time. Even with such "complex and costly hardware", Acosta et al. ^{Acos86} report that only a factor of 2 or 3 speedup in performance is possible. This limitation, of course, is the same as previously discussed: the hardware cannot see past basic blocks in order to find usable concurrency.

Over the last two decades, the cost of computer memory has dropped much faster than the cost of logic, making the construction of a VLIW, which replaces scheduling logic with instruction-word memory, practical and attractive. In conjunction with the global optimization ability of our compiler, we find

no remaining reasons to build run-time scheduling hardware. The scheduling problem is much better solved in software at compile-time. This "no control hardware" attitude permeates the design of the TRACE architecture.

An obvious potential disadvantage to the VLIW approach is that instruction code object size could grow unmanageably large, enough so that much of the performance advantage would be lost in extra memory costs and disk paging. We have addressed this problem in the design of the TRACE, and have a very satisfactory result to report in Section 9.

4. Trace Scheduling Compacting Compilation

Multiflow's Trace Scheduling compacting compiler automatically finds fine-grained parallelism throughout any application. It requires no programmer intervention, either in terms of restructuring the program so it fits the architecture or in adding directives which explicitly identify opportunities for overlapped execution. This is in sharp contrast to "coarse-grained" parallel architectures, including vector machines, shared-memory multiprocessors, and more radical structures such as hypercubes ^{Seit85} or massively-parallel machines. ^{Walt87} The process by which programs are converted into highly parallel wide-instruction-word code is transparent to the user.

To detect fine-grained parallelism, the compiler performs a thorough analysis of the source program. One subroutine or module is considered at a time. After performing a complete set of "classical" optimizations, including loop-invariant motion, common subexpression elimination, and induction variable simplification, the compiler builds a flow graph of the program, with each operation independently represented.

Using estimates of branch directions obtained automatically through heuristics or profiling, the compiler selects the most likely path, or "trace", that the code will follow during execution. This trace is then treated as if it were free of conditional branches, and handed to the code generator. The code generator schedules operations into wide instruction words, taking into account data precedence, optimal scheduling of functional units, register usage, memory accesses, and system buses; it emits these instruction words as object code. This greedy scheduling causes code motions which could cause logical inconsistencies when branches off-trace are taken. The compiler inserts special "compensation code" into the program graph on the off-trace branch edges to undo these inconsistencies, thus restoring program correctness.

This process allows the compiler to break the "conditional jump bottleneck" and find parallelism throughout long streams of code, achieving order-of-magnitude speedups due to compaction.

The process then repeats; the next-most-likely execution path is chosen as a trace and handed to the code generator. This trace may include original operations and compensation code. It is compacted; new compensation code may be generated; and the process repeats until the entire program has been compiled.

A number of conventional optimizations aid the trace selection process in finding parallelism. Automatic loop unrolling and automatic inline substitution of subroutines are both incorporated in Multiflow's compilers; the compiler heuristically determines the amount of unrolling and substitution, substantially increasing the parallelism that can be exploited.

More information about the design of the compiler will be forthcoming; interested readers may also find previous reported research enlightening. ^{Fish79, Elli86, Fish81, Fish84, Elli84}

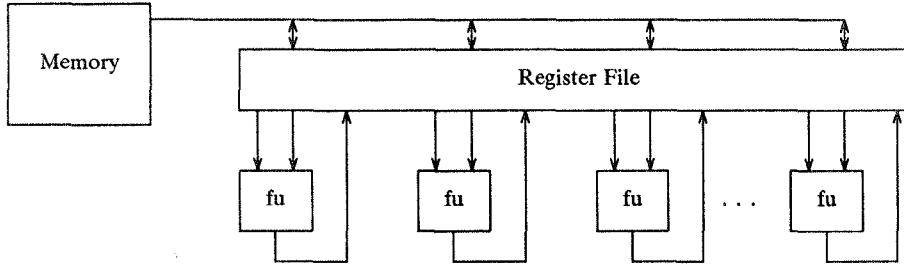


Figure 1. Block Diagram of an Ideal VLIW Execution Engine

5. The Ideal VLIW

The ideal execution vehicle for this compacting compiler would be a machine with many functional units connected to a large central register file. Each functional unit would ideally have two read ports and one write port to the register file, and the register file would have enough memory bandwidth (Very Large) to balance the operand usage rate of the functional units. A block diagram of this ideal engine is shown in Fig. 1. A centralized register file would simplify code generation; when scheduling operations, the selection of functional unit would be unimportant, and the code generator would only have to worry about when the operation was scheduled. The provision of a full set of separate read/write ports for each functional unit would guarantee the independence of each operation. As long as there were enough registers, no extraneous data movement would ever be needed.

However, any reasonably large number of functional units requires an impossibly large number of ports to the register file. Chip real estate and chip pinout limit the number of independently controlled ports which can be provided on a single register set. The only reasonable implementation compromise is to partition the register files, in some way that minimizes the additional workload on the compiler and also minimizes data traffic between the different register files.

Another problem in this “ideal VLIW” is the memory system. All modern computers have to deal with a significant speed mismatch between the logic used to build the processor and the access time of the dynamic RAMs used to build the memory. The memory architecture of our VLIW is perhaps the area where the greatest advantage is gained over other approaches; it is discussed in Section 6.4.

6. A Real VLIW

These were the goals for the TRACE processor design:

- A modular design, with an expandable number of functional units;
- Use standard, high-volume, low-cost electronics;
- Use standard DRAMs for main memory for high capacity at low cost;
- Deliver the highest possible performance for 64-bit floating point intensive computations;
- Perform well in a multi-user environment.

The processor is built of five board types: integer boards (I), floating point boards (F), a “global controller” (GC), memory

controllers (MC), and I/O Processors (IOP). Each board is an 18 by 18 inch, 8 to 10 layer printed circuit, interconnected via a single 19-slot backplane. The backplane connectors provide 630 pins usable for signals, plus power and ground connections. The core of the computational engine was built in 8000 gate CMOS gate arrays with 154 signal pins. Advanced Schottky TTL was used for “glue” logic and bus transceivers.

Research at Yale in machine architecture accompanied research into compilation across basic blocks. The focus of efforts at Yale was to develop buildable, scalable, technology-independent machine models which could be used to evaluate the success of such a compiler. At Multiflow, the design team spent the first year studying alternative partitionings, functional unit mixes, and opcode repertoires, with a specific product and implementation technology in mind. This allowed us to be much more aggressive in hardware support for effective compilation, and come much closer to an “ideal VLIW” structure than any machines we considered at Yale. Architectural alternatives were evaluated using a prototype easily-retargetable compiler and simulator.^{Elli86}

Given the implementation constraints, we decided that a maximum of eight 32-bit buses could traverse the edge connector. The number of buses we could support was one of the major constraints on CPU expandability, given the required balance between memory bandwidth and the rate of floating point operations to support general computations.

We partitioned the core processor into an Integer and a Floating unit (the “I” and “F” boards), and provided separate physical register files for the floating point functional units and the integer ALUs. This makes intuitive sense, since there is little need for performing integer operations on floating point operands (and vice versa), while it is often the case that a chain of floating point operations can proceed while the integer units are performing the address computations in parallel.

The unit of processor expansion is this Integer-Floating board pair. One, two, or four I-F pairs can be configured, corresponding to a 256-bit, 512-bit, or 1024-bit instruction word.

Two 32-bit buses carry data traffic between the boards of a pair (through a dedicated front-edge path, rather than the backplane). Each board carries its own register file/crossbar touching twelve 32-bit datapaths, handling four writes, four reads, and four bus-to-bus forwards in each minor cycle, plus bypassing from every write port to every read port. Sixty-four 32-bit registers are provided. This register file/crossbar is implemented in nine gate arrays; each is a 4-bit slice (byte parity is carried throughout the machine).

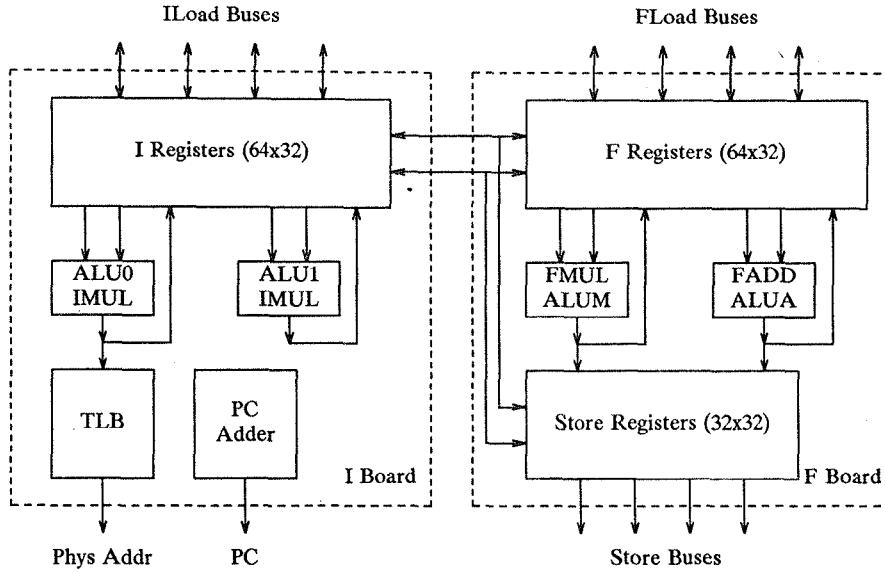


Figure 2. I and F Board Block Diagrams

6.1. Integer Operations

The integer instruction set comprises some 80-odd opcodes, including arithmetic, logical, and compare operations; high performance 16-bit primitives for 32-bit integer multiplication; and shift, bit-reverse, extract, and merge operations for bit and byte field manipulations. The integer instruction set (excluding multiplication) is implemented in a single gate array, four copies of which are included in a single I-F pair.

Every operation specifies its destination register at the time of initiation. A modifier (“dest_bank”) specifies which register file contains the target register: the local general register bank, the general register bank in the paired F unit, the store file in the paired F unit, a general register bank in another I unit, or a branch bank on an I or F board. Branch banks are small 1-bit register files used to control branching; see Section 6.5.2.

Substantial support was provided for injecting immediate constants into the computation. Each ALU can get a 6-bit, 17-bit, or 32-bit immediate provided on one operand leg, under the control of the instruction word. A 32-bit immediate field is flexibly shared between ALU0, ALU1, and a 32-bit PC adder which generates branch target addresses.

Included in the I board instruction set are pipelined load and store instructions for referencing memory. Memory addresses are 32-bit byte pointers. The memory system hardware operates only on 32-bit or 64-bit quantities; access to fields of other sizes is provided via extract/merge/shift operations which are arranged to accept the same 32-bit pointer, using the low bits to specify the field position.

The I board also includes dynamic address translation hardware and supports demand-paged virtual addressing. A Translation Lookaside Buffer provides a cache of 4K virtual-to-physical address translations on 8KB page boundaries. Simple paging is used; no segmentation or other address translation is provided. Traps are taken on TLB misses; trap-handling software manages TLB refills. The TLB is process-tagged so that flushes are unnecessary at context switches. Its indexing scheme includes a process-ID hash to minimize conflicts between entries for multiple processes.

Each instruction executes in two minor cycles, or “beats”. Each beat is 65ns. The I board ALUs perform unique operations, specified by new control words presented in both the early and late beats.

Figure 3 shows the format of the instruction word for a single I-F pair. This instruction word is replicated four times in a fully configured processor.

6.2. Floating Operations

The F board (floating point) was optimized for 64-bit IEEE standard floating point computation. It uses the same register file chips as the I board, providing sixty-four 32-bit registers (which are used in pairs for 64-bit quantities). The floating functional units each perform one new operation per instruction, or every other beat. The 32-bit datapaths carry 64-bit data to and from the floating point units in two beats.

A pipelined floating adder/ALU shares resources and opcodes with an integer ALU; the integer ALU has one beat latency, while the floating adder has six beat latency in 64-bit mode. A floating multiplier/divider similarly shares resources with another integer ALU. The multiplier has seven beat latency doing 64-bit multiplication, and 25 beat latency doing 64-bit division. New operations may be started on each functional unit in each instruction (except on the multiplier while division is in progress).

The integer instruction set, excluding memory references and integer multiply, is available on both ALUs on the F board. This was an implementation convenience. We found it desirable to provide “fast move” paths to allow data moves without the pipeline depth of the floating point units. We also included the integer SELECT operation, which provides the semantics of the C “?” operator without branching. It was simpler to include copies of the already-designed integer ALU than to dedicate another gate array to these more limited functions.

Word 0: I 0 ALU 0, Early beat.

31	25	24	19	18	16	15	13	12	11	7	6	1	0
opcode	dest	dest_bank	branch_test						src1	src2	Imm		

Word 1: Immediate constant 0 (early).

31	0
immediate constant (early)	

Word 2: I 0 ALU 1, Early beat.

31	25	24	19	18	16	15	13	12	11	7	6	1	0
opcode	dest	dest_bank	branch_test						src1	src2	Imm		

Word 3: F 0 FA/ALUA control fields.

31	25	24	23	22	17	16	15	11	10	6	5	4	3	1	0
opcode	64			dest			src1	src2				dest_bank			

Word 4: I 0 ALU 0, Late beat.

31	25	24	19	18	16	15	13	12	11	7	6	1	0	
opcode	dest	dest_bank							src1	src2	Imm			

Word 5: Immediate constant 0 (late).

31	0
immediate constant (late)	

Word 6: I 0 ALU 1, Late beat.

31	25	24	19	18	16	15	13	12	11	7	6	1	0	
opcode	dest	dest_bank							src1	src2	Imm			

Word 7: F 0 FM/ALUM control fields.

31	25	24	23	22	17	16	15	11	10	6	5	4	3	1	0
opcode	64			dest			src1	src2				dest_bank			

Figure 3. Instruction Word Format for one I-F pair

The floating point functional unit pipelines are “self-draining”; the destination register is specified when the operation is initiated, and a hardware control pipeline carries the destination forward, writing the target register when the operation completes. To allow interrupts to occur at any point in the program, by convention the target register of any pipelined operation is “in use” from the beat in which the operation is initiated until the beat in which it is defined to be written. If a trap or interrupt occurs while the pipelined operation is in process, the register gets written early, relative to the execution of the instructions immediately following in the program text.

The F board also carries the Store Register File. When an I board issues a Store opcode, it also issues a “Store Read Address” on a bus that all F boards monitor. Physical addresses are generated on the I boards, and data to be stored comes from the “Store Register File” on the F boards. The Store Register File, implemented using the same register chip used elsewhere, expands the number of register read ports and eliminates pipeline conflicts between memory stores and other operations.

6.3. System configuration

A fully configured TRACE processor incorporates four I-F pairs. With a 1024-bit instruction word that initiates 28 operations per instruction, it has peak performance of 215 “VLIW MIPS” and 60 MFLOPS. Figure 4 shows the top level architecture and backplane interconnect for the system. The entire CPU and its interconnect is synchronized to a single master clock.

The ILoad Buses, FLoad Buses, and Store Buses are each a set of 4 independently-managed synchronous 32-bit buses. The Load buses are multidirectional, and the Store buses are unidirectional. Each bus is independently scheduled by the compiler for each execution beat. Each Load bus carries a 10-bit control field, or “tag”, specifying the destination of the data carried on the bus in that beat; the tags are derived from instruction words which specify data moves or memory references. Because the “arbitration” is handled by the compiler, the buses are fast, simple, and cheap. This is a major advantage versus the complicated interconnects of a multiprocessor, where arbitration, buffering, interlocking, and interrupts are required. Pis85

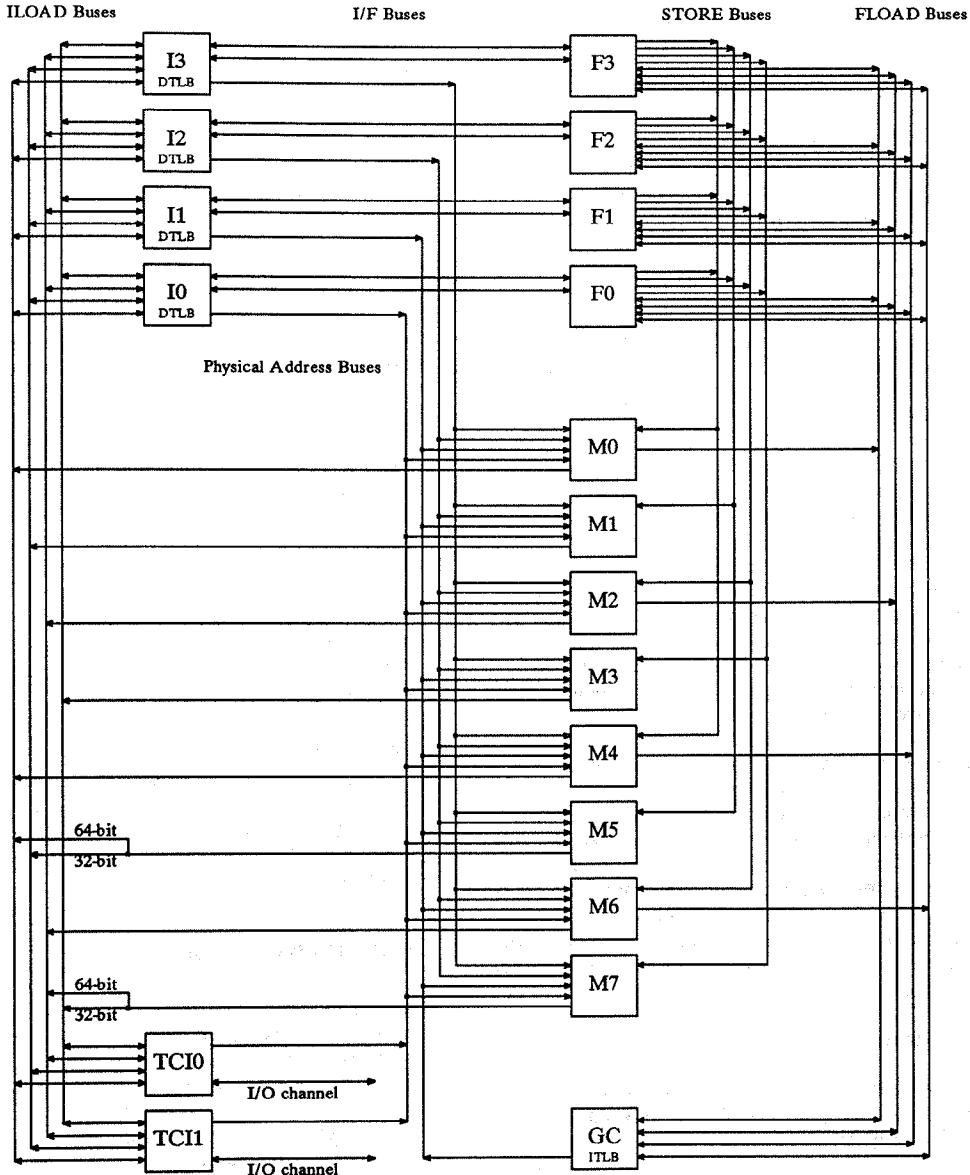


Figure 4. The TRACE Major Datapaths

Up to eight memory controllers comprise the memory system. Each controller carries up to 8 independent banks. Memory addresses are interleaved among controllers and banks. Each memory controller watches all four physical address buses for valid requests, and touches one Store bus, one I bus, and one F bus. A fully populated memory system comprises 512 MBytes of physical storage.

6.4. The Memory Subsystem

The speed of the CPU/memory interconnection in a computer system is a first order determinant of overall performance (the legendary “von Neumann bottleneck”). The designers of every modern computer system, from the IBM PC to the Cray-2, have had to deal with a substantial mismatch in speed between the cycle time of the processor and the access time of the memories. This mismatch ranges from a factor of 2 to a factor of 59. Two major approaches have been taken by the designers of these systems to handle the mismatch: caching and interleaving.

Cache memories provide lower latency than main memory when there is reasonable locality of reference in the access pattern. They can be expensive to implement, and their cost grows rapidly in systems attempting to issue more than one memory reference per cycle. While instruction caches are universally effective, data caches work poorly in many scientific applications, where very large arrays of data are repeatedly accessed; hit rates fall off rapidly, and system performance degrades to the performance of the backing store (main memory).^{Smit82}

Interleaved memories exploit parallelism among memory references to address the speed mismatch in a different manner. Memory addresses are spread across multiple independent banks of RAMs. The memory system is pipelined; while one or more new references may be initiated in each cycle, it takes multiple cycles for a single reference to complete. During several of those cycles, a single RAM bank will be tied up and unable to accept new requests; achieving performance requires that addresses be spread across multiple banks. Correctness depends upon somehow managing the referencing pattern, and avoiding references to banks that are busy.

When parallelism can be found in the memory referencing pattern, an interleaved memory system will provide much higher sustained performance for large scientific applications than any cached scheme of comparable cost. However, the management complexities and exposed parallelism of the interleaved approach have prevented all but supercomputer designers from building such memory architectures.

One major problem in building an interleaved memory system is managing the status of several simultaneously outstanding references. In a traditional scalar or scalar/vector computer, a hardware bank scheduler, or "stunt box", is required to track the busy status of each bank, watch each memory reference address, and prevent conflicts (by temporarily suspending all or selected portions of execution). Details such as out-of-order data returns can complicate the picture further. The complexity of such a scheduler grows as the square of the number of pending references to be managed.

Our VLIW computer system provides enormous memory bandwidth using an interleaved memory system, without memory-reference scheduling hardware and without a data cache. This is a major advantage of the VLIW approach in building a scientific computer.

6.4.1. Memory Implementation Details

Software sees a seven beat memory reference pipeline in the TRACE. The pipeline stages look like this:

0. The program says LD R1, R2, R3. R1 and R2 are added to form a virtual address. R2 may be replaced by a 6-, 17-, or 32-bit immediate constant.
1. The virtual address is looked up in the TLB.
2. The physical address is sent over the buses to the memory controller.
3. The desired RAM bank starts cycling.
4. RAM access continues.
5. Data is grabbed from the RAMs on the memory controller.
6. Data is sent over the buses to the CPU; simultaneously, ECC is checked.
7. Data is written into the register file, and the CPU can use the data in R3.

Like the floating point pipelines, the memory pipelines are "self-draining"; loads specify the destination register when the operation is initiated, and a hardware control pipeline carries the destination forward, tagging a data bus with it in the cycle when the data is sent to the CPU. This simplifies interrupt handling and the generation of "compensation code" for off-trace branch cases, when compared with the "pusher/catcher" approach found in most horizontally microcoded attached processors.

In a fully configured TRACE, four memory references may be started in each beat, to four independently generated addresses (one per I board). When each of these references is a 64-bit reference, this corresponds to a memory bandwidth of 492 megabytes per second.

However, a number of restrictions must be met:

- At most one reference may be initiated on any individual controller.
- No two references may be initiated which would require the use of the same bus to return their data.
- No two references should be initiated to the same RAM bank within four beats of each other.
- The total number of ILoad, FLoad, or Store buses used must not exceed the number available.
- The available number of register file write ports must not be exceeded.

In order to satisfy some of these requirements, the compiler must know quite a lot about the memory addresses being generated by the program. For example, it must be able to guarantee that the addresses for two simultaneously-issued LOAD operations will never be equal modulo the number of memory controllers.

6.4.2. The Disambiguator

The *disambiguator* is the module of the compiler which passes judgment on the feasibility of simultaneous memory references. Memory reference disambiguation — distinguishing between references which can be to the same location and those which cannot — is required in order to find parallelism among array references. When a loop is unrolled, for example, the disambiguator is called upon to answer whether a store into $C(I)$ can be moved above a reference to $C(I+J)$. The disambiguator builds derivation trees for array index expressions and attempts to solve the diophantine equations in terms of the loop induction variables.

Relatively simple extensions to the disambiguator allow the code generator, as it schedules memory references, to ask for any two references, "can these conflict, modulo the number of memory banks"? The answer can be "no", "yes", or "maybe". When the answer is "no", references can be scheduled simultaneously, at very high bandwidth, without any memory-bank management hardware. When the answer is "yes", the operations will not be scheduled simultaneously. When the answer is "maybe", as in the case of references to two arrays passed in as arguments to a subroutine (so that their base addresses are unknown), the compiler has to treat this as a conflict for certain resources, but may overlap the utilization of other resources, because the hardware provides a "bank stall" mechanism (described below).

6.4.3. Virtual Memory

Virtual memory posed a special challenge for a VLIW architecture issuing multiple memory references in every beat. Given that address translation is pipelined, TLB misses are not detected until several beats after the memory reference has been initiated. Since memory reference pipelining is exposed, this presents no problem; no computation could possibly depend upon the result, and we have several cycles in which to determine the correctness of the reference. On a TLB miss, hardware aborts the reference, and signals the processor to switch to Trap Mode to handle the failed reference. However, trap handling code cannot just load the TLB with the appropriate translation and return to the instruction; several more instructions have executed since the original failing operation, and they cannot be correctly reexecuted.

Each I board incorporates a “history queue” mechanism used by the trap handling code, which records uncompleted memory references and their virtual addresses (these memory accesses must then be handled by the trap code). These queues are read and the TLB contents are updated (or page faults are taken); the references are then replayed via special instructions which allow the queue contents to be reissued as new operations. As the queues are four entries deep, up to sixteen independent TLB misses can be pending on a single entry to the trap code.

The trap handling code is standard machine code resident at a specific physical address. It is executed with instruction stream virtual addressing disabled, but is otherwise normal; early versions were written almost entirely in C. No “microcode” is present anywhere in the processor; this is as close as we come to it. This provides great flexibility in the virtual architecture exposed to processes. (For instance, “copy-on-write” is a very simple change to the trap code, not a hardware change.)

6.4.4. Memory Summary and Comparisons to Earlier Work

Several major advances in the memory system beyond earlier research are incorporated in the Multiflow TRACE system:

- Only relative disambiguation is necessary. Unlike earlier proposed VLIW architectures, the presence of a full crossbar between address generators and memory controllers means that the disambiguator need only answer “is $\langle \text{exp1} \rangle$ ever equal $\langle \text{exp2} \rangle$ modulo N”, and not “what is the value of $\langle \text{exp1} \rangle$ modulo N”. This greatly improves the likelihood of successful disambiguations, particularly in subprograms where array base addresses cannot be known.
- The same datapaths are shared between intra-CPU and Memory-to-CPU traffic. This concentrates the bandwidth and better accommodates the bursty nature of computations, providing higher sustained performance without additional costs.
- A “bank-stall” mechanism was devised. When a given RAM bank is accessed, that bank goes busy for four beats. In cases when the disambiguator answers “maybe” to a bank conflict, the compiler has the option of moving references into potentially conflicting schedule positions. In this case the memory will “bank-stall” the CPU if an actual conflict occurs, until the bank busy time is satisfied. This “rolling the dice” can improve performance.

We believe that this software-managed parallel memory system is an important architectural breakthrough. It allows much higher memory performance than would otherwise be possible.

Although a run-time memory reference scheduler has more perfect information than a compile-time disambiguator (it sees no “potential” conflicts, only the real ones) it can do less in the way of scheduling its way around conflicts when they arise. Compile-time scheduling, with a larger perspective on the schedule, is more able to fill conflict times with useful work than hardware schedulers, which typically can only suspend execution until the conflict is resolved. Furthermore, it is dramatically simpler and less expensive to build a highly parallel memory system when no centralized control unit is required to verify the memory reference pattern.

6.5. Instruction Fetch Considerations

Fetching and managing the execution of 1024-bit instructions in a pipelined machine posed some interesting challenges.

We implemented a physically distributed, full-width instruction cache. Bits of the instruction word are cached on the boards that they control; the processor’s master sequencer (the “GC”) contains the cache tag and control logic. The cache is built out of 35ns 64K static RAMs, and holds 8K instructions with a total bandwidth of 984 MB/second. In a fully configured machine, this is 1 megabyte of cache. It is virtually addressed and process tagged; flushing the cache is required only when we “run out” of hardware process tag values, not when we context switch.

Instruction virtual addresses are translated to physical addresses during cache refill through a dedicated instruction-stream TLB. This TLB has 4K entries, and is process tagged, with an “Address Space ID” (ASID) hashing scheme (like data TLBs) to improve multiple process hit rates.

Instruction fetch is fully overlapped with execution, and never stalls or restrains the processor, except on cache misses. We provided an extremely large cache to minimize the overall miss rate, and took extreme care to ensure that we could refill the cache at high speed on a miss.

6.5.1. The Instruction Encoding Format

Most programs contain sections which have lots of parallelism that our compacting compiler can find. In these parts of the code, many operations can be packed into each instruction. To maximize performance, for these parts of the program, we want a very wide instruction capable of independently expressing as many operations as possible. However, other “suburbs” program sections often have much less available parallelism, so that only a few operations will be inserted into each instruction word, and longer sequences of less filled instructions will be generated. If we have optimized the computer for the highly parallel sections, then in these suburbs we will have many functional units idle for many instruction cycles. This means that large portions of the instruction word will contain only no-ops, and will substantially increase the memory size of the program without contributing to its performance.

Machine designers have historically dealt with this dilemma by compromising: compressing their instruction encodings by preselecting those combinations of operations that they expect will be most commonly used. This then required the compiler to find and use the “patterns” that the designers had provided, if the highest performance was to be obtained. We believe these encoding schemes work out poorly in conjunction with a compiler; we pursued a quite different solution.

We place no restrictions in the instruction on what combinations of operations can be invoked simultaneously. Object code size is minimized in a different way: we use a variable-length main memory representation of the fixed-length machine instruction. That is, the instruction cache outputs a fixed-length 1024-bit instruction in each clock cycle; bits of the instruction word are directly wired to the functional units that they control. The architecture in this sense has a fixed-size instruction. However, we use a main memory instruction representation that eliminates the no-ops, affording a significant space savings.

Implementation of this variable-size memory instruction format had to satisfy a number of serious constraints. One constraint was that the instruction format not penalize execution of in-cache instructions. When the instruction cache is loaded, the control information for the functional units must be in the "right places" so that the instruction fetch pipeline length remains minimal.

A second constraint is that refilling the cache on a miss must proceed at the highest possible rate, without a huge amount of hardware dedicated solely for filling the cache. Since the TRACE system possesses massive main memory bandwidth through its use of an interleaved memory system and many buses, this means that it must be possible to control the cache refill without inspecting and interpreting each word as it comes from memory.

The instruction set must facilitate an easy-to-implement correspondence between the Program Counter, cache locations, and main memory locations, so that variable-length instructions can be "unpacked" quickly into a fixed-width cache.

Given that variable-length instructions are being fetched from a parallel, interleaved memory system, the "schedule" of what word will be on each bus in each cycle, and knowledge of which field of the instruction cache is to receive that value, must be produced by a control unit in real time as the data is returned from main memory. For a practical implementation, this requires that the schedule be precomputed by that control unit. The instruction representation must be such that this control unit is as simple and fast as possible.

We store instructions in main memory in blocks of four. Each block is preceded by four 32-bit "mask" words, which specify which 32-bit fields of the instruction are present in the block; the others are filled in the cache with zeros (no-ops).

The cache refill engine fetches and interprets the mask words. It never has to see or process actual instruction fields destined for the various functional units. This engine decides upon the schedule of buses to be used, initiates the instruction field fetches, and then tags the fields as they fly by on the ILoad buses so that they are steered to the proper functional units' cache words. The real-time overhead of this scheme is very low, since the actual cache refill proceeds at the maximum memory bandwidth and cpu bus bandwidths (the same buses are used in refill as are used for general computation).

This cache refill engine is perhaps the most complex piece of hardware in the TRACE, starting up enormous numbers of pipelined loads from memory and then directing them to the various instruction cache memories distributed throughout the machine. For sequential code, the mask interpretation is overlapped with the execution of the current block of instructions, so the operation of this cache refill engine represents a low overhead on the overall performance of the machine.

6.5.2. Branching

The architecture includes compare-predicate operations, rather than test operators and condition codes. We found it helpful to include compare instructions which could write the general registers, to allow evaluation of IF chains without branching. The architecture includes a special one-bit-wide 7-element register file, called the "branch bank", which can hold the result of compare (and other) operations, and which can be used to control branching. This allows the compiler to perform register allocation on branch bank elements, and move compare operations independently of the actual branches. A typical code sequence would look like:

CEQ R1, R2, BB(R3)	Write BB 3 with 1 if R1 == R2, else write BB 3 with 0
BRANCH (R3) LABEL	The "branch_test" field selects R3

The branch operation can be issued in the early beat of every instruction; part of the immediate field is used as the displacement for the branch. The branch is taken if the selected branch bank element is a 1. This branching structure resembles the "delayed branch" of other RISC machines, in that operations following the compare-and-branch are unconditionally executed while the branch target is fetched.

Conditional branching becomes an interesting problem as we attempt to fill wider instructions.^{Fish83} Conditional branches occur every five to eight operations in typical programs; if we try to compact many more than five operations together, some mechanism will be required to pack more than one jump into a single instruction. We provided a multiway jump in the TRACE processor with multiple independent targets, with a software-controlled priority scheme.

Consider two jumps, with unique target addresses, which are initially sequential in the source program. If we want to pack them into a single execution cycle, we must establish a priority relationship between them, which defines which target address to branch to in the case that more than one of the simultaneous tests are true. The "highest priority" test whose condition is true provides the next address for execution. The priority relationship is driven by the original ordering of tests in the sequential program. The test that was originally first in the sequential program must be the highest priority; in the original sequential program, if the first test were true, then the second jump would never have been executed. Therefore, when we pack them together, we must arrange to ignore the results of the second lower-priority test if the first higher-priority test is true.

Each I unit can perform one test per instruction. A 32-bit "branch target adder" on each I board adds the current program counter to an immediate field of the instruction word. This computation yields a potential branch address. The branch arbitration mechanism determines which of four tests being performed simultaneously (on different I units) is the highest priority true test, and distributes the branch target associated with that test (defaulting to PC+1 when none of the four tests is successful).

Each I unit has nine bits of instruction word that control branching. Three bits ("branch_test") select an element from branch bank 0; another three select an element from branch bank 1. The values of the two selected bits are logically ORed to determine if "this board wants to branch". Three more bits (hidden in the immediate field) are defined by software to specify the relative priority of this test versus that of the tests being executed on the three other I units. These priority bits

show which other I boards it can preempt if its branch is true (which it does by sending "inhibit" signals to them directly). If no I board has a TRUE branch condition, then a central system controller board supplies a default PC value. Otherwise, the I board which has a TRUE branch condition and no inhibits is enabled to drive the new PC value onto the backplane.

This scheme is elegantly simple. It is fast, requiring only two gate delays (and one backplane traversal) to effect the arbitration. It is software-controlled, so that the compiler can adjust the relative priorities of the branches for each instruction. It allows the rapid selection of one of five potential next-addresses during every instruction; the fetching of the next instruction from that address is fully overlapped with the execution of the current instruction.

7. Exceptions and Optimizations

Some things you might take for granted, like traps and interrupts, have some subtle consequences when you try to rearrange execution order. We found we needed some unusual architectural features to enable more compiler code motion and optimization than a traditional approach to exception handling would have allowed.

Consider a FORTRAN loop that contains an array reference which is accessing across a row. If this loop is unrolled a number of times, and we allow the code generator to move the LOADs above the conditional branch that tests for the last iteration of the loop, several LOADs may be issued to addresses beyond the end of the program's current address space. The conditional jump will be all set to exit the loop, so that these references will be ignored (their data will never be used); but a conventional virtual memory system would terminate the program with a "Bus Error".

The architecture includes a special set of LOAD opcodes used by the compiler in the case when a LOAD moves above a conditional branch. When trap handling code sees one of these opcodes on a TLB miss, if no valid translation can be established for the reference, execution continues; the target register is loaded with a "funny number" to help catch bugs. These special opcodes are used only when necessary; we don't give up the helpful "Bus Error" traps when we don't have to, to assist in program fault isolation. This technique enables the compiler to be much more aggressive in code motions involving memory references.

A similar problem exists in floating-point exception detection and handling. Consider the fragment: IF (A .NE. 0) C = D/A. It's very much in the interests of performance to move divides up in the schedule; they take a long time. But if we want to detect division by zero, we must wait until the test has completed before initiating division.

Here again, we provided some assistance in the architecture. The processor has several floating exception modes, one of which is called "fast mode". In fast mode, floating exceptions cause traps only if the result is being written to the store file, being used in a compare, or being converted to integer form. Otherwise, a NaN ("Not-A-Number") or infinity will result from the offending computation, but no exception will be generated. As NaNs and infinities tend to propagate, any computation using the offending result will eventually cause a fault (by writing something to memory, for example). The trap will not occur at the most perspicuous point, but overall execution speed will be higher. (Note that floating underflows escape our notice in "fast mode", in that they are flushed to zero. We find this not to be a problem for many programs, and provide lower performance modes in which exceptions are detectable immediately.)

8. UNIX on the TRACE

A VLIW may appear to be an odd sort of CPU to make into a virtual memory timesharing system. Indeed, the original designers of the ELI-512 expected their machine to be useful only as a number-crunching back-end processor.^{Fish83} The problems associated with making this heavily pipelined parallel machine capable of servicing interrupts seemed daunting enough, let alone all the rest: supporting virtual memory on a CPU without microcode, the incredible number of registers that would have to be context switched, extending the architecture and compiler to support systems code in addition to its numerical chores, not to mention that long instruction words might make all the utility programs consume gigabytes of disk space.

We've figured out ways around all of these problems, but it is natural to wonder why we built the TRACE to run 4.3BSD UNIX in the first place. The reason is simple: modern numerical applications programs do much more than perform floating point calculations. They make the usual demands of a system for disk, graphic, and terminal I/O, but they can make these demands at rates far exceeding those of "I/O intensive" systems programs. And scientific applications programmers have the same desires for reasonable and friendly programming environments that system programmers do. Fulfilling all these demands, particularly for performance, with a smoothly integrated front-end/back-end processor seemed difficult and unnecessary, so we built the operating system to run directly on the CPU.

8.1. Support for a Multiple Process Environment

We have already described many of the architectural features needed to support an operating system: the instruction and data TLBs needed for virtual memory; mechanisms and constraints for dealing with exceptions; and the desire for interruptability that led to our pipeline handling philosophy.

The TRACE supports its multiuser operating system in the usual way. Appropriate protection modes and privileged instructions are provided so that the user process environment is maintained. All accesses to mapping hardware, I/O stimulus instructions, and the PSW are carefully protected. A limited set of traps to system mode are provided for system calls and breakpoints.

We were concerned about the effects of running multiple processes, and the overall impact that context switching would have on performance. Our goal was to support about as many users as would be comfortable on a large supermini but to support order-of-magnitude larger computations than current superminis could support.

Context switching is often considered to be simply the cost of saving and restoring registers. But the actual cost of a context switch also includes the interrupt time, scheduling overhead, and any penalty for cache purging and cold-start.^{Clar85} On many machines, the cost of purging the virtual address translation and instruction caches dominates register saving. The TRACE provides very large instruction and translation caches (see Sections 6.4 and 6.5), which are process tagged with an 8-bit "Address Space ID", or ASID. No purging of the instruction cache or translation buffers is necessary on a context switch; caches must be purged only every 255 address space mapping changes, when the set of ASIDs overflows.

Updating the ASID registers is cheap, so the high available memory bandwidth in the system permits a complete context switch in 15 microseconds. This figure holds in any machine configuration, because usable memory bandwidth increases as the number of registers. This performance is comparable to other machines that are trying to support our number of users.

8.2. Interrupts

Interrupt handling is almost entirely conventional. There is a priority interrupt system, with maskable interrupts from each device. When an enabled interrupt request arrives, execution suspends, the processor changes state, and execution resumes at a "trap" address. Since the pipelines are self-draining, after the maximum pipe depth time, all of the state of the processor is either in general registers or in main memory. A few hand-coded instructions begin saving registers while the pipelines drain; after several instruction times we enter C code to process the event.

8.3. Input/Output

Given an exposed architecture where the compiler knows about the machine resources being used throughout the system, it's difficult to allow I/O to "cycle steal" or otherwise share hardware resources on a fine-grained basis with program execution.

A memory-mapped I/O scheme would have required the CPU's memory interface to deal with devices with two distinct speeds: fast (to memory) and slow (to I/O devices). We chose not to implement our I/O this way. Instead, the CPU interacts with its devices through a surrogate called the I/O Processor (IOP). The IOP is based on an MC68010 with a multiported high bandwidth buffer memory and a "DMA engine" which can read and write blocks of main memory at half of peak memory bandwidth. The IOP interfaces to a VMEbus, a standard 32-bit asynchronous bus where the device controllers reside.

When the DMA engine wants to read or write main memory, it signals the GC. The GC suspends processor execution and allows pipelines to drain. The DMA engine then talks directly to memory at high speed; for example, 10 MB/s of I/O consumes only 4% of the machine's cycles in the largest CPU configuration. Execution resumes as soon as a burst of data has been transferred.

The I/O processor talks to the CPU using a bidirectional interrupt and a channel command protocol in main memory. Device drivers run on the I/O processor, a scheme which minimizes interrupts and CPU involvement in I/O operations. The IOP is also responsible for bringing the system up. A small operating system on the IOP supports execution of diagnostic and bootstrap programs.

We have devised a generic set of drivers on the TRACE side for each class of device on the IOP (disk, tape, ethernet, and terminal) which are very small, and which interface to device-specific drivers on the IOP. We have also implemented an I/O configuration system where all possible drivers are present (at a small cost in memory) and system device configuration is changed by editing a file on the diagnostic file system before booting UNIX.

8.4. Systems Code on a VLIW

The hundreds of thousands of lines of code which make up the UNIX kernel and utilities do not know they're running on a VLIW. One of our compilers is for the C language. Nearly all of the UNIX utilities, and a large chunk of the kernel, are written in portable C. (By actual count: 300 lines of assembly and 64K lines of C in the kernel; 1100 lines of assembly and 700 lines of C in the trap handlers.) The fact that our compiler performs exotic optimizations like inter-block compaction and transforms the code into a parallel form is irrelevant. We compile these programs and they do what they're supposed to do; *grep* doesn't know it's stretching the frontiers of technology, it just greps along at a terrific rate.

Trace Scheduling compacting compilation was originally conceived for numerical applications; we expected to run into problems handling systems code. The systems code in UNIX differs in several respects from numerical code. Systems code makes pervasive use of pointers, which leads to more difficult compiler optimization problems. The code tends to have even smaller basic blocks than numerical code. And most important, systems code has proportionately many more procedure calls than numerical code.

Pointers and small basic blocks have not been a problem. In fact, procedure call overhead seems to be the only issue that has required special attention. Performance on systems code is quite good (the C and Fortran compilers share a common back end).

The TRACE provides no special architectural support for procedure calls (other than the large memory bandwidth already built in). During the design, we considered several hardware mechanisms intended to minimize procedure call/return overhead, but none of them was both a clear performance win and clearly feasible. We decided to rely on the compiler to be clever with its use of registers and procedure inlining, and to develop a global register allocating linker, which builds a global call graph and minimizes register saves (currently in the works).^{Wall86} We expect this work to be complete by the time of the conference presentation, and will report on it there.

When we initially debugged UNIX on the TRACE, we restricted traces to basic blocks, and disabled loop unrolling; compiler heuristics for how much unrolling to perform had not yet been installed, and code grew unmanageably. Those heuristics are now in place, and their performance is remarkably good. The full compacting compiler optimizations work well for a wide variety of systems code, including the kernel itself, without undue code growth.

This result surprised us somewhat; we hadn't anticipated as much improvement on systems code as we got. Good performance on systems code is very desirable, as it restrains the proportionate growth of operating system overhead that is usually encountered on a parallel machine. Unlike "coarse-grained" architectures where systems code runs on a single scalar unit (and can become a substantial bottleneck), we retain the same OS-to-user balance found on more traditional systems.

9. Code Size: Initial Results

The "no-op" fields of an instruction are not represented in main memory, so the object code size of a program is directly proportional to the number of operations in the compiled program. There are thus three components to consider when comparing VLIW code density to that of other architectures:

- the number of bits required within the instruction set to express a given operation;
- the succinctness, or lack thereof, with which common high-level operations (like procedure call) can be expressed in the instruction set; and
- the number of new operations introduced through compiler optimizations and loop unrolling.

The VLIW encoding of each operation is roughly on par with other RISC machines. It is a three address architecture, all loads and stores are explicit, and there is minimal instruction encoding. The code expansion per operation is probably around 30 -- 50% when compared to a tightly encoded machine like the VAX or Motorola 68000. The variable-length main memory instruction encoding has an associated overhead of a few bits per operation, which coupled with main memory alignment constraints adds roughly an additional 5 -- 10%.

Operations that cannot be initiated in a single instruction cycle are broken down into constituent sub-operations. These constituents are usually substituted inline, although certain operations such as the block register save and restore associated with procedure call are implemented via special subroutines. The overall code expansion due to this, as compared to a machine like the VAX that has an extensive library of microcoded "subroutines", is difficult to quantify, but is probably in the neighborhood of 10 -- 20%.

The compiler performs an enormous number of optimizations, most of which reduce the number of operations in the program, but some of which increase the number of operations with the goal of increasing parallel execution. The three most notorious code-expanders are inter-block trace selection (which can produce compensation code), loop unrolling, and inline procedure substitution. All three of these are currently automatic and have been tuned to avoid undue code growth. These optimizations can increase the size of some small fragments of code by a large factor, but their overall effect seems to be to increase code size by a factor of around 30 -- 60%, although the user can increase or decrease these factors arbitrarily through the use of compiler switches.

Several large (100K -- 300K lines) FORTRAN programs have been built on the TRACE. After unrolling and trace selection, the code size is approximately 3 times larger than VAX object code (compiled with the VAX/VMS FORTRAN compiler).

The concern about code size led us to implement a shared-libraries facility very early in our UNIX development. This has substantially reduced the size of the UNIX utilities images. The UNIX utilities consume approximately 20MB of disk space on a VAX, and approximately 60MB on our VLIW using shared libraries.

UNIX has been running on the TRACE and supporting its own development for some time. The principal advantage of Multiflow's parallel processing technology is that it is transparent to its clients. Thus, most of the challenging problems in developing an operating system and programming environment for the TRACE come not from its VLIW nature but from our intention to make the system into a first rate environment for high performance engineering and scientific computation. A thorough discussion of our approach is beyond the scope of this paper.

10. Summary and Future Work

This paper has introduced the Multiflow TRACE Very-Long-Instruction-Word architecture. Before this machine was built, some designers and researchers predicted that the negative side-effects of the VLIW/compacting compiler approach (object code size, compensation code, context swap time, and procedure call/return overhead) would likely swamp the machine's performance gains. These predictions were wrong: we slew some of these dragons with cleverness, tamed a few, and couldn't even find the rest.

It is too early to be able to separate out all the different contributions to performance in the TRACE. Our future work will concentrate on quantifying the speedups due to trace scheduling vs. those achieved by more universal compiler optimizations. We will also be examining the efficacy of memory-bank disambiguation, speed/size tradeoffs of the fixed and variable instruction encoding schemes, and instruction cache usage statistics.

Our conclusion should be unsurprising: given an implementation technology, the best way to use it is to build a VLIW. If you build a standard scalar machine instead, you pass up significantly higher performance at only slightly higher cost; the extra functional units are cheap compared to the overhead of building the computer in the first place (memory, control, etc.). If you build a vector machine instead, the parallel hardware you build "turns on" only occasionally, and the speed of some vector code is all that will be improved. And if you build a multiprocessor instead, you pay the full overhead of instruction execution and run-time synchronization per functional unit, without getting the fine-grained speedups a VLIW can offer.

11. Acknowledgements

Thanks go to Chris Genly and Ben Cutler for help with the diagrams in this paper, and to Helen Spontak for easing scheduling constraints.

References

- Kate85. Manolis Katevenis, *Reduced Instruction Set Computer Architectures for VLSI*, MIT Press, Cambridge, Mass., 1985.
- Tjad70. G.S. Tjaden and M.J. Flynn, "Detection and parallel execution of independent instructions," *Transactions on Computers*, vol. C-19, no. 10, pp. 889-895, IEEE, October 1970.
- Fost72. C.C. Foster and E.M. Riseman, "Percolation of code to enhance parallel dispatching and execution," *Transactions on Computers*, vol. C-21, no. 12, pp. 1411-1415, IEEE, December 1972.
- Fish83. Joseph A. Fisher, "Very Long Instruction Word Architectures and the ELI-512," *Proceedings of the 10th Symposium on Computer Architectures*, pp. 140-150, IEEE, June, 1983.
- Elli86. John R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, MIT Press, Cambridge, Mass., 1986.
- Fish79. Joseph A. Fisher, "The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of

Processor Scheduling with Resources," *Technical Report COO-3077-161*, Courant Mathematics and Computing Laboratory, New York University, October 1979.

Henn81.

John L. Hennessy, N. Jouppi, F. Baskett, and J. Gill, "MIPS: A VLSI processor architecture," *Proceedings of the CMU Conference on VLSI Systems and Computations*, pp. 337-346, Computer Science Press, October 1981.

Radi82.

George Radin, "The 801 Minicomputer," *Proceedings SIGARCH/SIGPLAN Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 39-47, ACM, March 1982.

Toma82.

Robert M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *Computer Structures: Principles and Examples*, pp. 293-305, McGraw-Hill, 1982.

Thor70.

James E. Thornton, *Design of a Computer: The Control Data 6600*, Scott, Foresman & Company, Glenview, Illinois, 1970.

Acos86.

R.D. Acosta, J. Kjelstrup, and H.C. Torng, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors," *IEEE Transactions on Computers*, vol. C-35, no. 9, pp. 815-828, September, 1986.

Seit85.

Charles Seitz, "The cosmic cube," *Communications of the ACM*, vol. 28, no. 1, pp. 22-33, ACM, January 1985.

Walt87.

David L. Waltz, "Applications of the Connection Machine," *Computer*, vol. 20, no. 1, pp. 85-97, IEEE, January 1987.

Fish81.

Joseph A. Fisher, "Trace Scheduling: A technique for global microcode compaction," *Transactions on Computers*, vol. C-30, pp. 478-490, IEEE, July, 1981.

Fish84.

Joseph A. Fisher and John J. O'Donnell, "VLIW Machines: Multiprocessors We Can Actually Program," *CompCon 84 Proceedings*, pp. 299-305, IEEE, 1984.

Elli84. John R. Ellis, Joseph A. Fisher, John C. Ruttenberg, and Alexandru Nicolau, "Parallel Processing: A Smart Compiler and a Dumb Machine," *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*, ACM SIGPLAN Notices, June 1984.

Pfis85. Gregory F. Pfister and V. Alan Norton, "Hot-Spot Contention and Combining in Multistage Interconnection Networks," *Transactions on Computers*, vol. C-34, pp. 943-948, IEEE, October 1985.

Smit82.

Alan Jay Smith, "Cache Memories," *ACM Computing Surveys*, ACM, September 1982.

Clar85.

Douglas W. Clark and Joel S. Emer, "Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 31-62, February 1985.

Wall86.

David W. Wall, "Global Register Allocation at Link Time," *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, ACM SIGPLAN Notices, July 1986.

RETROSPECTIVE:

The DASH Prototype: Implementation and Performance

Daniel E. Lenoski

Silicon Graphics
lenoski@sgi.com

James P. Laudon

ZSP Corporation
laudon@zsp.com

Our paper entitled "The DASH Prototype: Implementation and Performance" was given at the 19th ISCA in Gold Coast, Australia in May of 1992. This paper outlined our implementation experience and initial performance details of DASH, the first hardware implementation of the ccNUMA architecture. DASH was a large multi-faceted research project at Stanford University led by John Hennessy, Anoop Gupta, Monica Lam, and Mark Horowitz. The overall goal of DASH was to break the scalability barrier of bus-based SMP machines and provide the massive parallelism of distributed memory while maintaining the shared-memory paradigm. While there had been previous switch-based SMPs built in the early 1980s (e.g., the Cray X-MP, Univ. of Illinois Cedar, BBN TC-1000, and IBM RP3), DASH added hardware support for global cache coherence. Hardware cache coherence improved processor performance and removed the burden of coherence from the user or compiler. During the late 1980's our group was not alone, there were efforts at MIT (Alewife and J-Machine), University of Wisconsin (Multicube), Encore Computer (Gigamax), Kendall Square Research, and the IEEE Scalable Coherent Interface (SCI) standards effort, but ours was the first to build a hardware implementation of this new class of machine.

The high-level structure of DASH was a collection of nodes, each including one or more processors and a portion of the global memory, connected by a scalable interconnect (a 2-D mesh). Directory-based coherence, originally proposed by Censier and Feautrier in the late 1970s, was employed since it removed the need for the global bus found in snoopy systems. While the original directory schemes used a central memory/directory, moving to a distributed organization scaled memory bandwidth naturally with the number of processors.

With this fundamental system structure in mind, and previous studies showing the potential of distributed-directories (see the Agarwal/Hennessy paper on directories in this collection), work began on the DASH prototype.

DASH Prototype Goals and Timeline

Scaling the cache-coherent SMP model to hundreds of processors raised many questions in the area of processor and system architecture, operating systems, compilers, programming languages, and parallel applications. We chose to build an actual hardware prototype of the architecture to address these questions as well as to:

- Understand the hardware complexities of actually building this type of machine.
- Provide more insight into the performance attributes of a real ccNUMA machine.
- Allow a comparison of real applications' complexities and performance (not simply small simulated kernels) among highly parallel shared-memory programs and their message-passing counterparts.

The difficulty of implementing a distributed directory protocol was of serious concern since it amounts to replacing the software controlled network interfaces on message-passing machines with hardware control for sending network messages to fetch remote memory and maintain cache coherence. At the time, it wasn't clear if this hardware complexity was tractable, and even if it was, would the performance of a ccNUMA be competitive with message-passing systems?

We began detailed architecture work on the system in fall of 1988. Early on, we made a choice to leverage an existing SMP system as our base

node because these machines provide the necessary hooks for controlling the processor caches from their bus interfaces. We were anxious to utilize a RISC-based SMP, and the recently announced SGI 4D/240 series was the only such machine on the market at the time. This choice turned out to be very fortuitous, since utilizing an existing system allowed us to leverage much of the system hardware and software and concentrate our efforts on the unique ccNUMA hardware and software.

Initial power-on of the prototype system was in the Fall of 1990 and a 16 processor system was stable in the Spring of 1991. We then started work on a larger 64 processor system, which resulted in a stable 48 processor prototype in the Spring of 1992. Nagging problems with our ribbon-cabled mesh links prevented us from reaching the goal of 64 processors in a single system (a 4x4 mesh of 4 processor nodes), but we were able to learn much from the 48 processor prototype.

Innovations in DASH

During the architecture phase of the project, our focus was on the coherence protocol and mechanisms that would minimize memory latency and maximize memory bandwidth. In addition, we realized that hiding memory latency would also be key since the distributed structure of a large ccNUMA would invariably lead to longer memory latency. Likewise, support for large-scale parallelism demanded that we pay attention to synchronization and inter-processor communication. Being one of the first to tackle these problems in the context of a ccNUMA machine, these goals led to many innovative solutions. These included:

- Software-controlled non-binding cache line prefetch to hide latency and increase memory pipelining.
- Release-consistency support with fence/memory barriers to help hide store latency.
- Queue-based test-and-set locks to allow efficient contended spin-locks.
- Fetch&Inc and Fetch&Dec (borrowed from the NYU Ultracomputer, but without combining) for support of efficient barrier synchronization and distributed queues.
- Update coherence and deliver instructions which provide low latency inter-processor word and cache line communication respectively.

The actual hardware implementation phase also demanded innovative solutions such as:

- An efficient “forwarding” coherence protocol which minimized latency for accessing dirty data and writing to shared cache lines.
- Support for both invalidate and update coherence within the same directory protocol.
- Separate request and reply paths that prevented dead-lock on the normal memory requests together with retry mechanisms that handled race conditions in the distributed directory protocol.
- A high-bandwidth DRAM directory access path which performed read-modify-write cycles under the shadow of the main memory’s fetch of 16-byte memory blocks.
- One of the first lock-up-free caches that implemented a remote access cache to track outstanding memory references and supplement the processor caches with features such as prefetch.

Lessons Learned

As one would expect, building and using the DASH prototype led to many new insights and lessons that were both positive and negative. The most positive result was that it was feasible to build a ccNUMA machine and to achieve good performance on highly parallel shared-memory applications. Furthermore, by analyzing the logic in the directory and network interface, the prototype demonstrated that adding hardware cache coherence added only 10% additional hardware over a non-coherent MPP system structure. Another lesson was that with close attention, it was possible to keep remote-to-local memory latency to within a 3 to 1 ratio. Several features included in the prototype proved very successful. Operations such as prefetch proved to be very powerful in hiding memory latency and improving the pipelining of memory operations. Fetch&Op performed at memory also greatly reduced the overhead of barrier-type synchronization by reducing the serialization time for atomic counter operations.

Other features that did not yield as much performance improvements as expected were queue-based locks and update and deliver operations. While these operations could greatly aid in specific low-level communication, the overhead of general communication associated with inter-processor

data sharing tended to swamp out the incremental enhancements that these operations provided. This was especially true on the prototype hardware where our remote access cache was as close as we could get the data to the processor (thus reducing latency by no more than a factor of 3).

Another somewhat unexpected result was the negative impact of using a bus-connected inter-node interface. Since memory operations needed to cross the processor's local bus twice and memory home's bus once, the resulting memory bandwidth when all processors are accessing remote memory was no better than one-third that of local memory. In fact, DASH's bus-bandwidth was a greater limit on global remote memory bandwidth than network bisection bandwidth. While not inherent to ccNUMA systems, this issue illustrated the limitations of simply extending a bus-based SMP with a ccNUMA network interface card.

The advantages of leveraging an existing SMP was also one of the indirect, but very positive, lessons from the DASH project. Using an existing SMP allowed a small university team to focus their attention on the important task of architecting and designing the hardware necessary to implement a ccNUMA machine. In addition, the choice of the SGI 4D/240 as the base node helped in the quick development of DASH, as its modest level of integration (by today's standards) allowed us to work primarily at the board level with PALs and FPGAs. Working at this level of integration reduced both design and debug time. There were some compromises due to leveraging an existing machine, but it reduced the time from concept to running real applications under Unix to less than 2.5 years. This increased the impact of the actual DASH hardware and helped validate the feasibility of the ccNUMA architecture.

Conclusions

The impact of the DASH project has been felt both in the academia and industry. Scalable shared-memory multiprocessors continue to be a

hot topic of research. DASH helped validate the viability of the ccNUMA approach and provide a baseline to evaluate improvements in coherence protocols, scalable directory storage, and alternative system architectures such as COMA. ccNUMA systems have now been commercialized by a number of vendors including HP/Convex, Silicon Graphics, Sequent, HaL, and Data General. The DASH prototype helped pave the way for these commercial developments by detailing many of the fundamental design problems with ccNUMA machines and demonstrating that the shared-memory paradigm could be scaled and realize both good performance and good cost-performance.

Building the DASH prototype would never have been possible without the hard work of a number of individuals. These included our co-authors: Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. Additional contributions to the hardware development were made by Kourosh Gharachorloo, Wolf-Dietrich Weber, Mark Horowitz, Tom Chanak, John Maneatis and Monica Lam. Help from Silicon Graphics, namely Jim Barton, Forest Baskett, John Burger, Doug Solomon and John Carlson, was also instrumental. Dan Lenoski was supported by Tandem Computers during his graduate work. John Toole and Gil Weigand at DARPA provided the funding to support the greater team and build the DASH prototype.

For additional details on DASH see:

- [1] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy, "The DASH Prototype: Implementation and Performance," *IEEE Trans. on Parallel and Distributed Systems*, 4(1)41-61, January 1993.
- [2] D. Lenoski and W.-D. Weber, *Scalable Shared-Memory Multiprocessing*, Morgan Kaufmann Publishers, San Francisco, CA 1995

Efficient Stream Compaction on Wide SIMD Many-Core Architectures

Markus Billeter*

Chalmers University of Technology

Ola Olsson†

Chalmers University of Technology

Ulf Assarsson‡

Chalmers University of Technology

Abstract

Stream compaction is a common parallel primitive used to remove unwanted elements in sparse data. This allows highly parallel algorithms to maintain performance over several processing steps and reduces overall memory usage.

For wide SIMD many-core architectures, we present a novel stream compaction algorithm and explore several variations thereof. Our algorithm is designed to maximize concurrent execution, with minimal use of synchronization. Bandwidth and auxiliary storage requirements are reduced significantly, which allows for substantially better performance.

We have tested our algorithms using CUDA on a PC with an NVIDIA GeForce GTX280 GPU. On this hardware, our reference implementation provides a $3\times$ speedup over previous published algorithms.

CR Categories: D.1.3 [Concurrent Programming]: Parallel Programming

Keywords: stream compaction, prefix sum, parallel sorting, GPGPU, CUDA

1 Introduction

Stream compaction, also known as stream reduction, is an important primitive building block for algorithms that exploit the massive parallelism that is emerging in mainstream hardware [Seiler et al. 2008; Fatahalian and Houston 2008].

Highly parallel algorithms tend to produce sparse data, or data containing unwanted elements, especially if each input element can produce a varying number of output elements. In order to maintain performance, it is often necessary to compact the data prior to further processing steps.

This can be observed in parallel breadth first tree traversal [Roger et al. 2007b; Zhou et al. 2008; Lauterbach et al. 2009]: after each traversal step, the list of open nodes must be pruned of invalid nodes, as otherwise an exponential explosion of nodes takes place.

Similar problems are encountered in many recent publications, e.g. in *ray stream tracing* [Wald et al. 2007], and GPU-based collision detection [Greß et al. 2006]. Another way to think of the compaction pass is as a form of load balancing; a compact input range makes it easier to provide an equal workload for all processors.

Contributions In this paper, we present a novel algorithm for compacting streams on graphics hardware, which offers substantially better performance than previously published algorithms. Most previous algorithms depend on computing and storing a prefix sum for all elements, which is then used in a second stage to move the valid data to a compact range. Our new approach avoids explicit construction of a prefix sum of the same size as the input data, which allows substantial savings in bandwidth and storage. On current hardware, our implementation offers a $3\times$ speedup, compared to previous published algorithms.

Our algorithms require very little synchronization, making use of only implicit atomicity in SIMD operations and global barrier synchronization. This makes our algorithm suitable for implementation on various many- or multi-core processors with wide SIMD instruction sets, e.g. current generation NVIDIA and AMD GPUs and also the upcoming Intel Larrabee GPU.

Our general approach can be applied to several related problems, which we illustrate by a few variations, such as *stream split* and *prefix sum*. We also demonstrate a high performance radix sort, in terms of stream split, which shows performance competitive with the currently fastest published implementation [Satish et al. 2008].

Organization of this paper Section 2 provides an overview of previous work. In Section 3, we describe our new algorithm, including a couple of variations. Section 4 goes into the details of a CUDA implementation. In Section 5, we present and compare detailed performance measurements of our algorithm. Section 6 contains discussion, and we finally conclude in Section 7.

2 Previous Work

The simplest implementation of stream compaction is a sequential algorithm, which is trivial to implement on an uniprocessor machine. The algorithm is shown in Listing 1.

```
1 j ← 0
2 for (i ← 0; i < N; i++)
3     if valid input[i]
4         output[j] ← input[i]
5         j++
```

Listing 1: Sequential compaction algorithm. Valid elements are moved from *input* to *output*.

Implementing an efficient stream compaction on parallel architectures is more challenging: the output location of each element in a stream depends on the state of every element before it. A trivial implementation with synchronization after each element would be very inefficient.

To overcome this, most of the previous approaches are based on performing a parallel *exclusive prefix sum* [Blelloch 1990; Chatterjee et al. 1990]. The prefix sum is performed on a stream containing a 1 for each valid element in the input and 0 for each invalid. The result of this operation is a stream containing, for each element, the number of valid elements preceding it. This information is then used to move each valid element to the new location, as illustrated in Figure 1.

*billeter@chalmers.se

†ola.olsson@chalmers.se

‡uffe@chalmers.se

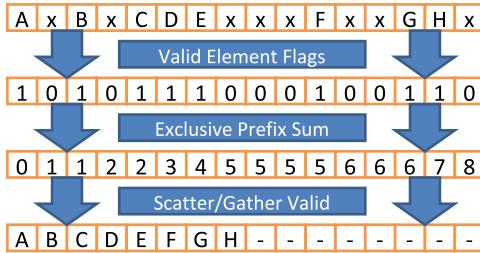


Figure 1: Main steps of performing compaction with a prefix sum. First a prefix sum of the valid element flags is computed. Then a gather or scatter step is used to move the valid input elements into the output vector.

This approach has been implemented on a GPU [Horn 2005]. Since the GPUs at that time lacked support for random write access to memory (*scattering*), a common workaround was to use *gathering*, where a binary search is performed to find the input element corresponding to each output. Both steps of the algorithm, the prefix sum and the $O(N)$ binary searches of $O(\log N)$ complexity, have an overall complexity of $O(N \log N)$.

The same approach was used again later [Sengupta et al. 2006], but the prefix sum is improved by using a *work-efficient* implementation, running in $O(N)$ time. However, the gathering step was unchanged and thus the overall complexity remained $O(N \log N)$.

A somewhat different approach [Ziegler et al. 2006; Greß et al. 2006] is to construct a tree containing the number of valid elements, which is similar to an up-sweep tree [Blelloch 1990]. Performing the gathering step can then be done by searching the tree to find the correct input element corresponding to each output. Again, the time complexity is $O(N \log N)$.

To improve the complexity, the algorithm can be applied to small fixed size chunks [Roger et al. 2007a]. Chunks of size K can be compacted with the earlier algorithms in $O(K \log K)$ steps. The individual compacted chunks are finally concatenated using line drawing hardware on the GPU. This design achieves a time complexity of $O(N)$.

Modern GPUs provide *geometry shaders*, which, together with *transform feedback*, can be used to implement stream compaction in a simple way. However, geometry shaders have so far not been able to deliver competitive performance [Roger et al. 2007a], and our own results confirm this.

Recent GPUs also support scattering, which can be used to replace the gathering stage [Sengupta et al. 2006; Horn 2005]. An implementation of this approach, available in the CUDPP library [CUDPP 2008], achieves an $O(N)$ time complexity.

Stream reduction, and its sibling, *stream splitting*, is intimately connected to sorting; e.g. a recent and fast sort [Satish et al. 2008] uses stream splitting to implement a radix sort.

3 Algorithm

The basic idea for our new algorithm follows the approach taken in Chatterjee et al. [1990] where $N > P$, i.e. the number of input elements is larger than the number of processors. The input stream is divided into P roughly equal, and continuous, ranges. Each processor can then count the number of valid elements in a single range independently. These counts are transformed using a parallel prefix sum, which gives an offset for each processor, where it can start writing valid elements. This enables a third phase to use sequential compaction within each of the P ranges. Listing 2 shows this

algorithm in pseudo code.

With CUDA [NVIDIA 2008], it is straightforward to implement the basic parallel algorithm in Listing 2 by using each thread as an individual processor, but performance will be poor. The reason is that the underlying hardware does not actually consist of a large number of independent scalar processors, but rather a smaller number of independent processor cores with wide SIMD units [Fatahalian and Houston 2008; Lindholm et al. 2008]. The SIMD units must access memory coherently to enable high performance.

We therefore develop an algorithm that is more suited to actual GPU architecture. In the next sections, we will first describe a model for how modern GPUs operate, which will allow us to design several flavors of efficient compaction algorithms, in the sections following.

```

1 // Phase 1: Count Valid Elements
2 in parallel for each processor p
3   count ← 0
4   for (i ← 0; i < Kp; i ← i + 1)
5     if valid inputp[i]
6       count++
7   processorCounts[p] ← count
8
9 // Phase 2: Compute Offsets
10 processorOffsets[0..P) ← prefixSum processorCounts[0..P)
11
12 // Phase 3: Move Valid Elements
13 in parallel for each processor p
14   j = processorOffsets[p]
15   for (i ← 0; i < Kp; i ← i + 1)
16     e ← inputp[i]
17     if valid e
18       output[j] ← e
19       j ← j + 1

```

Listing 2: Basic parallel algorithm. The number of elements processed by a processor is denoted K_p , and input_p is the associated range of input elements. The notation $[0..P)$ is used to describe the range of elements from 0 (inclusive) to P (exclusive).

3.1 GPU model

A modern GPU consists of a number of processor cores, in the order of 10s, that contain a number of ALU's that execute instructions in a SIMD fashion. On current, and announced, GPU hardware, functional SIMD width vary between 16 [Seiler et al. 2008] and 64. E.g. on the NVIDIA GTX 280 each processor core has a SIMD width of 8, which executes the same instruction four times, yielding a functional SIMD width of 32 [Lindholm et al. 2008]. A similar arrangement on current AMD hardware results in 64 wide SIMD [Fatahalian and Houston 2008].

To hide memory latency, each processor core executes a large number of threads that are grouped according to SIMD width. The processor can switch between groups with little or no overhead, allowing another SIMD group to execute when a long latency memory operation is initiated.

The SIMD groups execute independently from each other, and have access to a small, fast, shared memory and are, by virtue of SIMD execution, internally synchronized. Each SIMD group can be viewed as a *concurrent read concurrent write PRAM (Parallel Random Access Machine)* with a fixed number of processors and arbitrary resolution of write collisions. There exists a large number of parallel algorithms developed for the PRAM model, for example prefix sum [Hillis and Steele 1986] and reduction [Blelloch 1990].

The memory interface is very wide and the access is largely uncached, as thread switching is used to hide latency. As a consequence of this, hardware attempts to gather memory accesses from

a SIMD group into *transactions*. For optimal performance, the start address of the transactions must be aligned.

To summarize, one can view the GPU as a machine with P *virtual* processor cores, each with a functional SIMD width of S . The number P is chosen to ensure all physical processors have sufficient threads for efficient latency hiding, but is otherwise kept as small as possible. Keeping P small maximizes the amount of sequential work each processor can perform independently. We assume the memory transaction size, and required alignment, to be a multiple of the functional SIMD width, S . This is, on current hardware, a common requirement for good performance when accessing memory.

Notation In the following sections, we will make frequent use of the subscript, and index, p to indicate a specific virtual processor in the range $[0..P)$. The symbol s is similarly used to refer to the SIMD lane in range $[0..S)$.

The notation $[A..B)$ is used to access a range of $B - A$ elements simultaneously, starting at, and including, the element at index A . An example is $\text{output}[A..B) \leftarrow \text{tmp}[A..B)$ where $B - A$ elements are copied. We make use subscript notation to indicate SIMD variables. SIMD variables are otherwise treated like vectors of length S .

3.2 Parallel SIMD stream compaction

Given the model detailed in the previous section, it is natural to let each virtual SIMD processor take the role of a processor in the basic parallel algorithm described in Listing 2. We shall refer to each virtual SIMD processor as a *processor* for the remainder of this paper.

If each processor is used to perform the work of a scalar processor, $(S - 1)P$ SIMD lanes are unused. To make use of these computational resources, and to improve memory coherency, an internal SIMD compaction step is performed: each processor will now process S elements each iteration.

Our new algorithm extends the basic algorithm from Listing 2 by utilizing SIMD capabilities during phases 1 and 3. The remaining parts of the algorithm are unchanged. Input data is, as before, divided into ranges input_p of size K_p .

In the first phase, each SIMD lane is used to count the valid elements it encounters, independently. After processing the entire range, the processor performs a parallel sum-reduction, which yields the total number of valid elements in the range input_p . Pseudo code describing these modifications is provided in Listing 3.

```

1 // Phase 1: Count Valid Elements
2 in parallel for each processor p
3   count[0..s) ← 0[0..s)
4   for (i ← 0; i < Kp; i ← i + S)
5     in parallel for each SIMD lane s
6       if valid inputp[i + s]
7         count[s] ++
8
9   processorCounts[p] ← reduce(+, count[0..s])

```

Listing 3: The first phase, extended to make use of the SIMD capabilities to count the number of valid elements. After reducing the count from S individual SIMD lanes, the resulting total is stored in the vector processorCounts . We use a parallel reduction to sum the values in $\text{counts}[0..s)$, as shown on line 9. However, this reduction is performed only once per processor, and is thus not time critical.

The second phase remains identical to the basic algorithm in Listing 2: a parallel prefix sum [Horn 2005; Sengupta et al. 2006] is used to compute offsets for the processors. The prefix sum is not a time-critical part of the algorithm, as it is performed over a small

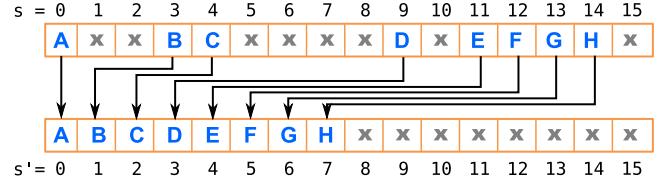


Figure 2: Illustration of the `compactSIMD` procedure. Here, the SIMD width is $S = 16$, and the number of valid elements is $Q = 8$. The element **A** moves from $s = 0$ to $s' = 0$, whereas the element **B** moves from $s = 3$ to $s' = 1$. The output is compact in the sense that the Q valid elements occupy lanes 0 through $Q - 1$.

number of elements. On current hardware, the number of processors P is several orders of magnitude lower than the number of input elements, N , commonly seen in GPU applications.

The final phase of the algorithm consists of moving the valid elements to the output vector. The pseudo code, shown in Listing 4, assumes the existence of a procedure `compactSIMD`. We present several variants of the `compactSIMD` procedure in the following sections. These procedures provide efficient compaction within a SIMD unit, as illustrated in Figure 2.

```

1 // Phase 3: Move Valid Elements
2 in parallel for each processor p
3   j ← processorOffsets[p]
4   for (i ← 0; i < Kp; i ← i + S)
5     a[0..s) ← inputp[i..i+S)
6     b[0..s), numValid ← compactSIMD a[0..s)
7     output[j..j+numValid) ← b[0..numValid)
8   j ← j + numValid

```

Listing 4: Each processor compacts its range, input_p , to the correct location in the global output vector, output , taking advantage of the SIMD processing capabilities.

The algorithm reads data with perfect coherency for each processor, which is a substantial improvement over the basic parallel implementation. Writes, however, while improved, are neither aligned nor whole transactions. Depending on the ratio of valid elements, the processor may write less than a full SIMD width of data each iteration. This can be improved, at the cost of a more complex algorithm, which uses buffering. Buffering is discussed in Section 3.5.

3.3 SIMD-Compaction with a prefix sum

This section describes an implementation of the `compactSIMD` procedure referred to in Section 3.2.

The goal is to move valid elements from a source SIMD lane s to a target SIMD lane s' in a fashion that groups valid elements in lanes $[0..Q)$, where $Q \leq S$ is the number of valid elements. This process is illustrated in Figure 2. After compaction, elements in lanes $[Q..S)$ are undefined.

```

1 procedure compactSIMD (a[0..s])
2   in parallel for each SIMD lane s
3     validFlags[s] ← valid a[s]
4
5   index[0..s), numValid ← prefixSum validFlags[0..s]
6
7   in parallel for each SIMD lane s
8     if validFlags[s]
9       s' ← index[s]
10      result[s'] ← a[s]
11
12 return (result[0..s), numValid)

```

Listing 5: Implementation of `compactSIMD` using a prefix sum. The procedure returns the number of valid elements, numValid , and r , which contains the valid elements in lanes 0 through $\text{numValid} - 1$.

An efficient SIMD prefix sum [Hillis and Steele 1986] is used to find the target lane index s' in $\log S$ steps. The `compactSIMD` procedure is summarized in Listing 5.

Note that we use the SIMD prefix sum described by Hillis and Steele [1986], which is not work-efficient [Sengupta et al. 2006]. A work-efficient implementation uses $2 \cdot \log S$ steps, whereas the prefix sum used here performs $\log S$ steps. This is preferable, since nothing is gained by letting SIMD lanes idle.

3.4 SIMD-Compaction with POPC

The second `compactSIMD` implementation uses a population count operation (`POPC`; count number of set bits) to find the target SIMD lane index for each element.

Assuming `POPC` is present in the native SIMD instruction set, the $\log S$ steps required by the SIMD prefix sum can be avoided. The architecture must also support a word size of S bits. Listing 6 shows the new `compactSIMD` procedure.

```

1 procedure compactSIMD (a[0..s])
2     m ← 0
3     in parallel for each SIMD lane s
4         if valid a[s]
5             m ← m | (1 << s)
6
7     in parallel for each SIMD lane s
8         if valid a[s]
9             m' ← m & ((1 << s) - 1)
10            s' ← POPC m'
11            result[s'] ← a[s]
12
13    numValid ← POPC m
14
15    return (result[0..s], numValid)
```

Listing 6: Implementation of `compactSIMD` using a population count. The variable `m` must be large enough to store S bits. Setting `m`, on line 5, assumes that the architecture allows simultaneous setting of bits. This is not always the case, and a workaround is described in Section 4.2.

The number, s' , of valid elements in front of the current element is found by masking the corresponding bits in `m`. Then, applying `POPC` gives the desired offset. This is shown on lines 9 and 10 of Listing 6.

Building the bit mask, as shown on line 5 in Listing 6, is not permitted by our model, because of current GPU hardware limitations. However, a SIMD instruction set will often allow the creation of a mask register that can be broadcast to all SIMD lanes. This problem is discussed further in Section 4.2, in context of our CUDA implementation.

3.5 Buffering

As noted in Section 3.2, the algorithm achieves perfect coherence when reading memory. Writes are always consecutive, but to maximize bandwidth they must also be aligned, and of the optimal transaction size, i.e. a multiple of the SIMD width S (discussed in Section 3.1).

To achieve alignment and full write transactions, the algorithm can make use of fast on-chip storage to buffer elements. The on-chip storage is usually very small, so the algorithm should buffer a minimal number of elements.

The buffering only affects the last phase of the algorithm. In Listing 7, the third phase is modified to buffer elements. Elements will be added to the buffer until S elements can be written to the output.

The strategy is to buffer S elements: this is a convenient number, since a single iteration can produce at most S valid elements. The buffer is flushed to the output vector only when it is full, guaranteeing complete write transactions. To ensure proper alignment, the algorithm must first move enough valid elements to reach the required alignment boundary.

As the alignment phase will write at most $S - 1$ elements, these writes will not contribute significantly to the run time of the algorithm and can therefore be unbuffered, as long as the buffer is correctly initialized.

```

1 // Phase 3: Move Valid Elements will full buffering
2 in parallel for each processor p
3     j ← processorOffsets[p]
4     i, #buffered, buffer ← alignOutput()
5     for (; i < Kp; i ← i + S)
6         a[0..s) ← input[i..i+S)
7         d[0..s), numValid ← prefixSum valid a[0..s)
8         in parallel for each SIMD lane s
9             if valid a[s] && d[s] + #buffered < S
10                buffer[d[s] + #buffered] ← a[s]
11                if numValid + #buffered > S
12                    output[j + s] ← buffer[s]
13                    j ← j + S
14                if valid a[s] && d[s] + #buffered >= S
15                    buffer[d[s] + #buffered - S] ← a[s]
16                    #buffered ← (#buffered + numValid) % S
17         in parallel for each SIMD lane s
18             if s < #buffered
19                 output[j + s] ← buffer[s]
```

Listing 7: The buffered implementation of the third phase. The buffer is flushed when it is full, and the overflowing elements are then written to the buffer. The `alignOutput` procedure moves enough elements to align `j` to the next multiple of S , and initializes the buffer and counters. There are many short branches in the inner loop, however, they can be compiled into efficient predicated instructions.

3.6 Analysis

Our algorithm avoids computation of a prefix sum of N elements, when compared to previous approaches. This *global* prefix sum is replaced by an efficient counting pass and a prefix sum of only P elements. As a result, the bandwidth usage is substantially reduced, e.g. nearly halved for 32 bit data.

Auxiliary storage requirements are only in the order of $O(P)$ elements. Note that, like previous algorithms, compaction is not performed in place.

In the previous work, time complexity is usually not given with the number of processors P as a factor. One reason is that earlier programming models, e.g. through graphics APIs, made it difficult or impossible to perform this analysis. Today, with a more direct control over execution on the hardware, it makes sense to use this information in the analysis to guide our design of more efficient algorithms.

To summarize, our algorithm consists of three distinct phases. The time complexity of each phase is simple to estimate:

$$\text{Phase 1: } O\left(\frac{N}{PS} + \log S\right)$$

$$\text{Phase 2: } O(\log P)$$

$$\text{Phase 3: } O\left(\frac{N}{PS} \cdot \log S\right)$$

Thus, for all three phases, the overall time complexity becomes

$$O\left(\frac{N}{PS} \cdot \log S + \log P\right) \quad (1)$$

Asymptotic behavior is hence proportional to $O(N)$ when $N \gg P > S$.

The complexity presented in equation 1 indicates that using wide SIMD (large S) is disadvantageous. Indeed, if the algorithm is run independently on each SIMD lane, as done in Listing 2, the time complexity is reduced to $O\left(\frac{N}{P'} + \log P'\right)$, where $P' = PS$. However, this introduces a memory access pattern that is generally very inefficient on current hardware.

It is possible to perform further trade-offs between computational load and better memory access patterns by using buffering techniques.

4 CUDA Implementation

We have implemented our algorithm, and its variations, using CUDA [NVIDIA 2008]. Our development hardware is an NVIDIA GTX 200 series GPU.

4.1 CUDA Introduction and Specifics

CUDA is described by NVIDIA as a *SIMT, Single Instruction Multiple Thread*, programming model. This model lets the programmer write a scalar program that executes, seemingly independently, on a SIMD lane. The hardware, however, actually executes the threads in SIMD fashion.

On the G80 and later NVIDIA architectures, the functional SIMD width S is 32. Each group of 32 threads is referred to as a *warp*. The warps are further grouped into *blocks*, of which a number can execute simultaneously on a processor core. We only make use of the warp level which maps to the role of a processor in our model. We do not use the block level, since this would require explicit synchronization.

While it is possible to choose S smaller than 32, which would reduce the computational workload according to equation 1, this results in a worse memory access pattern.

Our algorithms assume that it is possible to shuffle data between SIMD lanes. In CUDA, this is achieved by the use of *shared memory*, a fast memory area that is shared between all threads in one block.

When choosing an appropriate value of P , one has to take several factors into consideration: while it is advantageous to choose a small P , in order to maximize the sequential work performed by each processor, there must be enough jobs available for latency hiding to work.

For full occupancy, i.e. full use of hardware resources, a thread block must have at least 128 threads (four warps) [NVIDIA 2008]. An NVIDIA GTX280 GPU has 30 processor cores, and therefore we should start at least as many blocks.

In order to determine good values, we measured performance using different configurations, and arrived empirically at the following values: we use 120 thread blocks with four warps each. This configuration results in $P = 480$ virtual processors, which offered best performance in all our tests.

The current implementation assumes that N is a multiple of S for simplicity. If $\frac{N}{S}$ is not evenly divisible by P , our algorithm will adjust the amount of data assigned to each processor.

4.2 Algorithm Implementation

Implementation of phase 1 follows closely the pseudo code given in Listing 3. Phase 2 can be implemented by applying an existing prefix sum implementation, such as in CUDPP [CUDPP 2008].

Phase three of our algorithm uses the `compactSIMD` procedure to compact elements to consecutive SIMD lanes. In CUDA, we have to perform this compaction into shared memory, and then flush the shared memory to our global output vector. We refer to this version of the algorithm as *staged*, since output is assembled in shared memory prior to flushing.

A second option, as described in Section 3.5, is to fully buffer and align the output. This results in whole transactions, also known as *coalesced writes* in CUDA terminology, each time the shared memory buffer is flushed.

A third variation available in CUDA is to directly scatter valid elements into the global output vector, bypassing the staging or buffering step in shared memory.

One final alternative dynamically chooses between the staged and scattered variants on a processor (warp) level based on a heuristic involving the per-range ratio of valid elements. In Section 5, we shall show that this *selective* variant performs best on our target hardware.

Prefix sum based `compactSIMD` Implementation of the prefix sum based algorithm poses no new challenges. We closely follow the pseudo code presented in Section 3. All four output alternatives presented in this section were tested, as the prefix sum based implementation promised best performance on our target hardware. Additionally, we implemented optimizations presented in Section 4.3.

POPC based `compactSIMD` Implementing the `POPC` based approach is more challenging, since CUDA does not allow us to quickly build a bit-mask for all SIMD lanes. To build such a mask, and to maintain a good access pattern, we work on $S \times S$ blocks of elements.

For each $S \times S$ block, we construct an $S \times S$ bit-matrix in shared memory, during the first phase. This matrix is transposed in S steps using bit-wise operations, yielding S words each containing flags for S consecutive elements. The first phase stores these words to global memory; in the third phase we broadcast one word at a time to all lanes. Each lane applies a mask and uses the `POPC` to find the number of preceding elements.

However, the `POPC` operation is not a native instruction in the GTX 280 architecture. Instead the operation compiles to a binary bitwise reduction, which performs $\log S$ operations. Thus, it does not, in CUDA, save any computation over the prefix sum version presented in Section 3.3.

4.3 Optimizations and Problems

We have encountered some problems that are related to the memory access pattern. Also, bandwidth can be increased by using different transaction sizes.

Our tests show that it is often advantageous to load elements as 32×64 bits, rather than the more obvious 32×32 bits for 32 bit data. Thus, during each iteration, we load and handle two 32 bit elements, instead of one. Table 1 presents measured bandwidths using different transaction sizes.

Table 1: Measured pass-through memory bandwidth in CUDA, using different transaction sizes. The pass-through kernel divides data into the same ranges input_p as our stream compaction implementation. The input data is then simply copied from the input buffer to the output buffer. Measured on an NVIDIA GTX280 GPU.

Size (32×)	32 bit	64 bit	128 bit
Bandwidth (GB/s)	77.8	102.5	73.4

Using 64 bit accesses requires 64 bit alignment. Since writing data in the third phase may be misaligned (odd number of preceding elements), elements are written using 32 bit writes. Using a fully buffered algorithm enables the use of 64 bit writes. This optimization is also specific to 32 bit (or smaller) data.

The division of data into ranges that each processor can handle independently sometimes conspires to give a suboptimal memory access pattern.

We presented average bandwidth in Table 1 for a pass-through kernel. For certain combinations of P and N , bandwidth drops by almost one order of magnitude. For instance, one such region exists for $P = 480$ and $N = 24M$ 32 bit elements. The region with reduced bandwidth is relatively narrow, as shown in Table 2.

We believe that this is a hardware related issue, but have insufficient information about hardware memory operation to be certain. A simple workaround that we have implemented is to sequentially apply the stream compaction to smaller amounts of data. Since the first bandwidth drop for $P = 480$ is observed at approximately $N = 11M$, we will simply subdivide the data into chunks of 10M, and compact these sequentially. In Section 5, we show that any performance penalty incurred by this workaround is minimal.

Table 2: Pass-through bandwidth for a small region around $N = 24M$ 32 bit elements. At $N = 24M$, the bandwidth falls to one fourth of the expected bandwidth, but for larger N , it quickly returns to the expected values. Measured on an NVIDIA GTX280 GPU.

$N =$	24M – 100k	24M	24M + 100k
Bandwidth (GB/s)	105.4	24.0	106.0

5 Results

We have compared our implementations with existing stream compaction solutions, e.g. using geometry shaders with transform feedback and the freely available CUDPP library [CUDPP 2008]. We also present data from previously published stream compaction algorithms.

Timings and tests were performed on random data, with a specific ratio of valid elements. For verification purposes, both uniform (white) and frequency filtered brownian noise was used. Only results using uniform noise are presented, as we observed no significant differences in performance for other kinds of noise.

All results were measured on a Intel Core2 Quad Q9300 at 2.5GHz with a NVIDIA GTX280 graphics card with 1GB of memory. We used the CUDA 2.1 release.

5.1 Performance

In Figure 3, we show the performance of our implementations for an increasing proportion of valid 32 bit elements. Our prefix sum based implementations all use the optimized 64 bit loads, as described in Section 4.3.

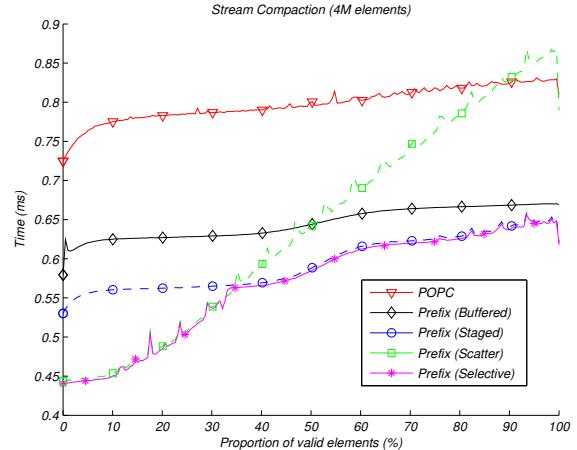


Figure 3: Time (in milliseconds) required to compact 4M (2^{22}) 32 bit elements with a varying ratio of valid elements. We have compared our implementations, using both the prefix sum and the POPC based compactSIMD implementations, as well as staged, scattered and buffered variations. In addition, our selective implementation, which automatically switches between staged and scattered output on a warp level, is shown.

The graph shows that our *selective* implementation, which dynamically chooses between staged and scattered operation based on the result of the previous phases, is the fastest for all densities. This is the implementation we will use in further performance comparisons.

The buffered version is, despite a better memory access pattern, slower for all densities. Performance is almost constant, which indicates that the algorithm is becoming computational bound. This is expected to change on future hardware, as discussed in Section 6.

The scattered version shows a near-linear performance scaling. Since we cannot use 64 bit writes, but must resort to several scattered 32 bit writes, the number of write transactions quickly increases.

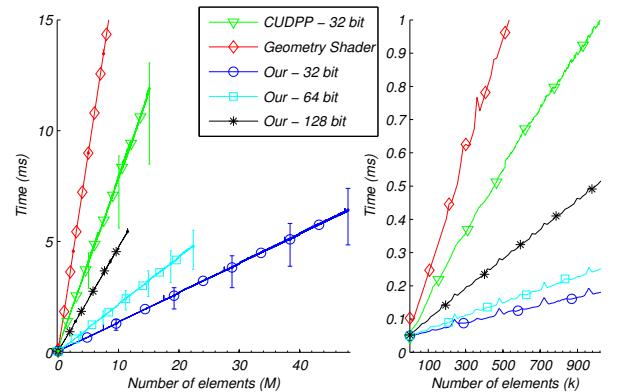


Figure 4: Time (in milliseconds) required to compact a varying number of elements. We compare our best implementation with the CUDPP implementation and geometry shaders. The geometry shader plot is cut off to provide a better view of CUDPP and our implementation. The error bars in the left figure display variations in time as the proportion of valid elements is changed. The graphs represent the average time for varying proportions of valid elements. Also shown are curves for compaction of 64 bit and 128 bit elements.

Figure 4 shows how the performance compares to other algorithms for increasing numbers of elements. A clear linear relation between

performance and number of elements can be observed, as previously indicated in Section 3.6. No observable discontinuities exist at multiples of 10M elements, showing that the penalties from the workaround described in Section 4.3 are indeed minimal.

Our implementation outperforms CUDPP roughly by a factor 3, and the geometry shader based algorithm by an order of magnitude. Our algorithm uses much less memory, which can be seen in Figure 4: under equivalent conditions, our algorithm runs out of memory much later. Times reported for CUDPP include an additional pass that creates an array of flags indicating valid elements, which is required by the CUDPP API. This pass takes approximately 0.31 ms; if excluded from the measurements, our implementation outperforms CUDPP by a factor of 2.65.

Also observable in Figure 4 is that our algorithm can compact the same amount of 128 bit elements faster than competing algorithms can compact 32 bit elements.

Further comparisons to other earlier algorithms are summarized in Table 3. Our algorithm performs approximately three times faster than any previously reported. Since all the algorithms have linear asymptotic behavior, this observation can be expected to hold for larger N on future hardware.

Table 3: Comparison of compaction performance with competing techniques. If available, we have used reference implementations for measurements on our hardware. We have created our own CUDA implementation of the algorithm presented in Ziegler et al. The reported times are averages over uniform distributions with 0% to 100% valid elements.

		Time (4M, 32 bit)	Time (relative)
Our	GTX280 min / max	0.561 ms 0.435 ms / 0.648 ms	1.0×
CUDPP	GTX280 min / max	1.81 ms 1.22 ms / 1.95 ms	3.22×
Ziegler et al. [2006]	GTX280 min / max	2.54 ms 0.539 ms / 4.04 ms	4.53×
Geometry Shaders	GTX280 min / max	7.05 ms 7.03 ms / 7.09 ms	12.6×
Roger et al. [2007a]	8800 GTS min / max	10.6 ms 9.09 ms / 11.4 ms	18.9×

Results from some earlier works [Horn 2005; Sengupta et al. 2006] are not included in Table 3. This is because the publicly available CUDPP implementation uses the same basic strategy, and offers higher performance.

5.2 Global Prefix Sum

In order to find the correct output offsets for the valid elements, our algorithm computes, albeit temporarily, what amounts to a global prefix sum over the validity of elements. It is therefore simple to modify the algorithm to compute a global prefix sum of the elements.

Using the variant that utilizes a SIMD prefix sum, as described in Section 3.3, we need to change phase 1 to sum the input values, as opposed to counting valid elements. Similarly, phase 3 must be modified to perform a prefix sum on the values, and add the offset from phase 2. Phase 2, however, remains unchanged.

Since the algorithm always reads and writes S elements, it achieves perfect alignment and coherence for both reads and writes. In Table 4, we compare the performance to some recent parallel prefix sum implementations [Dotsenko et al. 2008; Sengupta et al. 2007]. An optimized version of the algorithm in Sengupta et al. [2007] is implemented in CUDPP [CUDPP 2008].

Table 4: Required time to compute a prefix sum of 32M elements. We compare our implementation to two different algorithms. Measurements performed by Dotsenko et al. used older hardware, but also compare performance to CUDPP. We have included prefix sum performance reported by Dotsenko et al. for CUDPP.

		Time (32M, 32 bit)
Our	GTX280	3.7 ms
CUDPP	GTX280	5.3 ms
	8800 GTX	11.50 ms
Dotsenko et al. [2008]	8800 GTX	8.5 ms

If desired, our algorithm can compute the prefix sum in-place. E.g. a prefix sum over 220M elements (or 880M byte of data) takes 25.3 ms to compute using our algorithm.

5.3 Stream Split and Radix Sort

Another commonly used primitive is *stream split*, by which a stream is partitioned into two compact ranges. This is useful, for example, when constructing binary trees. The stream split is simply two compactions, where the second has the negated validity.

Modifying our stream compaction algorithm to perform this operation is simple. All required information is already computed in phases 1 and 2. The third phase must be modified to output all the invalid elements in the same way as the valid ones at the end of the range.

To test the efficiency of the split operation, we implemented a radix sort and compared it to the fastest state of the art implementation [Satish et al. 2008], provided in the CUDA 2.1 SDK. The results are presented in Table 5.

Table 5: Comparison of our stream split based radix-sort and the currently fastest published implementation. Our implementation shows almost identical performance, but is more flexible. Our implementation operates on interleaved key-value pairs; Satish et al. have separate arrays for keys and values. We can handle separate keys and values by a pre- and postprocessing step that transforms the separate arrays into interleaved data and back.

Input Data (4M elements)	Satish et al.	Our	CUDPP
32 bit keys only	27.6 ms	28.2 ms	50.1 ms
32 bit key, 32 bit value interleaved separate	-	35.4 ms 38.5 ms	- -
32 bit key, 96 bit value interleaved	-	61.7 ms	-

Our performance is comparable with that of Satish et al. [2008], and outperforms the CUDPP library [CUDPP 2008]. The implementation is very simple, we did not perform any in-depth analysis or special optimization. We simply invoke the stream split operation once for each bit in the radix sort key. The simplicity makes it very flexible, allowing any data type and number of bits as radix keys.

6 Discussion

It is commonly assumed that, on future hardware, the compute-to-bandwidth ratio will increase. To see how our algorithms might perform if this is the case, we lowered the memory clock on our test GTX 280. We tested our algorithms with a compute-to-bandwidth ratio that is twice that of a GTX 280. In this scenario, the implementation that employs buffering using shared memory, described

in Section 3.5, outperforms the scattering and staging variants at high proportions of valid elements.

A more conventional cache hierarchy, with e.g. write-combining caches, on future hardware may favor our more simplistic implementations. Our manual buffering techniques, which incur some computational overhead, would be made superfluous.

Our new algorithm should map well to the upcoming Intel Larrabee. This architecture sports a native instruction `vcompress` [Abrash 2009], which is similar to our `compactSIMD` procedure.

7 Conclusion

We have presented a new algorithm, with several variations, for efficient stream compaction on the GPU; all variations perform, to the best of our knowledge, better than any previously published work.

Since stream compaction is a commonly used primitive, applying our algorithm should improve the performance in many existing applications, e.g. tree traversal algorithms [Lauterbach et al. 2009], GPU raytracing [Roger et al. 2007b] and algorithms utilising sorting [Satish et al. 2008].

The algorithms make minimal demands on the capabilities of the hardware, and should thus be implementable efficiently on most multi-core SIMD processors, including, for example, the upcoming Intel Larrabee GPU [Seiler et al. 2008] or AMD graphics hardware.

Since all global communication is limited to a single cheap pass, this algorithm should scale well with an increasing number of independent processor cores.

The algorithm also illustrates a successful general strategy for minimizing synchronization and maximizing the use of independent SIMD processors. Different algorithms can be formulated by dividing the work into independent chunks and then combining the results. This is illustrated by our implementation of a high performance stream split and radix sort.

Source code of our reference implementations will be made available online.

Acknowledgments

We would like to thank Erik Sintorn and the anonymous reviewers for their valuable comments.

References

- ABRASH, M., A First Look at the Larrabee New Instructions (LRBn), 2009.
<http://www.ddj.com/hpc-high-performance-computing/216402188>.
- BLELLOCH, G. E. 1990. Prefix Sums and Their Applications. Tech. rep., Synthesis of Parallel Algorithms.
- CHATTERJEE, S., BLELLOCH, G. E., AND ZAGHA, M. 1990. Scan Primitives for Vector Computers. In *In Proceedings Supercomputing '90*, 666–675.
- CUDPP: CUDA data parallel primitives library, 2008.
<http://www.gpgpu.org/developer/cudpp/>.
- DOTSENKO, Y., GOVINDARAJU, N. K., SLOAN, P.-P., BOYD, C., AND MANFERDELLI, J. 2008. Fast scan algorithms on graphics processors. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, ACM, New York, NY, USA, 205–213.

FATAHALIAN, K., AND HOUSTON, M. 2008. A closer look at GPUs. *Commun. ACM* 51, 10, 50–57.

GRESS, A., GUTHE, M., AND KLEIN, R. 2006. GPU-based Collision Detection for Deformable Parameterized Surfaces. *Computer Graphics Forum* 25, 3 (Sept.), 497–506.

HILLIS, W. D., AND STEELE, JR., G. L. 1986. Data parallel algorithms. *Commun. ACM* 29, 12, 1170–1183.

HORN, D. 2005. Stream reduction operations for GPGPU applications.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast BVH Construction on GPUs. In *Proceedings of the Eurographics Symposium on Rendering*, the Eurographics Association, Eurographics and ACM/SIGGRAPH.

LINDHOLM, E., NICKOLLS, J., OBERMAN, S., AND MONTRYM, J. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* 28, 2, 39–55.

NVIDIA, CUDA Zone: Toolkit & SDK, 2008.
<http://developer.nvidia.com/object/cuda.html>.

ROGER, D., ASSARSSON, U., AND HOLZSCHUCH, N. 2007. Efficient Stream Reduction on the GPU. In *Workshop on General Purpose Processing on Graphics Processing Units*, D. Kaeli and M. Leeser, Eds.

ROGER, D., ASSARSSON, U., AND HOLZSCHUCH, N. 2007. Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU. In *Rendering Techniques 2007 (Proceedings of the Eurographics Symposium on Rendering)*, the Eurographics Association, J. Kautz and S. Pattanaik, Eds., Eurographics and ACM/SIGGRAPH, 99–110.

SATISH, N., HARRIS, M., AND GARLAND, M. 2008. Designing Efficient Sorting Algorithms for Manycore GPUs. NVIDIA Technical Report NVR-2008-001, NVIDIA Corporation, Sept.

SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. 2008. Larrabee: a many-core x86 architecture for visual computing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, ACM, New York, NY, USA, 1–15.

SENGUPTA, S., LEFOHN, A. E., AND OWENS, J. D. 2006. A Work-Efficient Step-Efficient Prefix Sum Algorithm. In *Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures*, D–26–27.

SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. 2007. Scan Primitives for GPU Computing. In *Graphics Hardware 2007*, ACM, 97–106.

WALD, I., GRIBBLE, C. P., BOULOS, S., AND KENSLER, A. 2007. SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering. Tech. Rep. UUSCI-2007-012.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time KD-tree construction on graphics hardware. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, ACM, New York, NY, USA, 1–11.

ZIEGLER, G., TEVS, A., THEOBALT, C., AND SEIDEL, H.-P. 2006. GPU Point List Generation through Histogram Pyramids. Technical Reports of the MPI for Informatics MPI-I-2006-4-002, June.

Parallel Processing: A Smart Compiler and a Dumb Machine

Joseph A. Fisher, John R. Ellis,
John C. Ruttenberg, and Alexandru Nicolau

Department of Computer Science, Yale University
New Haven, CT 06520

Abstract

Multiprocessors and vector machines, the only successful parallel architectures, have coarse-grained parallelism that is hard for compilers to take advantage of. We've developed a new fine-grained parallel architecture and a compiler that together offer order-of-magnitude speedups for ordinary scientific code.

Introduction

Compilers have traditionally played second fiddle to hardware projects in parallel processing. Parallel architectures have been built to be hand coded, and attempts at compiler writing were mere afterthoughts. These attempts have been unsurprisingly unsuccessful.

The two most common types of parallel architectures built to date have been vector machines and multiprocessors. Compiling (or simply hand coding) for either requires matching an overview of the coarse structure of the application to that of the hardware. It's conceivable that hand coders and compilers might someday be good at this; but so far they haven't been, and there's no reason for optimism. There has been a general failure at culling large amounts of parallelism from ordinary applications.

So instead of building an architecture first and a compiler second, we have simultaneously developed a compiler and an architecture intended for scientific computing. Using a technique called **trace scheduling**, the Bulldog compiler finds large amounts of parallelism in ordinary scientific code. Taking advantage of this parallelism requires a new architecture, which we call VLIW (Very Long Instruction Word).

The Bulldog compiler is finished, and it compiles ordinary scientific programs into highly parallel machine code for a large class of VLIWs, achieving order-of-magnitude speedups over traditional architectures. We think VLIW architectures are practical in the very near

future, and we're building a VLIW machine, the ELI (Enormously Long Instructions) to prove it.

In this paper we'll describe some of the compilation techniques used by the Bulldog compiler. The ELI project and the details of Bulldog are described elsewhere [4, 6, 7, 15, 17].

VLIW Architectures

Highly parallel machines that actually have been built fall into two broad classes: multiprocessors and vector machines. Both classes provide coarse-grained parallelism which is hard for a compiler to use.

With multiprocessors, a compiler must minimize communication and synchronization while trying to keep all the processors busy, avoiding the delays when one processor must wait for another. This forces a compiler to look for large sections of relatively independent control and data; compilers have only been able to do this for programs consisting of simple data-independent inner loops.

With vector machines, a compiler must find large aggregates in the program that can be fetched and operated upon simultaneously using relatively simple operators. This requires finding a high degree of regularity in the data and control, and compilers haven't been able to do that either for very many programs.

Instead of coarse-grained parallelism inaccessible to a compiler, VLIWs provide fine-grained parallelism that a trace-scheduling compiler can easily use. In a VLIW machine, every resource is completely and independently controlled, by which we mean:

Timing control. Every single action takes an amount of time predictable by the compiler. The time may vary according to the operation.

Flow control. There is a single thread of control, a single instruction stream, that initiates each fine-grained operation; many such operations can be initiated each cycle.

Communications control. All communications are completely choreographed by the compiler and are under explicit control of the compiled program. The source, destination, resources, and time of a data transfer are all known to the compiler. There is no sense of packets containing

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

destination addresses or of hardware scheduling of transfers.

Such fine-grained control of a highly parallel machine requires very large instructions, hence the name Very Long Instruction Word architecture.

Figure 1 shows a picture of a hypothetical VLIW machine. It has 16 clusters connected by simple data buses. Each cluster is a reduced instruction set processor that has local registers, instruction memory, optional data memory, a few functional units implementing integer and/or floating scalar operations, and a partial crossbar connecting these elements within the cluster.

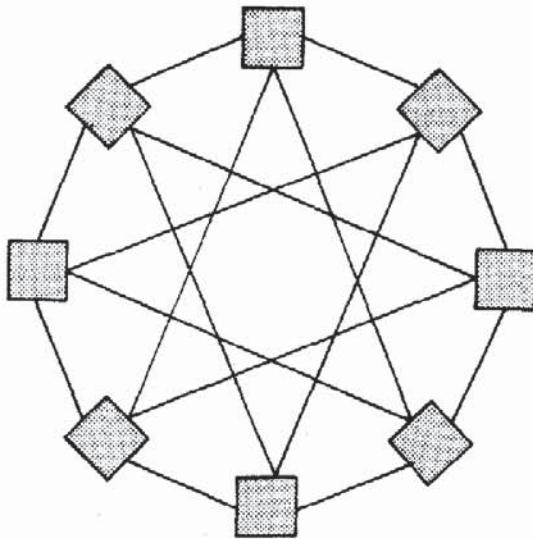


Figure 1: A hypothetical VLIW; each box is a separate cluster

All the clusters run in lockstep and are controlled by a single instruction stream. An instruction specifies the action of every element of every cluster independently; for example, one instruction may initiate a floating add in cluster 1, a floating subtract in cluster 2, an integer multiply in cluster 3, a register transfer between clusters 1 and 4, etc. Consequently, instructions will be very large (at least several hundred bits).

VLIW machines are far too large to have a single crossbar connecting all their elements. Instead, the clusters are connected by buses for transferring scalar values. It may well take several hops to move a value between distant clusters.

VLIWs need not have the regularity implied by the picture. The interconnections between the clusters, the type and number of elements within the clusters, and the connections between cluster elements can (and probably will) be asymmetric.

Before the advent of trace scheduling, it wasn't practical to build VLIW machines because no mere mortal could program them by hand. It is just barely possible to program horizontally microcoded machines and wide processors such as the FPS-164 and the MARS-432, but the amount of effort involved is tremendous. Programming a VLIW with 16 or more times the number of functional units is out of the question. Without a compiler for a high-level language, VLIWs would be useless.

Compilers for VLIWs

At first blush compiling high level languages for VLIWs might appear to be an impossible task, given that they are programmed at such a low, detailed level. But in fact the Bulldog compiler isn't that much different from a traditional optimizing compiler.

A traditional compiler parses the source program into an intermediate code, optimizes that intermediate code, and then translates the intermediate code into machine code. Usually, the translation to machine code is done one basic block at a time, perhaps after registers have been globally allocated.

It wouldn't be hard to construct a basic-block code generator for VLIWs. Several such code generators were written for machines with limited fine-grain parallelism such as the FPS-164, the CDC machines, and the scalar portion of the Cray [18]. Part of the problem is equivalent to that of statically scheduling a set of interdependent jobs with different resource requirements on a fixed set of processors; this problem has been studied for years and there are many practical solutions [5].

But basic blocks have severely limited parallelism; experiments showed early on that one could expect at most a two- or three-times speedup by executing basic blocks in parallel [9, 19]. A basic block-based code generator couldn't hope to keep a VLIW with 16 or 32 processors busy. So no one ever built a VLIW.

Later experiments [14] showed, however, that if one ignored the artificial constraints imposed by basic blocks, ordinary scientific programs contained large amounts of parallelism—factors of 90 on average. If only a compiler could find it, such parallelism is more than enough to keep a VLIW busy.

Trace Scheduling

Trace scheduling finds much of that factor-of-90 parallelism by giving more than one basic block at a time to the code generator. To generate machine code, the compiler repeatedly traces out a path of many basic blocks in the intermediate-code flow graph and hands that entire path to the code generator. These paths, or **traces**, contain much more parallelism than basic blocks. The code generator treats the trace of blocks almost as if it were a single, very large basic block.

The compiler picks a trace, generates code for it, picks another trace, generates code for it, and so on until the entire flow graph has been translated to machine code. Estimates of execution frequency guide the compiler in picking traces; the blocks most likely to be executed comprise the first trace, those next likely to be executed comprise the second trace, and so on. Figure 2 shows a simple program and the traces selected from it.

The current compiler uses loop nesting and programmer-supplied hints to make reasonable guesses about block execution frequency; this method appears to work fairly well without too much help from the programmer. One could easily imagine an automatic profiler that would supply execution counts based on sample runs of the program, though it's doubtful that it would do much better than the current method of guessing.

For various reasons, a trace never extends past a loop boundary. That is, a trace can include only blocks from

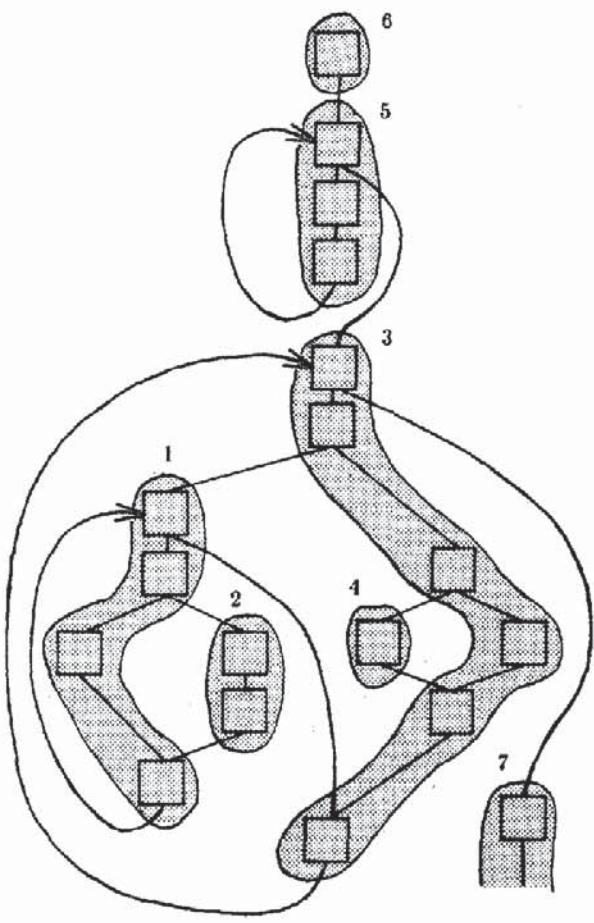


Figure 2: A flow graph with the traces selected from it

the same loop, but no blocks from containing or contained loops.

To further increase the parallelism of traces, the compiler unrolls the bodies of inner loops as many as 32 times immediately after parsing the source program into intermediate code. For example, a loop such as:

```
i := 1
LOOP {
    IF i > n THEN EXIT
    body
    i := i + 1
}
```

unrolled three times would look like:

```
i := 1
LOOP {
    IF i > n THEN EXIT
    body
    i := i + 1

    IF i > n THEN EXIT
    body
    i := i + 1

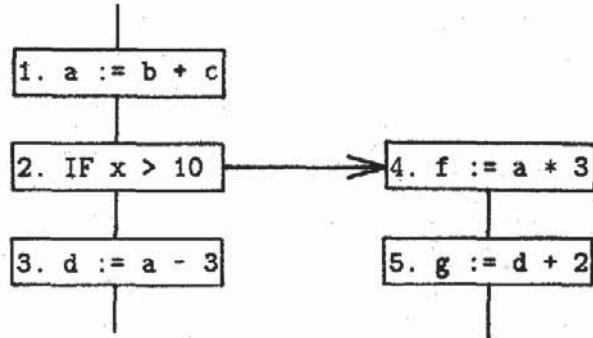
    IF i > n THEN EXIT
    body
    i := i + 1
}
```

This unrolling produces much longer traces, increasing the potential parallelism available to the code generator. (Later we'll see other uses for unrolling.)

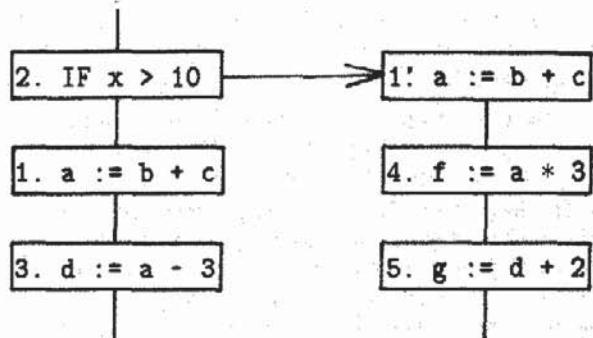
To get parallel code the code generator must substantially reorder the trace's intermediate-code operations, filling machine instructions with operations that come from widely separated places in the program; time-critical operations are usually scheduled early, while non-critical operations are often delayed. In a basic-block code generator of a traditional compiler, this reordering is relatively easy [1, 18].

By doing one basic block at a time, a traditional code generator is assured that all jumps into the block from the outside are to the block's first instruction, and that there is at most one conditional jump in the block, which must be at the end. But looking at figure 2, one immediately notices that traces consisting of many blocks will have more than one conditional jump and that there will be jumps from outside the trace into the middle of the trace. This complicates the task of reordering considerably; in addition to the normal data-precedence rules for basic block operations, the compiler must also worry about jumps off the trace and jumps into the trace.

Let's first consider reordering in the presence of conditional jumps. Suppose that we have the following fragment of a flow graph:



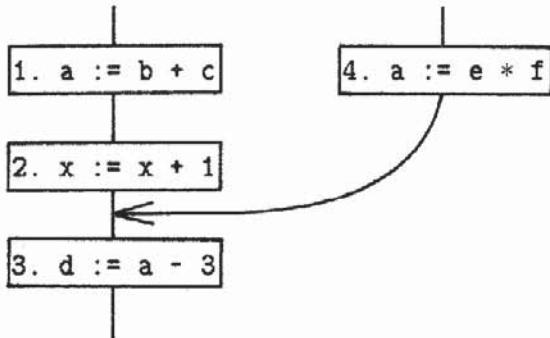
and that the current trace consists of operations 1, 2, and 3. Suppose that the code generator decides that operation 1 is not time-critical and should be moved below the conditional jump 2. If it moves 1 below 2, then operation 4, which reads the variable *a* written by 1, will get the wrong value of *a*. So if 1 is moved below 2, the compiler will have to make a copy of 1, 1', on the off-trace edge of the jump:



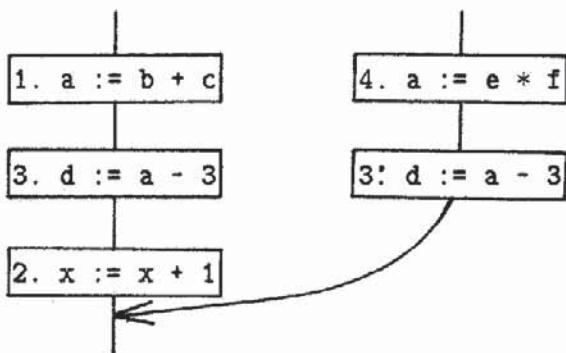
Conversely, suppose that the code generator decides that 3 is time-critical and would like to move it before the jump. Because 3 writes the variable *d*, and 5 reads the previous value of *d*, moving 3 above the jump would be

incorrect, since 3 would then get the wrong value of **d**. If the value of **d** were not used on the off-trace edge of the jump, then moving 3 above the jump would be permissible.

What about jumps from blocks outside the trace into the middle of the trace? For example, assume that in the following fragment the current trace consists of operations 1, 2, and 3:



Suppose 3 was time critical and the code generator wanted to move it before 2, above the spot where 4 jumps to the trace. By itself, this motion is incorrect, because 4 writes variable **a** and 3 reads it; 3 would no longer get the correct value of **a** from 4. The solution is to make a copy of 3, 3', on the incoming edge right below 4; in this way, no matter which path is executed, **d** will still get the same value.



The general rules for code motions relative to jumps and rejoins within a trace are:

If a trace operation moves below a conditional jump, a copy of it must be placed on the off-trace edge of the jump.

A trace operation that writes a variable can't move above a conditional jump if the variable is live on the off-trace edge of the jump.

If a trace operation moves above a rejoin to the trace, then a copy of it must be placed on the off-trace rejoining edge.

In these examples we've only considered simple operations moving past jumps and rejoins to the trace, but conditional jumps as well may move past other jumps and rejoins. The same rules apply, though there are some minor complications in copying conditional jumps.

After generating machine code for a trace, the copies of intermediate code operations resulting from the code motions are inserted into the flow graph. They will be

selected and compiled as part of later traces. One might think that excessive code motion would cause an explosion in copied operations, resulting in a very large object program, or perhaps that trace scheduling doesn't even terminate. In fact, it does terminate [15], and experiments show that the amount of copying is quite acceptable [4].

Memory Reference Disambiguation

Indirect memory references arising from pointer dereferencing and array indexing pose special problems for a trace-scheduling compiler. Long traces contain many such indirect references, and in order to take advantage of the potential parallelism in the trace, the code generator must be able to reorder the references as it does other operations in the trace. To see why, consider this fragment of a trace:

1. $v[i] := e1$
2. $x := v[i]$
3. $v[j] := e2$
4. $y := v[j]$

Without knowing anything about the indices **i** and **j**, a compiler must assume that **i** could equal **j**, and thus that operation 3 must be executed after both 1 and 2; under this assumption, there is no available parallelism in the fragment. But if the compiler knew somehow that **I** and **J** were never equal, then 1 and 3 could be performed in parallel and 2 and 4 in parallel, a doubling in speed. Analogous situations arise from dereferencing pointers.

To achieve the most parallelism, the compiler must disambiguate as many memory references as possible, determining whether they could possibly be to the same memory location. Disambiguating pointer dereferences is tough; there are few obvious clues in the program to help the compiler determine whether two pointers might point at the same object. But in our target domain of scientific code, the inner loops consist almost entirely of array references, and it usually isn't hard to disambiguate such references.

The disambiguator is a separate module of the Bulldog compiler. The code generator asks the disambiguator questions of the form, "Can these two vector references possibly refer to the same memory location?" The disambiguator answers yes, no, or I-don't-know. The I-don't-know answers are the ones that restrict parallelism.

How does the disambiguator disambiguate two vector references $v[i]$ and $v[j]$? Using the conventional flow analysis of reaching definitions, the disambiguator derives symbolic expressions e_i and e_j for the indices **i** and **j** in terms of the induction variables and loop invariants of the loops enclosing the two references. It then compares the two expressions symbolically to see if they could possibly be equal; that is, it sees if there are any integer-valued solutions to the equation $e_i - e_j = 0$.

For example, suppose that for the following code the code generator asked about the two vector references $v[j]$ and $v[k]$:

```

m := e1
FOR i := 1 to n DO {
    j := i + m
    v[j] := e2
    k := j + 1
    x := v[k]
}

```

The disambiguator derives the expressions $i+m$ for index j and $i+m+1$ for index k . The two indices are equal if and only if $(i+m)-(i+m+1)=0$. The disambiguator simplifies that to $-1=0$ and concludes that j could not possibly equal k ; therefore, $v[j]$ and $v[k]$ refer to different memory locations.

Often the equation doesn't simplify so neatly; for example, what about $4I+2J+1=0$? Finding solutions to integer-valued equations is a well known problem in number theory, and for linear equations the question is easy to answer; luckily almost all derivations of vector indices in scientific code are linear. (There are no integer solutions to $4I+2J+1=0$.)

But what about the copying of operations resulting from code motions during trace scheduling? The program is continually changing due to these copies being inserted in the flow graph, and it might seem that the flow analysis information must be incrementally recomputed after each trace. Fortunately this is not the case, and a static reaching analysis is sufficient [15]; intuitively, this is because the trace-scheduling algorithm preserves, in a loose sense, the reaching definitions of copies.

Our experience so far has been that this simple method of disambiguation completely disambiguates most memory references in most scientific programs. But this isn't good enough—if only two references in an inner loop were not disambiguated, actual parallelism could decrease by half or even more. Unfortunately, we've found that to make the disambiguator more sophisticated would not only be difficult to implement, but it would also make compilation unacceptably more expensive. And we had several example loops, including the inner loop of Fast Fourier Transform, that could be easily disambiguated by hand, but for which we had no practical automatic techniques. So no matter what level of disambiguator functionality we settled on, it still wouldn't be able to handle all the time-critical inner loops of scientific programs.

Some way was needed for the programmer to tell the compiler that two memory references are indeed to different locations. We've implemented an assertion facility by which the programmer can tell the compiler key facts about the program; if the compiler can't automatically distinguish two memory references, it consults the programmer-supplied assertions.

For example, suppose that the compiler can't disambiguate the references in this code fragment:

```

x      := v[i]
v[j+k+i] := y

```

The programmer can add an assertion:

```

ASSERT j+k > 0
x      := v[i]
v[j+k+i] := y

```

that the compiler uses to deduce that the two vector references are to different locations.

How does the programmer know where assertions are needed? The compiler tells him. Whenever it encounters two references it can't distinguish, it prints out information identifying them and the simplified symbolic expression representing the difference of the vector indices. For the above example, it would print out the question:

$j+k = 0?$

So far, it has always been immediately clear to the programmer what assertions are needed to completely disambiguate the program. Typically, only one or two assertions are required for any one program; the compiler performs all the drudge work of applying the assertions to disambiguate individual memory references.

The Global Memory Bottleneck

Many designs of parallel architectures fail because of lack of memory bandwidth. They have small, fast, local memories clustered around the computing elements, with large aggregate data stored in a larger, slower, shared global memory. For programs that manipulate large aggregates, especially for scientific programs, the global memory is a severe bottleneck; it can't fetch and store elements of the aggregate data fast enough to keep the computing elements busy. Put another way, it is easy to build a dual-ported memory, but very hard (and expensive) to build an 8- or 16-ported memory.

Most fast machines use a cache combined with interleaved memory banks to provide higher bandwidth. For example, by putting even addresses into one bank and odd addresses into another, the bandwidth doubles, since the two banks operate in parallel. But this design doesn't scale up easily, because there is still a single central controller that accepts memory requests and distributes them to the individual banks. Servicing two requests at a time is easy; servicing 8 or 16 at a time becomes a nightmare.

We solved the memory bottleneck problem as we solved other problems, using a combination of new architecture and smart software. We noticed that in scientific programs most of the memory references result from small inner loops enumerating through the elements of large arrays. Further, the central memory controller isn't really needed for those accesses, since the particular bank of each access could be predicted at compile time. If computing elements could access individual banks without going through the central controller, the memory bottleneck would be alleviated.

Unfortunately, even in scientific code it is not always possible to compute the banks of memory references at compile time. Even if the architecture supports direct reference to banks, it must still support general references for which the bank is not known statically.

In the ELI architecture, each memory bank has a **frontdoor** and a **backdoor**. The frontdoor provides direct access for memory references known at compile time to be in the bank. The backdoors of all the banks are connected to a more traditional central memory controller; a memory reference whose bank is unknown at compile time must be made through the controller. If the compiler can statically determine the bank of a memory reference, it will generate code to reference the bank directly through the frontdoor; otherwise, it will generate a slower backdoor reference.

To determine the bank of a memory reference, the Bulldog compiler uses techniques very similar to memory disambiguation. Flow analysis is used to derive a symbolic expression for the index of a memory reference; the modulo of that index relative to the number of banks yields the bank. If the compiler can't uniquely determine the bank, the programmer can help by adding assertions.

The compiler also has to apply some source transformations. For example, consider the following implementation of vector addition:

```
FOR i := 1 TO n DO
    a[i] := b[i] + c[i]
```

Suppose we know that our machine has 8-way interleaving of memory. By unrolling the body of the loop 8 times:

```
FOR i := 1 TO n BY 8 DO
    a[i+0] := b[i+0] + c[i+0]
    a[i+1] := b[i+1] + c[i+1]
    a[i+2] := b[i+2] + c[i+2]
    a[i+3] := b[i+3] + c[i+3]
    a[i+4] := b[i+4] + c[i+4]
    a[i+5] := b[i+5] + c[i+5]
    a[i+6] := b[i+6] + c[i+6]
    a[i+7] := b[i+7] + c[i+7]
```

it isn't hard to determine at compile time the bank of each memory access within the loop, given the starting address of the vectors. In general, the compiler needs to unroll such loops some multiple of the number of banks.

More sophisticated compiler techniques are used when loops aren't as well behaved. For example, if the starting index of a loop is not a constant but a variable, a memory reference in the loop body could easily be in different banks for different executions of the loop. But by adding a special pre-loop, the compiler can guarantee that all the references in the loop body are to known banks. The pre-loop executes a copy of the loop body until the index reaches a known value modulo the number of banks, at which point control transfers to the main loop.

For example, given the following loop:

```
FOR i := m to n DO
    a[i] := b[i] + c[i]
```

the compiler (assuming 8 banks) would transform that into:

```
FOR i := m to n DO
    IF 0 = i MOD 8 THEN
        temp := i
        BREAK
    a[i] := b[i] + c[i]
FOR i := temp to n DO
    ASSERT 0 = i MOD 8
    a[i+0] := b[i+0] + c[i+0]
    a[i+1] := b[i+1] + c[i+1]
    a[i+2] := b[i+2] + c[i+2]
    a[i+3] := b[i+3] + c[i+3]
    a[i+4] := b[i+4] + c[i+4]
    a[i+5] := b[i+5] + c[i+5]
    a[i+6] := b[i+6] + c[i+6]
    a[i+7] := b[i+7] + c[i+7]
```

Code Generation

Generating machine code from intermediate basic blocks for a traditional architecture is well understood—compilers do it every day. The two main problems are register allocation and instruction selection. A compiler must decide whether to keep particular values in memory or in registers. It must also map intermediate operations onto one or more machine instructions, which may be difficult if the machine has a rich instruction set.

The problems faced by a VLIW compiler generating code for a large trace are somewhat different and more complex.

Foremost, a VLIW compiler must worry about packing many machine operations into a single, large, parallel machine instruction. A traditional code generator merely outputs a stream of machine instructions, one or more per intermediate operation, that are appended together to form the object code. But a VLIW code generator must juggle the machine operations to get as many as possible to fit into each parallel machine instruction.

Because VLIWs are essentially reduced-instruction-set processors, there is no problem in selecting machine operators for intermediate code operations, since the intermediate code operations closely correspond to the machine level. But unlike a traditional machine, a VLIW offers many hardware functional units implementing the same operator, and the compiler must choose which one to use for a particular intermediate operation. Because of the long data paths between distant elements, the code generator must try to cluster operations to minimize data movement between elements. This problem is called **operation placement**.

For example, a VLIW machine may have 16 memory banks and 32 different functional units implementing the integer-add operation. To minimize data movement, the compiler must try to perform the vector indexing calculations on integer ALUs near the memory bank containing the vector elements.

Data routing is the problem of choosing data paths (buses and registers) to move data between elements of the machine. Between a source and destination there might be several paths, and the compiler must pick one that will least conflict with other activities. The move might take several hops between the source and destination, and the compiler must allocate a register after each hop to temporarily hold the value.

Finally, register allocation is tougher with a VLIW, since it could have at least as many register banks as functional units. The compiler must not only decide when to move a value into a register from memory but also which banks will hold the value. Sometimes it's advantageous to copy a value into several banks so that it can be used by many functional units simultaneously.

Obviously, operation placement, data routing, and register allocation are all interdependent. Compilers for existing horizontally-microcoded machines haven't had to deal with these problems because the target architectures offer little choice: An operation can be done in only one or two functional units, there are only one or two paths between any two points in the machine, and a functional unit is serviced by only one or two register banks.

We've built two code generators for the Bulldog compiler, one that uses a sophisticated strategy and one that uses a much simpler strategy but handles a more realistic range of machine models. The two code generators differ primarily in their approach to operator placement and register allocation.

The code generators get a trace of basic blocks as input and produce parallel machine code as output, treating the trace as if it were one very large basic block. Like many traditional code generators, our code generators convert the intermediate operations into a directed acyclic graph. The nodes of the DAG represent operations, and there is an edge between two nodes if one node uses the value produced by the other. They then form a **schedule** of machine instructions by traversing the nodes in some topological order, choosing machine operations for intermediate operators and filling the instructions of the schedule with the machine operations chosen. To prevent illegal code motions past jumps and to force undisambiguated memory references to be evaluated in the correct order, new edges are introduced to prevent one node from being evaluated before another.

The Operation-Scheduling Code Generator

Of the two code generators, the operation scheduler [17] uses the more sophisticated strategy. Operation placement, data routing, and register allocation are all delayed as long as possible, and the decisions about a particular intermediate operation are not made until the very point when the operation is placed on the schedule of machine instructions.

The parameterized machine model used by the current operation scheduler is limited in one important sense: Every functional unit has only a single feasible register bank to use for its result. This means that its register bank choices are in some cases fully constrained by the choice of functional units. But this is a restriction of the current implementation, not of the general technique.

To generate code for a trace, the operation scheduler forms an expression DAG. It then enumerates the nodes of the DAG (operations) in a topological order, placing the operations on the schedule of machine instructions. As each operation is considered, the code generator chooses a functional unit, data paths to deliver the operands to the functional unit, and a register bank to hold the result, and it finds cycles on the schedule where these actions can be placed.

To make these choices, the operation scheduler first calculates an earliest cycle that an operation could be scheduled based on the availability of operands. For each operand, a list is kept giving all the cycles and locations the operand is available. An operation can be started only after all the operands become available. (The reordering constraints of trace scheduling and disambiguation also affect the earliest cycle.)

The operand availability lists are also used to compute a search list of likely functional units for an operation. Functional units closest to the operands are considered first, and distant units are considered last. That is, the list is ordered by the longest data path of any operand to the functional unit.

```

proc SearchForBinding( operation )
  incr cycle from EarliestCycle( operation ) do
    for each fu in FunctionalUnitSearchList( operation ) do
      if fu is available at cycle and the operands
        can be fetched to fu's inputs by cycle
        and there is a register for the result at
        cycle
      then
        Schedule operation to take place in fu
        at cycle.
        Schedule the data movements for the
        operands and the result.
      return

```

Figure 3: Algorithm for binding intermediate operations

```

proc FindDataPath( start-bank, end-bank, start-cycle,
  due-cycle )
  if start-cycle > due-cycle then
    return false
  if start-bank == end-bank then
    return true
  incr cycle from start-cycle to due-cycle do
    for next-bank in SP[ start-bank, end-bank ] do
      if the data-path from start-bank to next-
        bank is
          available in cycle
      then if FindDataPath( next-bank, end-
        bank, cycle + 1,
          due-cycle )
      then
        return true
    return false

```

Figure 4: Algorithm for finding data paths

Figure 3 sketches the algorithm that binds intermediate operations to particular functional units, data paths, and register banks and schedules the machine operation.

Starting with the earliest cycle an operation could be scheduled, each functional unit in the search list is considered in turn. If the functional unit is not in use that cycle, if the operands can be moved to the inputs of the functional unit by that cycle, and if a register is available to hold the result, then the operation is scheduled on that functional unit in that cycle. Otherwise, the next cycle is considered, and the entire search process repeated.

The dynamic method for finding data paths relies on a short-path table indexed by register banks. For every pair of register banks R_i and R_j , $SP[R_i, R_j]$ gives a list of register banks immediately adjacent to R_i that are on short paths from R_i to R_j .

The search for a data path is performed by the recursive procedure shown in figure 4. **FindDataPath** returns true if it can find a path between register banks **start-bank** and **end-bank**. The parameter **start-cycle** is the first cycle that the value in question is available in the starting register bank, and **due-cycle** is the last allowable cycle the value may be delivered to the ending register bank. To find a path between **start-bank** and **end-bank**, the procedure looks for a path from **start-bank** to an adjacent register bank **next-bank**; if it finds

one, it then recursively looks for a path from **next-bank** to **end-bank**.

It's possible that the order in which operations are considered may affect the parallelism of the machine code. In general, there are many topological orderings of a DAG, and the code generator must choose one. We've experimented with several ordering heuristics, including height in the DAG (maximum distance to an exit) and estimated execution counts. The preliminary results have been mixed; there haven't been great differences between the heuristics.

The List-scheduling Code Generator

The list-scheduling code generator [4] is the simpler of the two code generators. Only a sketch of the algorithm will be given here.

The code generator uses a parameterized machine model capable of describing a large class of realistic VLIW architectures. The elements of the model are register banks, functional units (memory banks, adders, multipliers, etc.), and the connections between them. Arbitrary topologies of elements can be constructed. A shortest-path table is computed from the machine description giving the time delay and shortest path between any two elements.

Specified for each register bank are the number of registers and the number of input and output ports. Specified for each functional unit are the operations implemented by that unit, the time delay of the operations, and how frequently pipelined operations can be initiated. Associated with every register bank and functional unit are sets of resources required to perform the operations of that element; similarly, every point-to-point connection between elements has an associated set of resources required to move data across the connection. These resource sets let us describe conflicts due to hardware limitations, e.g. that only one of two buses may be used in any cycle, or that a memory bank can initiate at most two reads or writes every three cycles.

Generating code for a trace consists of three main phases: representing the trace as a directed acyclic graph, functional unit assignment, and list scheduling. The assignment phase picks functional units for each of the intermediate operations, and the list-scheduling phase then enumerates the nodes in a topological order, packing them into machine instructions.

The assignment phase is analogous to the register allocation of traditional compilers, and in fact was inspired by the top-down-greedy register-allocation algorithm [3]. Traditional register allocation tries to assign a limited set of registers to the operations of the DAG, minimizing the movement of data between registers and memory. Analogously, the assignment phase allocates functional units to intermediate operations, minimizing the costly movements of data between distant functional units.

The assignment algorithm simplistically assumes that the functional units are the only limited resource and that there will never be any bus or register-port conflict when moving values between functional units. Using a recursive procedure **Assign**, the code generator attempts to pick a good functional unit for each node (intermediate operation) in the DAG, making a guess as to which cycle

```

for each node with no successors (readers) do
    Assign( node, empty-set )

proc Assign( node, estimated-destinations )
    /* Assigns a functional unit to node. estimated-
       destinations is a guess as to the set of functional
       units where the value produced by node might be
       used. */

    if node is already assigned then
        return
    for each operand of node do
        Assign( operand, LikelyFUs( node, estimated-
            destinations ) )

    Pick one of LikelyFUs( node, estimated-destinations )
    and assign it to node. Estimate the the earliest cycle
    in which it can be scheduled and record the functional
    unit as being busy during that cycle.

proc LikelyFUs( node, estimated-destinations )
    /* Returns a set of functional units that could com-
       pute node and move its value to estimated-
       destinations as early as possible. */

```

Consider each functional unit capable of computing node. For each unit, estimate the earliest cycle that the values of the operands could be moved to the unit, the operation computed, and its value moved to the closest of estimated-destinations. Return the functional units having the earliest such cycles.

Figure 5: The functional-unit assignment algorithm

the operation will be scheduled. The measure of goodness of an assignment is how early the operation can be scheduled on the assigned functional unit and its produced value moved to the functional units of the operations reading the value.

Assign, shown in figure 5, recursively propagates from the exits to the entrances of the DAG estimates of where an operation can be best computed. When it reaches the entrance nodes, it then works its way back to the exit nodes, making final assignments of functional units to operations.

Once functional units have been assigned to operations of the DAG, the list-scheduling phase emits actual machine code by enumerating the nodes in a topological order and filling in the schedule of machine instructions. The instructions are formed in order: first cycle 0, then cycle 1, then cycle 2, etc. To form the next instruction, the list scheduler considers all nodes that are **data ready**, i.e. nodes all of whose predecessors have already been scheduled. It fills the instruction with as many of the data-ready operations as possible using first-fit; when no more can be squeezed into the current instruction, it is emitted and a new instruction started.

During assignment and list-scheduling the code generator is often faced with a choice of several nodes. For example, at each step in list-scheduling there are many data-ready nodes, only some of which will fit into the current instruction. In such cases, the code generator orders the nodes by height (maximum distance to an exit of the

DAG), on the assumption that the nodes of greatest height are the most time-critical and should take priority.

The destination register bank and register for a value produced by an operation are chosen on the fly when scheduling the operation. The list scheduler looks for an available register bank on the shortest path between the functional unit producing the value and the functional units that will be using the value.

Data movements between distant register banks are also scheduled on the fly during list scheduling. As soon as a value-producing operation is scheduled, the list scheduler looks at all the operations reading the value. If any are more than one register bank away, the list scheduler inserts special copy nodes into the DAG between the producing node and the distant reading nodes that will move the value to the distant functional units. These copy nodes will be scheduled just like normal operations, getting the values to the reading functional units as early as free hardware resources will allow.

List scheduling Versus Operation Scheduling

How do the two code generators compare?

We haven't yet run extensive experiments, but preliminary results indicate that on simple machine models there is little difference in the quality of the object code produced. Why little difference? It would seem that the exhaustive branch-and-bound search methods of the operation scheduler would surely do better than the simple heuristics of the list scheduler. But the operation scheduler offers only a simplified machine model with few interdependencies; the critical resource in the model appears to be functional units, not data paths or register bank-access. Since the list scheduler assumes that the only critical resources are functional units, it's not surprising that there is no difference between the two code generators on the simplified model.

It's likely that with complicated machine models having limited data paths and complex topologies, an operation scheduler would generate better code than the list scheduler. But expanding the branch-and-bound search of the operation scheduler to efficiently handle more realistic machine models might make the operation scheduler more complicated.

As for compilation time, right now the list scheduler is slightly faster. Both code generators take time linearly proportional to the size of the input trace. But the list scheduler time is linearly proportional to the size of the machine model, whereas the branch-and-bound search of operation scheduling takes time exponentially proportional to the complexity of the machine model. We're not sure how severely this exponential factor might slow down operation scheduling with complex machine models.

The complexity of the implementations are roughly comparable (about 7000 lines of code), always an important consideration for practical compilers. Again, the operation scheduler might become significantly more complicated when it is expanded to handle more realistic machine models.

Looking ahead, perhaps a combination of the two code generators might provide the best solution. The functional unit assignment algorithm of the list scheduler

could be used to heuristically guide the branch-and-bound search of the operation scheduler.

Preliminary Results

We're currently running extensive experiments measuring the performance of our compiler. As test data we're collecting a quite respectable library of scientific Fortran routines. We're running all of the routines through the compiler and measuring their performance on four machine models.

The **ideal machine** has infinite resources: infinite registers and functional units, no communications penalty, and 1-cycle operations. Performance of the ideal machine shows how much parallelism trace scheduling and disambiguation can find; parallelism is measured by taking the ratio of sequential operations to ideal machine instructions.

The **simple ELI**, used by both the operation-scheduling and the list-scheduling code generators, is an 8-cluster machine similar to figure 1, each cluster having 4 functional units connected by a complete crossbar to a multi-ported register bank. The simple ELI is more realistic than the ideal machine, but is still impractical.

The **realistic ELI** is an 8-cluster machine, each cluster having partial crossbars and multiple, practically ported register banks. It is very close to the actual ELI currently being designed. Only the list scheduler can generate code for the realistic ELI.

The **pipelined-sequential machine** is a traditional machine "built with the same technology" as the realistic ELI. But it has one cluster for which only one pipelined operation can be initiated every cycle, and the register bank allows only 2 reads and 1 write every cycle. This model resembles a CDC 6600 or the scalar portion of the Cray.

Comparing the simple and realistic ELI models with the pipelined-sequential model will tell us how much of the extra parallelism offered by the ELI models is actually being used by the compiler.

Our library currently consists of:

Simple matrix operations (multiply, transpose, add, reduce)

Generating prime numbers

Convolution

FFT

LU decomposition

Tridiagonal solvers (3 versions)

Quicksort

LINPAK inner loops (LINPAK is a widely used Fortran package for solving linear systems)

Soon we'll be adding the following routines taken from Forsythe, Malcom, and Moler [8]:

Spline interpolation

Integration using adaptive quadrature

Initial value ODE, RKF45

Solving non-linear equations

One-dimensional optimization

Singular value decomposition

Complete results for running all of these programs on all four models will be presented at the conference and in

the theses soon to appear [4, 17]. We've compiled all of the first group for the ideal machine and a few for the simple ELI and the realistic ELI.

So far we've only had time to analyze Fast Fourier Transform in any depth. On the ideal machine, we've got a speedup of 47 (that's the ratio of sequential ideal instructions to parallel ideal instructions). On the simple 8-cluster ELI we've got a speedup of 7.5 (the ratio of pipelined-sequential instructions to simple ELI instructions). The input vector size was 512.

Of the other programs, we've run many of them through the compiler and generated correct code, but we haven't analyzed and tuned them yet for performance (disambiguating memory references, unrolling and reorganizing loops for maximum parallelism, etc.). However, their performance on the ideal machine shows how much parallelism the trace scheduling and disambiguation are finding. From the results shown in figure 6 you can see that the Bulldog compiler is finding quite a bit of parallelism in ordinary scientific code. We're confident that on most of the routines we'll get realistic ELI performance similar to FFT's performance.

Program	Speedup
FFT	47
Tridiagonal solver	9
LU decomposition	12
LINPAK inner loops	11
Prime number generation	13
Matrix multiply	25*
Convolution	25*

*The parallelism found by the compiler in these simple programs is limited only by the size of the input data.

Figure 6: Ideal machine speedup (sequential instructions/parallel instructions)

Previous Work

Early parallelism experiments [9, 19] indicated that there was very little available parallelism in ordinary programs. The pessimism of those experiments combined with the difficulty of hand-coding VLIWs focused research on multiprocessors, vector machines, and data flow machines and away from VLIWs.

Data flow machines are still a gleam in the researcher's eye. Maybe they'll eventually provide thousand-fold parallelism, but there are still too many unsolved problems. Meanwhile, the ELI project has demonstrated a practical hardware and software architecture that offers mere ten-fold speedups right now.

There has been little success in compiling programs for multiprocessors. For example, the Cm* project [11] was hamstrung by the difficulty in distributing programs among the multiple processors.

The major effort in automated code production for vector machines and multiprocessors was undertaken at the University of Illinois. Kuck and his group developed a system, Parafrase, whose main goal is to generate code for fast, highly parallel machines [16]. Parafrase relies on ex-

tensive global data-dependence analysis and no global flow information. A memory-reference disambiguation mechanism eliminates superfluous dependency edges in the data-dependency graph due to ambiguous array references [2]. Using a large library of source transformations, Parafrase attempts to fit the available parallelism to the target architecture. Because the architectures cannot use fine-grained, operation-level parallelism, the disambiguator and the transformations operate at a coarse, all-or-nothing level, ignoring anything that cannot fit the mold. As a result, Parafrase ignores large amounts of parallelism existing in ordinary programs.

Many of the ideas of our project were originally motivated by the research on the compilation of high level languages into horizontal microcode. Fisher's thesis introduced the concept of trace scheduling and discussed heuristics for using list scheduling to generate horizontal microcode from vertical microcode [5]. Sites used list scheduling to optimize the code for the scalar, pipelined portion of the Cray [18]. Since then there have been many papers about variants on trace scheduling and other techniques attempting translation into horizontal microcode [12, 13].

Our research significantly differs from this microcode research. First, the VLIW architectures we've been studying are not horizontally-microcoded machines—they are reduced-instruction-set processors that hide the detailed complexity and asymmetry of microcode, while offering many times the architectural parallelism of current microcoded machines. Second, most of the microcode compilation techniques generate vertical microcode with registers, functional units, and data paths already assigned; then they try to compact the vertical code into horizontal code, perhaps reassigning registers and functional units in the process [10]. Finally, most of the microcode research is paper research with very few people actually building real compilers.

Acknowledgements

Other participants in the ELI project include John O'Donnell, Charles Marshall, Abhiram Ranade, Mark Sidell, Doug Baldwin, and Richard Kelsey.

This work was supported in part by the National Science Foundation grants #MCS 81-06181 and #MCS 83-08988 and the Office of Naval Research contract #N000014-82-K-0184.

References

- [1] A. V. Aho and J. D. Ullman.
Principles of Compiler Design. Addison-Wesley, 1977.
- [2] Utpal Banerjee.
Speedup of ordinary programs.
Technical Report UIUCDS-R-79-989, University of Illinois Department of Computer Science, October 1979.
- [3] William A. Barrett and John D. Couch.
Compiler Construction: Theory and Practice. Science Research Associates, Chicago, 1979, pages 581-587.

- [4] John R. Ellis.
Bulldog: A Compiler for VLIW Architectures.
 PhD thesis, Yale University, July 1984.
 Expected.
- [5] J. A. Fisher.
 The optimization of horizontal microcode within and beyond basic blocks: An application of processor scheduling with resources.
 U.S. Department of Energy Report COO-3077-161,
 Courant Mathematics and Computing
 Laboratory, New York University, October 1979.
- [6] Joseph A. Fisher.
 Very long instruction word architectures and the
 ELI-512.
 In *The 10th Annual International Symposium on Computer Architecture*, pages 140-150. IEEE Computer Society and Association for Computing Machinery, June 1983.
- [7] Joseph A. Fisher and John J. O'Donnell.
 VLIW Machines: Multiprocessors we can actually program.
 In *Compcon 84*, pages 299-305. IEEE Computer Society, February 1984.
- [8] George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler.
Computer Methods for Mathematical Computations.
 Prentice-Hall, 1977.
- [9] C. C. Foster and E. M. Riseman.
 Percolation of code to enhance parallel dispatching and execution.
IEEE Transactions on Computers 21(12):1411-1415, December 1972.
- [10] John Hennessy and Thomas Gross.
 Postpass code optimization of pipeline constraints.
ACM Transactions on Programming Languages and Systems 5(3):422-448, July 1983.
- [11] Anita K. Jones and Edward F. Gehringer, editors.
 The Cm* multiprocessor project: A research review.
 Technical Report CMU-CS-80-131, Computer Science Department, Carnegie-Mellon University, July 1980.
- [12] Association for Computing Machinery.
12th Annual Microprogramming Workshop, 1979.
- [13] Association for Computing Machinery and IEEE Computer Society.
The 16th Annual Microprogramming Workshop, 1983.
- [14] Alexandru Nicolau and Joseph A. Fisher.
 Using an oracle to measure parallelism in single instruction stream programs.
 In *14th Annual Microprogramming Workshop*, pages 171-182. ACM Special Interest Group on Microprogramming, October 1981.
- [15] Alexandru Nicolau.
Parallelism, Memory Anti-aliasing and Correctness Issues for a Trace Scheduling Compiler.
 PhD thesis, Yale University, June 1984.
 Expected.
- [16] D. A. Padua, D. J. Kuck, and D. H. Lawrie.
 High speed multiprocessors and compilation techniques.
IEEE Transactions on Computers 29(9):763-776, September 1980.
- [17] John C. Ruttenberg.
Delayed Binding Code Generation for a VLIW Supercomputer.
 PhD thesis, Yale University, June 1984.
- [18] Richard L. Sites.
 Instruction ordering for the Cray-I computer.
 Technical Report CS-023, Department of Electrical Engineering and Computer Science, University of California at San Diego, July 1978.
 Ellis remembers reading this six years ago. He's talked to Sites, who remembers his work on this problem quite well, but doesn't remember writing the tech report. Ellis has also talked to the secretary responsible for distributing UCSD Computer Science reports, and she claims this report really does exist. But we haven't yet received our copy.
- [19] G. S. Tjaden and M. J. Flynn.
 Detection and parallel execution of independent instructions.
IEEE Transactions on Computers 19(10):889-895, October 1970.

Intel's MMX Speeds Multimedia

Instruction-Set Extensions to Aid Audio, Video, and Speech

By Linley Gwennap

The first major extension to the x86 instruction set since 1985 will greatly improve the venerable architecture's handling of emerging multimedia applications. Collectively known as MMX, these 57 new instructions accelerate calculations common in audio, 2D and 3D graphics, video, speech synthesis and recognition, and data communications algorithms by as much as 8x. Overall, users will see a 50–100% performance improvement or more on these types of programs when using MMX instructions, as Figure 1 shows.

Intel plans to implement MMX throughout its product line in 1997. The first instantiation of MMX will be the P55C, a Pentium derivative due in 4Q96. MMX will also be included in Klamath (see page 3), a cost-reduced Pentium Pro that we expect to debut in 1H97. By the end of 1997, these two devices (and their successors) will displace most or all of Intel's non-MMX processors. AMD plans to incorporate MMX in its future processors (see MPR 1/22/96, p. 4), and we expect Cyrix will follow suit.

MMX is designed to have no impact on the operating system, making it compatible with existing x86-based OSs. Applications can take advantage of MMX in two ways: either by calling MMX-enabled drivers, such as a graphics driver, or by adding MMX instructions to critical routines. Most applications will take the driver route.

Intel has been working with dozens of key software and hardware vendors for months to help them add MMX to their applications and drivers. Today's public disclosure of the instruction set will enable any programmer to begin recoding their software for the new instructions. The number of MMX-enabled drivers and applications will build quickly once P55C systems are released.

These new instructions will provide PC users with a highly visible performance boost on many of today's most performance-critical applications. This boost should foster increased growth in the PC market and give Intel a leg up on competitors, such as PowerPC, that are lagging in adopting similar technology.

Simple Software Model

The mark of a good organization is learning from past mistakes. In designing the MMX software model, Intel took pains to avoid creating any new modes or new user state that would complicate an already complex architecture. MMX instructions can be used in any processor mode and at any privilege level. They generate no new interrupts or exceptions. These features eliminate the need for changes in the operating system to allow use of the new instructions.

From the programmer's view, there are eight new MMX registers (MM0-MM7) along with new instructions that operate on these registers. But to avoid adding new state, these registers are mapped onto the existing floating-point registers (FP0-FP7). When a multitasking operating system (or application) executes an FSAVE instruction, as it does today to save state, the contents of MM0-MM7 are saved in place of FP0-FP7 if MMX instructions are in use.

The obvious drawback is that programs cannot use both FP and MMX instructions within the same routines, as both share the same register set. This is rarely an issue, since most programs don't use FP at all, and those that do typically

Continued on page 6

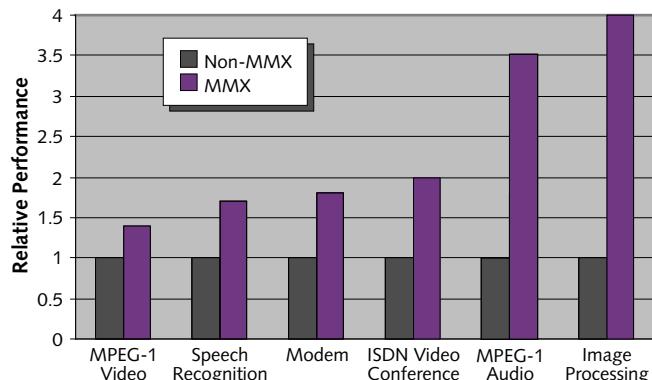


Figure 1. MMX improves performance on most multimedia applications by 50–100%, according to simulations of the forthcoming P55C processor. MPEG-1 performance refers to decoding; image processing is pixel manipulation, as in Photoshop. (Source: Intel)

Inside: Intel's River Map ◇ FX!32 ◇ DSPs: 56800, 320C2xx ◇ 3D Chips

MMX

Continued from page 1

use these calculations to generate data, while MMX is typically used in separate routines that display data. For 3D graphics, Intel recommends that geometry calculations remain in floating point while MMX is used to accelerate 3D rendering routines.

Without a mode bit, there is no foolproof way to prevent FP instructions from corrupting MMX data, and vice versa. Intel has taken some precautions, however, to trap the most common foolish situations. When data is loaded from memory into any MM register, it marks all the FP registers as busy, causing any subsequent FP instruction to trap. At the end of an MMX routine, the programmer must insert an EMMS instruction to restore the registers for FP use.

The converse situation is mostly covered. Taking advantage of the fact that the MMX registers are 64 bits wide while the FP registers are 80, MMX instructions always set the 16-bit exponent to NaN (not a number) while storing the result in the 64 fraction bits. Thus, although there is no way to trap an MMX instruction that is executed during a sequence of FP instructions, any subsequent FP calculation on the modified data will produce a floating-point exception. Alas, an

FST instruction could store the corrupted data in memory; Intel could not find an easy way to plug this hole.

Single Instruction, Multiple Data

The new instructions, listed in Table 1, use a SIMD (single instruction, multiple data) model, operating on several values at a time. In this respect, they are similar to multimedia instructions in the Motorola 88110, HP's PA-7100LC, and Sun's UltraSparc. Using the 64-bit MMX registers, these instructions can operate on eight bytes, four words, or two double words at once, greatly increasing throughput.

Figure 2 shows the three new data types: packed byte, packed word, and packed double word. (RISC devotees should note that Intel words are 16 bits, and double words are 32 bits.) These data types are particularly suited to multimedia because many algorithms work on small data sizes.

For example, audio data is usually stored in 8-, 12-, or 16-bit samples; the average person cannot appreciate further precision. Video is represented in pixels, commonly encoded as RGB (red, green, blue) triplets. Each of the three color values can be stored in 4, 6, or 8 bits; the last provides 16 million possible colors, more than most people can discern.

Most of the new mnemonics begin with "P" for packed; for example, PADD means packed add. The opcodes all begin

with the byte 0F, as do existing long jump, set byte, and Pentium-specific instructions. MMX uses previously reserved values for the second byte (none of which is used by other x86 vendors). The next two (or more) bytes provide the two operands, using the same encodings as other x86 instructions, except the target registers are the MMX registers, not the integer registers (EAX, etc.).

For example, the MOVD and MOVQ instructions can move data to and from memory using the same multitude of addressing modes as the standard MOV instruction; they also move data from one MM register to another. The MOVD instruction can even exchange data with the integer registers. Likewise, PADD performs register-to-register or memory-to-register operations, just like the integer ADD instruction. One exception is that register-to-memory mode is not supported in MMX.

Group	Mnemonic	Opcode*	Description
Data Transfer, Pack, Unpack	MOV[D,Q]	6E/7E,6F/7F	Move [double,quad] to/from MM register
	PACKUSWB	67	Pack words into bytes with unsigned saturation
	PACKSS[BW,DW]	63,6B	Pack [words into bytes, doubles into words] with signed saturation
	PUNPCKH [BW,WD,DQ]	68,69,6A	Unpack (interleave) high-order [bytes, words, doubles] from MM register
Arithmetic	PUNPCKL [BW,WD,DQ]	60,61,62	Unpack (interleave) low-order [bytes, words, doubles] from MM register
	PADD[B,W,D]	FC,FD,FE	Packed add on [byte, word, double]
	PADDS[B,W]	EC,ED	Saturating add on [byte, word]
	PADDUS[B,W]	DC,DD	Unsigned saturating add on [byte, word]
	PSUB[B,W,D]	F8,F9,FA	Packed subtraction on [byte, word, double]
	PSUBS[B,W]	E8,E9	Saturating subtraction on [byte, word]
	PSUBUS[B,W]	D8,D9	Unsigned saturating subtraction on [byte, word]
Shift	PMULHW	E5	Multiply packed words to get high bits of product
	PMULLW	D5	Multiply packed words to get low bits of product
	PMADDWD	F5	Multiply packed words, add pairs of products
Shift	PSLL[W,D,Q]	F1/71,F2/72, F3/73†	Packed shift left logical [word, double, quad]
	PSRL[W,D,Q]	D1/71,D2/72, D3/73†	Packed shift right logical [word, double, quad]
	PSRA[W,D]	E1/71,E2/72†	Packed shift right arithmetic [word, double]
Logical	PAND	DB	Bitwise logical AND
	PANDN	DF	Bitwise logical AND NOT
Logical	POR	EB	Bitwise logical OR
	PXOR	EF	Bitwise logical XOR
Compare	PCMPEQ[B,W,D]	74,75,76	Packed compare if equal [byte, word, double]
	PCMPGT[B,W,D]	64,65,66	Packed compare if greater than [byte, word, dbl]
Misc	EMMS	77	Empty MMX state

Table 1. Intel's MMX multimedia extensions include 57 new opcodes. Brackets indicate a set of options where only one may be chosen for a given instruction. B=byte, W=word, D=double word, Q=quad word.

*All opcodes start with 0F followed by the extension byte shown here. †Opcodes with 71, 72, and 73 as the second byte end with a third byte: Dr (PSRL), Er (PSRA), or Fr (PSLL), where "r" is the first operand.

Pack and Unpack Instructions Shuffle Bytes

In many cases, byte or word data is already stored in consecutive locations in memory and thus can be operated on by the new instructions. If the data is stored as aligned 32-bit values, however, it may be necessary to rearrange it to the packed format. The PACKxxDW instruction reads two double words from memory and combines them with two double words in a register, resulting in four packed 16-bit words.

If the original item exceeds the maximum value expressible in 16 bits, it is saturated: items that are too small are set to the smallest possible value, and items that are too large are set to the largest possible value. There are two options for calculating the saturation values, signed and unsigned, depending on how the source data is expressed. Similarly, the PACKxxWB instruction converts packed word data to packed byte data.

These operations can be reversed to unpack data. The mellifluous mnemonic PUNPCKxBW converts packed bytes into packed words, zero-extending the bytes, as Figure 3(a) shows. It can also interleave two sets of byte data into word data, as Figure 3(b) demonstrates. Similarly, PUNPCKxWD and PUNPCKxDQ convert packed words to double words and packed double words to quad words, respectively.

New Instructions Calculate in Parallel

Once the data is packed, calculations proceed in parallel. Each MMX calculation combines two 64-bit operands and produces a 64-bit result, so packed-byte instructions calculate eight results in parallel. Similarly, packed-word instructions generate four results, and instructions that operate on packed double words produce two results. Because most x86 instructions generate only one result at a time, this parallel calculation ability is the key to MMX's performance gains.

The performance boost is even greater on the P55C, due to the way MMX instructions are issued. The current Pentium design, while nominally two-way superscalar, executes only one FP calculation at a time and cannot pair these instructions with integer operations. The P55C allows pairing of MMX and integer instructions and can even pair MMX instructions with each other as long as they use different function units. Thus, the P55C can calculate up to 16 results (of one byte each) per cycle, helping to generate the performance gains seen previously in Figure 1.

The calculation instructions are all similar and, for the most part, straightforward. As Figure 3(c) shows, the PADDW (packed add word) instruction performs a parallel add of each of the four words in the source operand with the corresponding word in the destination operand, storing the result in the specified destination. Any carry out of a 16-bit addition is ignored. The results of the integer flags (carry, overflow, zero, etc.) are unchanged by any MMX calculation.

Both the add and subtract instructions can operate on packed bytes, words, or (in some cases) double words. Mnemonically, the suffixes B, W, or D are appended to indicate the data type. A set of logical operations (AND, AND

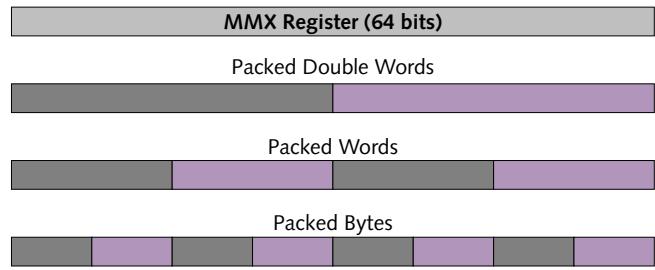


Figure 2. MMX adds three new data types: packed byte, packed word, and packed double word.

NOT, and OR) operate on a bit-by-bit basis and thus do not require a data-type suffix. They are identical to the existing logical instructions, except that they operate on MMX registers instead of integer registers.

New shift instructions differ from integer shift instructions in that each packed data element is treated individually. For example, PSLL (packed shift logical left) shifts items left while filling the lower bits of each with zeroes. The logical right shift fills the upper bits with zeroes, while the arithmetic right shift inserts sign bits. The shift count can be specified by an immediate value or an MMX register. These instructions can be used to quickly multiply or divide signed and unsigned data by powers of two.

Saturating and Unsaturating Arithmetic

The add and subtract instructions have three variations. The default (no suffix) option is simple, nonsaturating arithmetic. The other two options apply saturating arithmetic; as with the saturating PACK instructions, any overflow causes the result to be "clamped" to its maximum value, and underflows set the result to the minimum value. The suffix S indicates

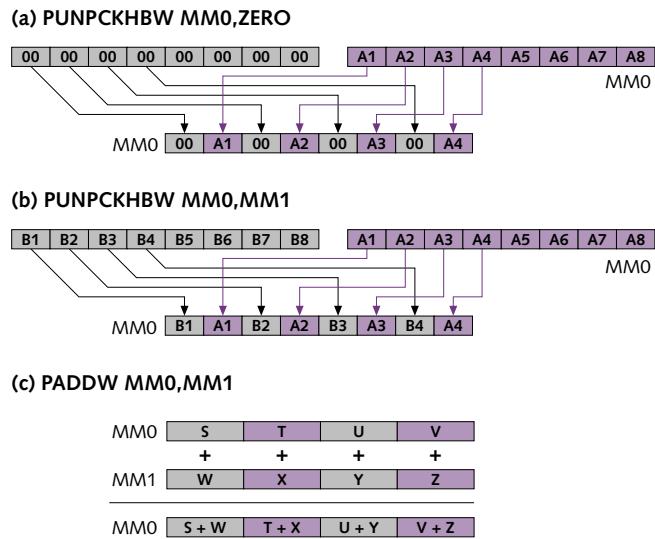


Figure 3. (a) Unpack operation converts packed bytes to words with zero extension. (b) PUNPCK can also interleave bytes from two MM registers. (c) Parallel add instruction calculates four 16-bit sums simultaneously.

cates signed saturating arithmetic; the most significant bit in each field is treated as a sign bit.

The third case is unsigned saturating (US) arithmetic, typically used for pixel operations. When two intensities, for example, are added, the result can never be whiter than white or blacker than black; saturating arithmetic handles this automatically, avoiding the long series of overflow and underflow checks needed with traditional instruction sets. In fact, a single PADDUSB instruction could replace 40 non-MMX x86 instructions.

These options are not completely orthogonal, a fact that should not surprise any x86 programmer. Although the nonsaturating form supports bytes, words, and doubles, the saturating forms cannot handle 32-bit data. The combination of the extra logic required to perform saturation with the longer carry chain of the 32-bit adder failed to meet the cycle-time requirement of the P55C.

For most situations, the saturating and nonsaturating forms are equivalent. In fact, it is dangerous to use the nonsaturating form for values that might cause an overflow, as it has no overflow trap. Of the nonsaturating adds and subtracts, only the 32-bit versions are typically used, to compensate for the lack of 32-bit saturating instructions.

Two Instructions Perform 16-bit Multiplication

Simple multiplication is handled by PMULHW and PMULLW. These instruction operate only on 16-bit values. Because the result of a multiplication can be twice the width of its operands, PMULHW stores only the high-order word of the result in the destination register. (Of course, it generates and stores four results in parallel.) In some situations, this 16-bit result will provide adequate precision.

For full 32-bit precision, the second half of the result is generated by PMULLW. The results of the two instructions must then be combined using PUNPCKWD, which interleaves the two destination registers. The following code multiplies the four words in MM1 by the four words in MM2, storing the

four products as two double words in MM1 and two double words in MM2:

MOVQ MM0, MM1	;Make copy of MM1
PMULHW MM0, MM2	;Calculate high bits in MM0
PMULLW MM1, MM2	;Calculate low bits in MM1
MOVQ MM2, MM1	;Make copy of low bits
PUNPCKHWD MM1, MM0	;Merge first two dwords
PUNPCKLWD MM2, MM0	;Merge second dwords

This code calculates four products in 6 cycles on a P55C, whereas a non-MMX Pentium requires 10 cycles to complete a single $16 \times 16 \rightarrow 32$ -bit integer multiplication.

Multiply-Add Speeds Signal Processing

The packed multiply-add instruction differs from the other calculation instructions in that the data type of the result is different from that of the source. As Figure 4(a) shows, PMADDWD multiplies two pairs of 16-bit words, then sums each pair, producing two 32-bit results. It executes in just three cycles on a P55C and is fully pipelined.

Multiply-add is at the heart of many audio and video algorithms, such as the fast Fourier transform (FFT). This procedure multiplies two vectors and accumulates the sum of the products. Using PMADDWD, a P55C can multiply and accumulate four vector entries per cycle (assuming the loop has been unrolled three times) while freeing the second pipe to perform loads, stores, index calculations, and branches. This is eight times the peak FP performance of a Pentium (which has no FP multiply-add instruction), not counting the advantage of executing instructions in the second pipe.

One drawback to MMX is the lack of a multiply or multiply-add for 32-bit operands. A fast 32-bit multiplier consumes four times more die area than a 16-bit multiplier, and Intel felt this feature was not worth the extra area. Besides, multiplication of 32-bit data can be performed using the standard integer multiply instruction. Although this instruction takes 10 cycles in the Pentium core and is not pipelined, it requires 4 cycles on Pentium Pro (and presumably Klamath) and, more important, is fully pipelined.

The integer multiplier, however, operates on the integer registers, not the MMX registers, and it cannot perform parallel calculations like the MMX units. Furthermore, there is no integer multiply-add instruction in x86. Because 16-bit precision is inadequate for advanced audio algorithms, such as wavetable sound, and for most 3D geometry calculations, the lack of a 32-bit multiply-add prevents these types of routines from taking advantage of MMX.

Parallel Comparisons Eliminate Branches

The MMX extensions include parallel compare operations that seem awkward at first but will produce big performance savings, particularly for Klamath and its successors. The PCMPEQW instruction, for example, compares two packed words; the fields in the result are set to zero if the comparison is false (not equal, in this case) or all ones if the comparison is true (equal), as Figure 4(b) shows.

(a) PMADDWD MM0,MM1

MM0	S	T	U	V
	×	×	×	×
MM1	W	X	Y	Z

MM0	(S × W) + (T × X)	(U × Y) + (V × Z)
-----	-------------------	-------------------

(b) PCMPEQW MM0,MM1

MM0	21	35	47	58
	==	==	==	==
MM1	21	75	44	58

MM0	FF	00	00	FF
	true	false	false	true

Figure 4. (a) Packed multiply-add sums two pairs of products with a single instruction. (b) Packed compare-if-equal compares packed words and generates a packed Boolean output, where all zeroes indicates false and all ones indicates true.

This function is useful when combining or overlaying two images. For example, a common video technique known as chroma keying allows an object (such as the weatherman) in front of a blue screen to be superimposed on another image (such as the weather map). In a digital implementation, this technique requires combining two images such that any blue pixels in the first image are replaced by the corresponding pixels in the second image.

Assume that $X[i]$ is the first image, $Y[i]$ is the background image, and the result is put back into $X[i]$. Using traditional x86 code, a single iteration might look like this:

```
CMP X[i], BLUE      ;Check if blue
JNE next_pixel     ;If not, skip ahead
MOV X[i], Y[i]      ;If blue, use second image
```

In this case, three instructions are needed per pixel.

Using MMX instructions, this sequence can be recoded as follows, assuming 16-bit pixels:

```
MOV MM1, X[i]        ;Make a copy of X[i]
PCMPEQW MM1, BLUE   ;Check four pixels in X[i]
PAND Y[i], MM1       ;Zero out non-blue pixels in Y
PANDN MM1, X[i]      ;Zero out blue pixels in X
POR MM1, Y[i]        ;Combine two images
```

Note that this sequence assumes all pixels are in MMX registers. The compare instruction generates four results in register MM1, setting each to zero if the corresponding pixel is not blue. The PAND combines this result with $Y[i]$, zeroing any pixels corresponding to non-blue values in $X[i]$. Conversely, the PANDN zeroes the blue pixels in $X[i]$.

At first glance, this routine appears to be about 2.5× faster than the non-MMX routine, processing four pixels in five instructions. The actual performance will be even better, however, because the second routine eliminates a branch. Although modern processors predict branches, they mispredict perhaps 10–20% of the time. Pentium's misprediction penalty is 4–5 cycles, while Pentium Pro (and presumably Klamath) takes an average of 15 cycles to recover from a misprediction. Thus, eliminating branches in this way significantly improves performance.

Outshined by VIS But Ahead of Others

In many ways, MMX is quite similar to the VIS instruction set (see MPR 12/5/94, p. 16) developed by Sun for UltraSparc. Both MMX and VIS pack 8-, 16-, and 32-bit data into 64-bit registers for parallel operations, including addition, multiplication, comparisons, and logical operations. Both use the floating-point registers to store these values. Both perform saturating and unsaturating arithmetic.

To this baseline feature set, VIS adds some highly specialized instructions. For example, PDIST calculates the sum of the absolute values of the differences of two sets of eight pixels. This instruction vastly accelerates the motion estimation process in MPEG and other video-compression algorithms, allowing UltraSparc to perform real-time MPEG-1 encoding. MMX will accelerate motion estimation compared with non-MMX processors, but not as much as VIS.

For More Information

For more information on MMX, obtain the document *Intel MMX Technology Programmer's Reference Manual* by calling Intel at 800.628.8686, or access the Web at www.intel.com/pc-suppl/multimed/mmxt/index.htm, or contact your local Intel sales office.

UltraSparc also includes instructions to accelerate the discrete cosine transform (used in video decompression), pixel masking, and 3D rendering. As with other SPARC instructions, the VIS instructions use three-operand encoding rather than the two-operand MMX style, which would alleviate some of the awkwardness seen in the above MMX code examples.

The VIS instructions also operate on 32 registers instead of the limited set of 8 MMX registers. A 4×4 matrix of constants, commonly used in digital filters, fits within the VIS register set but requires extra memory accesses in MMX. MMX has one advantage in its multiply-add instruction; it takes two instructions to perform this task under VIS.

While MMX may not go quite as far as VIS, it provides a much wider range of multimedia-oriented instructions than any other popular instruction set. HP's recent processors include some parallel arithmetic (see MPR 1/24/94, p. 16) but operate on only two 16-bit quantities at once.

To date, the other leading desktop RISCs—PowerPC, MIPS, and Alpha—have somehow failed to implement multimedia instructions, despite the significant performance benefits and minimal cost. One advantage that PowerPC has over current Intel chips is in floating-point performance, which can be used to speed audio processing and speech recognition, for example. For these multimedia applications, MMX processors should improve Intel's position.

Both NexGen and Cyrix have been developing their own multimedia extensions to the x86 instruction set. AMD's purchase of NexGen and its subsequent licensing agreement with Intel (see MPR 1/22/96, p. 5) ensure that company's processors will move to MMX, starting with the K6. Cyrix has said its M2 processor, due in early 1997, will include its own multimedia extensions. We expect Cyrix will eventually switch to MMX, although perhaps not in the first version of the M2.

MMX to Appear Mainly in Drivers

Programming in MMX is challenging. Taking full advantage of the SIMD architecture often requires unrolling loops and carefully arranging instructions, yet there is no compiler support planned other than allowing in-line MMX assembly code. Intel plans to provide libraries of routines for common multimedia functions as well as an assembler and debugger that support MMX. Over time, third parties will also supply MMX tools and library code.

Most multimedia applications will take advantage of the new instructions simply by calling MMX-enabled drivers or including the new library routines. One advantage of relying on drivers is that an application can automatically take advantage of a hardware accelerator for 3D graphics, sound, or MPEG decoding if one is installed. This model, of course, pushes the coding effort onto the driver writers.

A few applications will have to incorporate MMX instructions directly. These include programs that do image processing (e.g., Photoshop) or speech recognition, since APIs for these tasks are not yet defined. Fortunately, a significant speedup can often be obtained by simply modifying a few critical inner loops.

One problem is managing separate versions of each application for MMX and for non-MMX systems, which will be the majority of the installed base for several years. Software can check bit 23 of the CPUID to determine if a processor implements MMX. Again, simply relying on drivers eliminates this problem for the application.

Improved Multimedia Fuels PC Sales Growth

Although the installed base of MMX processors is nonexistent today, it will grow rapidly. We project that more than half of Intel's 1997 processor shipments, and virtually all thereafter, will contain MMX, totaling more than 30 million processors by the end of 1997. The 50–100% performance gain will motivate multimedia software vendors to use MMX; those that don't will be uncompetitive. Intel expects

dozens of drivers and applications to be shipping with MMX code when P55C systems first appear; this number will quickly increase during the course of 1997.

While the two are not directly connected, MMX is clearly designed to accelerate native signal processing (NSP). NSP will be used mainly in low-end systems to perform multimedia tasks; more expensive PCs are likely to include hardware accelerators. The P55C will significantly increase the baseline multimedia capabilities of low-end systems without accelerator chips. As Klamath reaches the mainstream in 1998, it will offer another big performance boost, possibly eliminating accelerator chips even in midrange systems.

It's not every day that you get a sizable step up in performance with minimal die cost, but that's what MMX promises. Once the combination of MMX-based processors and applications reaches the market, it should increase the growth of PC sales, particularly in the already hot consumer market, where multimedia is used most today. Even businesses will see the benefit as more use their PCs for video-conferencing and similar tasks.

One problem will be measuring this new performance level. Current benchmarks (SPEC95, Winstone, etc.) do not measure improvements of this type. The computer industry, with prodding from Intel, will most likely come up with new benchmarks to solve this problem, but perhaps not in time for the P55C's debut. Fortunately, the increase in graphics and video performance is something the buyer can see. **M**

S U B S C R I P T I O N O R D E R F O R M

Please start my subscription to Microprocessor Report.

Each subscription includes 17 issues per year (every three weeks). Include credit card information or payment in U.S. funds on U.S. bank (purchase orders accepted from U.S. companies only).

1-year subscription (U.S. & Canada: \$495, Europe: £375, Elsewhere: \$595)

2-year subscription (U.S. & Canada: \$895, Europe: £645, Elsewhere: \$1095)

Please add applicable sales tax for the following states: GA, IN, KY, MA, TX, or WA. Please add GST tax if in Canada.

My check is enclosed.

Bill my company: P.O. No. (*copy attached*) _____

Charge my: Visa MasterCard American Express

Card # _____ Exp. _____

Signature _____

Name _____ Title _____

Company _____

Address _____

City _____ State _____ Zip _____

E-Mail _____

Phone () _____

**MICRODESIGN
RESOURCES** 874 Gravenstein Hwy. South, Sebastopol, CA 95472;
Phone 707.824.4001, Fax 707.823.0504,
or e-mail to cs@mdr.zd.com

European Orders: Parkway Gordon, Westwood House, Elmhurst Rd.,
Goring, Reading, RG8 9BN, U.K.; Phone 441.491.875386; Fax 441.491.875524

**MICRODESIGN
RESOURCES**

M A R C H 14 D I N N E R M E E T I N G

The Future of Network-Centric, Low-Cost Computers

*Tim Hyland,
Director, Product Management
Oracle Corporation*

The concept of a network-centric \$500 computer has rapidly become one of the computer industry's most-discussed topics, and development work on various network-centric, low-cost computing devices is well under way. Here's your chance to hear the latest on Oracle's plans for such devices, direct from the manager of the company's Network Computer project, Tim Hyland.

The dinner meeting will be held at 6:00 pm on March 14 at The Westin Hotel, Santa Clara. The \$99 cost includes all handouts, dinner, wine, and hors d'oeuvres. Seating is limited and advance payment is required to reserve space.

**To register, call 800.527.0288
(outside U.S., call 707.824.4001)**



Performance and Energy Analysis of the Restricted Transactional Memory Implementation on Haswell

Bhavishya Goel, Ruben Titos-Gil, Anurag Negi, Sally A. McKee, Per Stenstrom
*Chalmers University of Technology
Gothenburg, Sweden*
`{goelb, ruben.titos, negi, mckee, per.stenstrom}@chalmers.se`

Abstract—Hardware transactional memory (HTM) implementations are becoming increasingly available. For instance, the Intel Core™ i7 4770 implements Restricted Transactional Memory (RTM) support for Intel Transactional Synchronization Extensions (TSX). We compare performance and energy expenditures of RTM to those of the TinySTM software transactional memory system by orthogonally varying workload characteristics using Eigenbench microbenchmarks. We show that crossover points exist over the range of workload characteristics for TinySTM and RTM performance and energy figures. We then apply the insights we gather to explain the performance and energy comparison results between RTM and TinySTM for the STAMP benchmark suite. We compare RTM performance overheads to those of other synchronization primitives like spinlock and compare-and-swap (CAS). Finally, we conduct a case study of a few STAMP applications to assess the impact of programming style on RTM performance and investigate what kinds of software optimizations can help overcome RTM’s hardware limitations.

I. INTRODUCTION

Transactional memory (TM) [1] simplifies some of the challenges of shared-memory programming. The responsibility for maintaining mutual exclusion over arbitrary sets of shared-memory locations is devolved to the TM system, which may be implemented in software (STM) or hardware (HTM). TM presents the programmer with fairly easy-to-use programming constructs that define a *transaction* — a piece of code whose execution is guaranteed to appear as if it occurred atomically and in isolation.

Academic research has explored this design space in depth, and a variety of proposed systems take advantage of transaction characteristics to simplify implementation and improve performance [2], [3], [4], [5]. Hardware support for transactional memory has been implemented in Rock [6] from Sun Microsystems, Vega from Azul Systems [7], and Blue Gene/Q [8] and System z [9] from IBM. Haswell is the first microarchitecture from Intel to provide such hardware support. Transactional Synchronization Extensions (TSX), the Intel instruction set extensions for HTM, allow programmers to run transactions on a best-effort HTM implementation — the platform provides no guarantees that hardware transactions will commit successfully, and thus the programmer is required to provide a non-transactional

path as a fallback mechanism. Intel TSX provides two software interfaces to execute atomic blocks: Hardware Lock Elision (HLE) is an instruction set extension to run atomic blocks on legacy hardware, and Restricted Transactional Memory (RTM) is a new instruction set interface to execute transactions on the underlying TSX hardware.

Here we compare the Haswell RTM’s performance and energy to those of other approaches for controlling concurrency. We use a variety of workloads to test the susceptibility of RTM’s best-effort nature to performance degradation and excessive energy consumption. We compare RTM performance to TinySTM, a software transactional memory (STM) implementation that uses time to reason about the consistency of transactional data and about the order of transaction commits. We find that although RTM often demonstrates performance advantages, TinySTM outperforms RTM under some conditions. We highlight these crossover points and analyze the impact of thread scaling on energy expenditure.

We find that RTM performs well with small to medium working sets when the amount of data (particularly that being written) accessed in transactions is small. When data contention among concurrent transactions is low, TinySTM performs better than HTM, but as contention increases, RTM consistently performs better. RTM generally suffers less overhead than TinySTM for single-threaded runs, and it is more energy-efficient when working sets fit in cache.

II. EXPERIMENTAL SETUP

We use an Intel 4th Generation Core™ i7 4770 processor for our experiments. The four physical cores can run up to eight simultaneous threads with hyper-threading enabled. The processor has 32 KB private L1 cache, 256 KB private L2 cache, and 8 MB shared L3 cache, with 16 GB of physical memory on board. We compile all microbenchmarks, benchmarks, and synchronization libraries using gcc v4.8.1 with `-O3` optimization flag. We use the `-mrtm` flag to access the Intel TSX intrinsics. We schedule threads on separate physical cores (unless running more than four threads) and fix the CPU affinity to prevent migration.

We modify the *task* example from *libpfn4.4* to read both the performance counters and the processor chip energy via the Running Average Power Limit (RAPL) interface. We

verify these energy figures against measurements at the ATX CPU power supply input of the motherboard and find them to be quite accurate.¹ We implement Intel TSX synchronization as a separate library and add RTM definitions to the STAMP *tm.h* file. When transactions fail more than eight times, we invoke reader/writer lock-based fallback code to ensure forward progress. If the return status bits indicate that an abort was due to another thread's having acquired the lock (in the fallback code), we wait for the lock to be free before retrying the transaction. The following shows pseudocode for a sample transaction.

Algorithm 1 Implementation of BeginTransaction

```

while true do
    nretries ← nretries + 1
    status ← _xbegin()
    if status = _XBEGIN_STARTED then
        if arch_read_can_lock(serialLock) then
            return
        else
            _xabort(0)
        end if
    end if
    {*** Fall-back path ***}
    while not arch_read_can_lock(serialLock) do
        _mpause()
    end while
    if nretries ≥ MAX_RETRIES then
        break
    end if
end while
arch_write_lock(serialLock);
return

```

III. MICROBENCHMARK ANALYSIS

A. Basic RTM evaluation

We first carry out microbenchmark studies to assess RTM's limitations. These tests and their results are explained below.

RTM Capacity Test. To test the limitations of read-set and write-set capacity for RTM, we create a custom microbenchmark, results for which are shown in Fig. 1. The abort rate of write-only transactions tops out at 512 cache blocks (the size of L1 data cache). We suspect this is because write-sets are tracked only in L1, and so evicting any transactionally written cache line from L1 results in a transaction abort. For read-sets, the abort rate saturates at 128K cache blocks (the size of L3 cache). This suggests that evicting transactionally read cache lines from L3 (but not L1) triggers transaction aborts, and thus RTM maintains performance for much larger read-sets than write-sets.

RTM Duration Test. Since RTM aborts can be caused by system events like interrupts and context switches, we carry out an experiment to study the effects of transaction duration (measured in CPU cycles) on success rate. For this analysis, we use a single thread, set the working-set size to 64 bytes, and set the number of writes inside the transaction to 0. This tries to ensure that the number of aborts due to memory events and conflicts remains insignificant. We gradually increase the duration by increasing the number

¹We graphed both data sets and verified that the resulting curves have the same shape (although their *y*-axis scales necessarily differ).

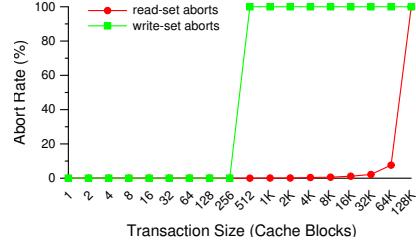


Figure 1. RTM Read-Set and Write-Set Capacity Test

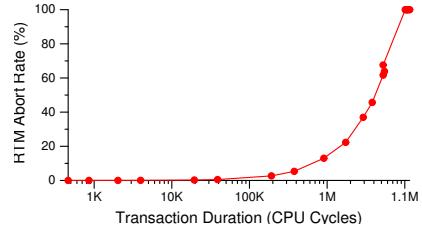


Figure 2. RTM Abort Rate vs. Transaction Duration

of reads within the transaction. Fig. 2 shows the results. Transaction duration begins to affect the abort rate beyond 30K CPU cycles, and when duration reaches more than 10M cycles, all transactions abort (although these figures are likely machine dependent).

RTM Overhead Test. Next we attempt to quantify the performance overheads that RTM incurs compared to spin locks and atomic instructions such as compare-and-swap (CAS). For this comparison, we create a microbenchmark that removes elements from a queue. We first compare RTM against the spinlock implementation found in the Linux kernel (*arch/x86/include/asm/spinlock.h*). For RTM, we simply retry the transaction on aborts. The queue comes from the STAMP [10] library; we modify a version to use CAS in *queue_pop()*. We initialize the queue with 1M elements and let threads extract elements until the queue is empty, with no static division of the work among threads.

We perform three sets of experiments. First, to observe the cost of starting a transaction in RTM when there is no contention, we run experiments with a single thread. We then repeat the experiment with four threads to generate a high-contention workload. Finally, we lower the contention by making threads work on local data for a fixed number of operations after each critical section completes. Table I summarizes execution times normalized to those of the lock-based version.

Contention	Type of synchronization			
	None	Lock	CAS	RTM
None	0.64	1	1.05	1.45
Low	N/A	1	0.64	0.69
High	N/A	1	0.64	0.47

Table I
RELATIVE OVERHEADS OF RTM VERSUS LOCKS AND CAS

Characteristic	Definition
Concurrency	Number of concurrently running threads
Working-set size	Size of frequently used memory
Transaction length	Number of memory accesses per transaction
Pollution	Fraction of writes to total memory accesses inside transaction
Temporal locality	Probability of repeated address inside transaction
Contention	Probability of transaction conflict
Predominance	Fraction of transactional cycles to total application cycles

Table II
TM CHARACTERISTICS STUDIED

Table I shows that the cost of starting a transaction in RTM is considerable, making it perform worse than the other alternatives when executing non-contended critical sections with few instructions. Compared to using locks and CAS, RTM suffers a slowdown of around 45%; compared to unsynchronized code the slowdown grows to a factor of two. In contrast, our multi-threaded experiments reveal that RTM exhibits roughly 30% and 50% lower overhead than locks in low and high contention, respectively, while CAS is in both cases around 35% better than locks. In contrast to locks and CAS, transactions avoid hold-and-wait behavior, which seems to give RTM an advantage for the scenarios analyzed. When comparing locks and CAS in this case, the higher overheads for locks is likely due in part to the ping-pong coherence behavior of the cache line containing the lock and to the cache-to-cache transfers of the line holding the queue head.

B. Eigenbench characterization

To compare RTM and STM in detail, we next study the behaviors of Hong et al.’s Eigenbench [11]. This parameterizable microbenchmark attempts to characterize the design space of TM systems by orthogonally exploring different transactional application behaviors. Table II defines the seven characteristics we use to compare performance and energy expenditure of the Haswell RTM implementation and the TinySTM [12] software transactional memory system. Hong et al. [11] provide a detailed explanation of these characteristics and the equations used to quantify them.

Unless otherwise specified, we use the following parameters in our experiments, results for which we average over 10 runs. Transactions are 100 memory references (90 reads and 10 writes) in length. We use one small (16KB) and one medium (256KB) working set size to demonstrate the differences in RTM performance. Since working set sizes less than 8 MB have little influence on TinySTM’s abort rate, we only show TinySTM results for the smaller working set size. To prevent L1 cache interference, we run four threads with hyper-threading disabled as our default, and we fix the CPU affinity to prevent thread migration. For each characteristic, we compare RTM and TinySTM performance and energy (versus sequential runs of the same code) and transaction-abort rates. For the graphs in which we plot two working-set sizes for RTM, the speedups and energy

efficiencies given are relative to the sequential run of the same size working set.

Working-Set Size. Fig. 3 shows Eigenbench results as we increase each thread’s working set from 8K to 128MB over a logarithmic scale. RTM outperforms TinySTM for smaller working sets. The performance of both RTM and TinySTM drops once the combined working sets of all threads exceed the 8MB L3 cache. RTM performance suffers more because events like eviction of read-set from L3, page faults, and interrupts trigger a transaction abort, which is not the case for TinySTM. Fig. 3(c) shows a steep rise in RTM abort rates as working sets approach the L3 cache size. The speedups of both RTM and TinySTM are lowest at working sets of 4MB: at this point, the parallelized code’s working sets (16 MB in total) exceed L3, but the working set of the sequential version (4 MB) still fits. For working sets above 4MB, the sequential version starts encountering L3 misses, and thus the relative performance of both transactional memory implementations begins to improve. TinySTM’s false conflict detection increases sharply when the working-set size reaches 16 MB. For working-set sizes above 64MB, RTM abort rates decrease compared to those for snakker working sets (4-32MB), which makes RTM outperform TinySTM. As for energy efficiency, RTM runs are more energy efficient than both TinySTM and the sequential runs for working sets of up to 1 MB.

Transaction Length. Fig. 4 shows Eigenbench results as we increase the transaction length from 10 to 520 memory operations. When the working set (16KB) fits within L1, RTM outperforms TinySTM for all transaction lengths. For 256 KB working sets, RTM performance drops sharply when the transaction length exceeds 100 accesses. Evicting write-set data from L1 triggers a transaction abort, but when the working set fits within the L1 cache, such evictions are few. As the working set grows, the randomly chosen addresses accessed inside the transactions have a higher probability of occupying more L1 cache blocks, and hence they have a higher likelihood of being evicted. In contrast, TinySTM shows no performance dependence on working set size. The overhead associated with starting the hardware transaction affects RTM performance for very small transactions. As observed in the working-set analysis above, RTM is more energy efficient than both the sequential run and TinySTM for all transaction lengths when using the smaller working set. When using the larger working set, RTM expends more energy for transactions exceeding 120 accesses.

Pollution. Fig. 5 shows Eigenbench results when we test symmetry with respect to handling read-sets and write-sets by gradually increasing the fraction of writes. The pollution level is zero when all memory operations in the transaction are reads and one when all memory operations are writes. When the working set fits within L1, RTM shows almost no asymmetry in its behavior. But for the larger working-set size, RTM speedup suffers as the level of pollution

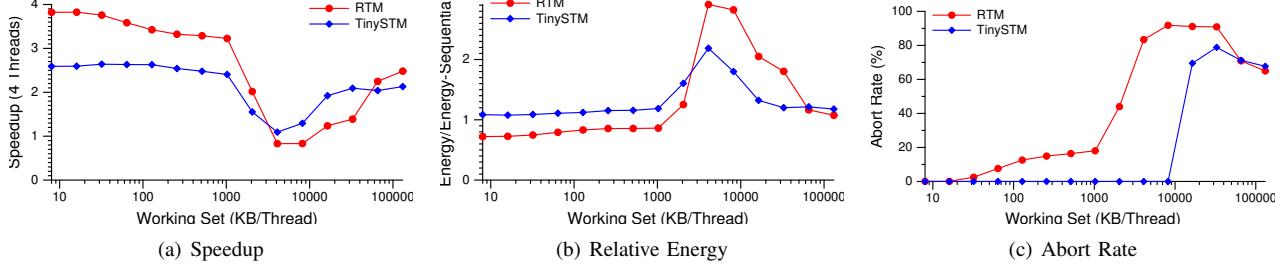


Figure 3. Eigenbench Working-Set Size Analysis

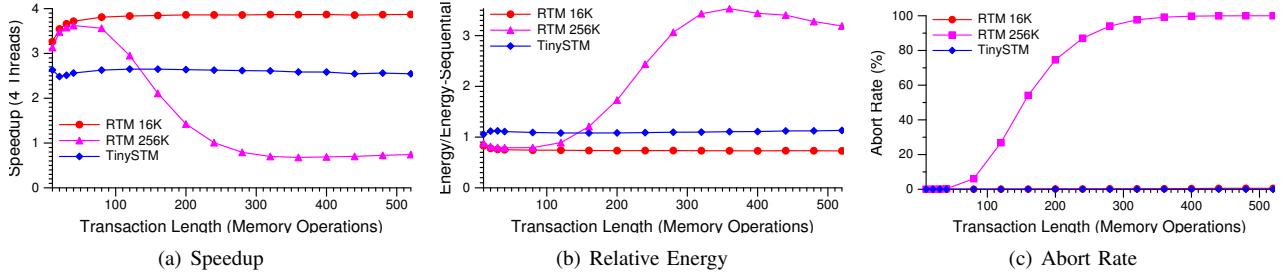


Figure 4. Eigenbench Transaction Length Analysis

increases. TinySTM outperforms RTM when the pollution level increases beyond 0.4.

Temporal Locality. We next study the effects of temporal locality on TM performance (where temporal locality is defined as the probability of repeatedly accessing the same memory address within a transaction). The results in Fig. 6 reveal that RTM shows no dependence on temporal locality for the 16KB working set, but performance degrades for the 256 KB working set (where low temporal locality increases the number of aborts due to L1 write-set evictions). In contrast, TinySTM performance degrades as temporal locality increases, indicating that it favors unique addresses unless only one address is being accessed inside the transaction (locality = 1.0).

Contention. This analysis studies the behavior of TM systems when the level of contention is varied from low to high. For this analysis, we set the working-set size to 64KB per thread. The level of contention is calculated as an approximate value representing the probability of a transaction causing a conflict (as per the probability formula given by Hong et al. [11]). The conflict probability figures shown here are calculated at word granularity and hence are valid only for TinySTM. Since RTM detects conflict at the granularity of cache line (64 bytes), the contention level is higher for RTM for the same workload configuration. When the degree of contention among competing threads is low, TinySTM considerably outperforms RTM. As contention increases, however, TinySTM performance degrades while RTM performance remains almost the same.

Predominance. We study the behavior of the TM systems

when varying the fraction of application cycles executed within transactions to the total number of application cycles. For this analysis, we use the larger working set (256 KB/thread), we set contention to zero, and we vary the predominance ratio from 0.125 to 0.875. Fig. 8 shows that performance for both TinySTM and RTM suffers as the predominance of transactional cycles over non-transactional cycles grows. This can be attributed to the overhead associated with TM systems: for the same level of predominance, TinySTM introduces more overhead because it must instrument the program memory accesses.

Concurrency. Next we study how the performance and energy of RTM and TinySTM scale compared to those of the sequential code when concurrency is increased from one thread to eight. Fig. 9 shows that RTM scales well up to four threads. When running eight threads, the L1 cache is shared between two threads running on the same core. This cache sharing degrades performance for the larger working set more than for the smaller working set because hyper-threading effectively halves the write-set capacity of RTM. In contrast, TinySTM scales well up to eight threads. For the small working set, RTM proves to be more energy-efficient than either TinySTM or the sequential runs.

The results from the Eigenbench analysis helps us in identifying a range of workload characteristics for which either RTM or TinySTM is better performing or more energy efficient. We next apply the insights gained from our microbenchmark studies to analyze the performance and energy numbers we see for the STAMP benchmark suite.

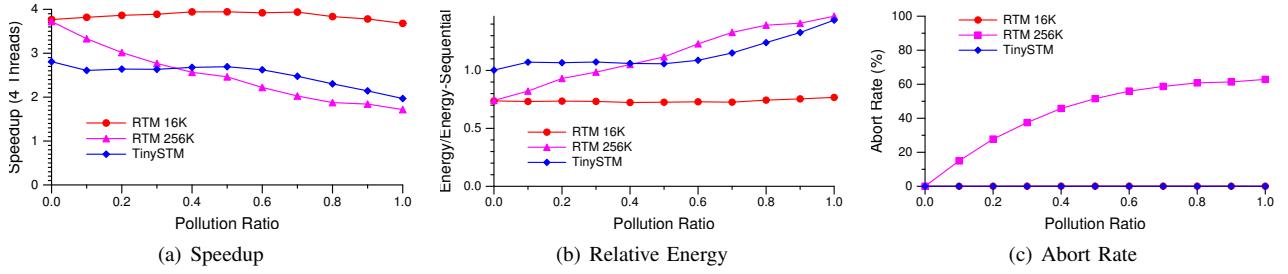


Figure 5. Eigenbench Pollution Analysis

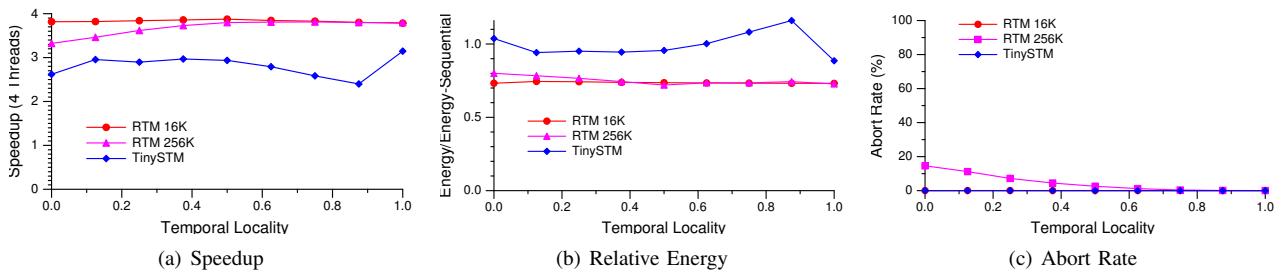


Figure 6. Eigenbench Temporal Locality Analysis

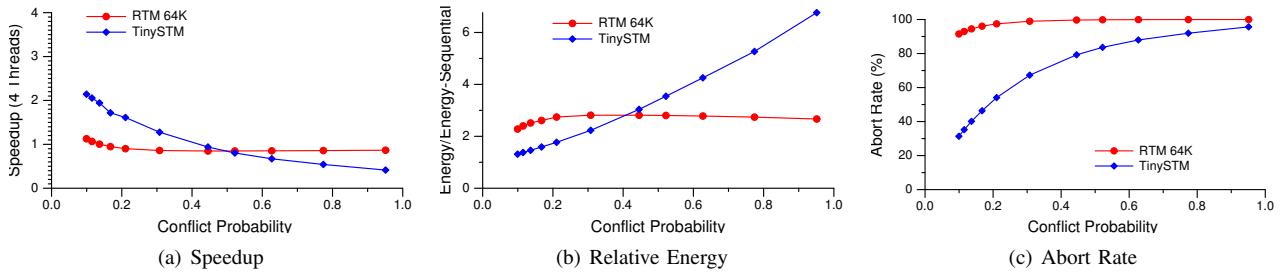


Figure 7. Eigenbench Contention Analysis

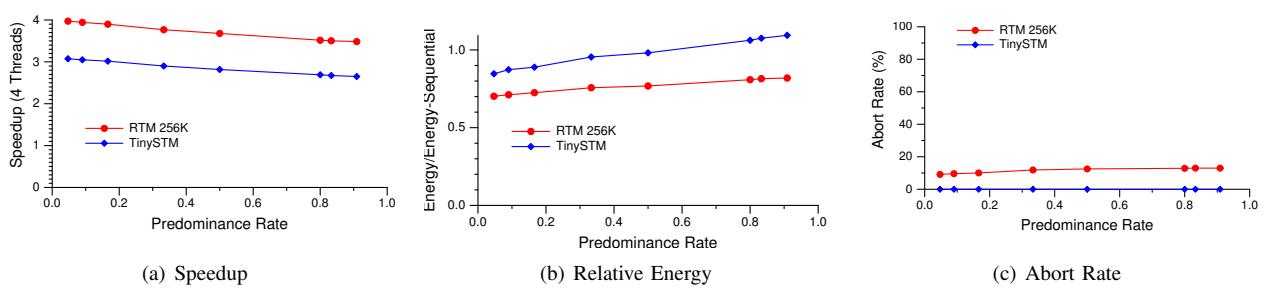


Figure 8. Eigenbench Predominance Analysis

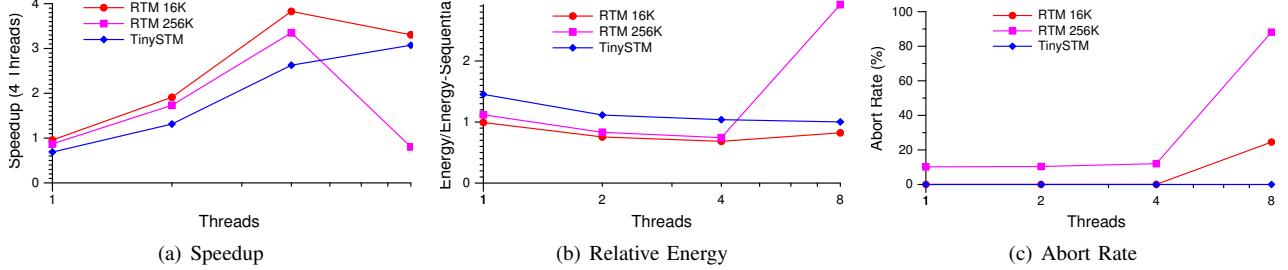


Figure 9. Eigenbench Concurrency Analysis

IV. HTM VERSUS STM USING STAMP

We next use the STAMP transactional memory benchmark suite [10] to compare the performance and energy efficiency of RTM and TinySTM. For this set of experiments, we use the lock-based fallback mechanism explained in section II. We run the benchmarks with input sizes recommended for running on hardware platform that create large working sets and high contention. All results are averaged over 10 runs. Fig. 10 shows STAMP execution times for RTM and TinySTM normalized to the execution time of a sequential (non-TM) run. Fig. 11 shows the corresponding energy expenditures normalized to the energy expenditure of a sequential run. For single-threaded runs, normalized execution time and energy of single-threaded TM versions of the benchmarks illustrate the overheads incurred by the TM systems.

`bayes` has large working set and long transaction length. As we saw from the Eigenbench transaction length analysis in Fig. 4, RTM performs worse compared to TinySTM for applications exhibiting such characteristics. As expected, RTM does not improve the performance of `bayes` as the number of threads scale, and TinySTM performs better overall. Since the execution time taken by algorithm to learn the network dependencies depends on the order of computations, we see significant deviations in learning time for multi-threaded runs.

`genome` exhibits medium transaction length and medium working-set size with low contention. The dominating transaction length in `genome` is less than 100 accesses. Recall that in the working-set analysis shown in Fig. 3(a) (for transaction length 100), RTM slightly outperforms TinySTM for working-set sizes up to 4MB. On the other hand, TinySTM performs better than RTM when contention is low (Fig. 7(a)). The confluence of these two factors within `genome` results in similar performances for both RTM and TinySTM up to four threads. For eight threads, as expected, TinySTM’s performance continues to improve, whereas RTM’s suffers from increased resource sharing among hyper-threads.

`intruder` is also a high-contention benchmark. As with `genome`, RTM performance scales well from one to four

threads. Since `intruder` executes very short transactions, scaling to eight threads does not cause as resource limitation issues as it did for `genome`, and thus RTM and TinySTM see similar performance. Even though this application uses a small to medium-sized working set — which might otherwise give RTM an advantage — its performance is dominated by very short transaction lengths that limit speedup.

`kmeans` is a clustering algorithm that groups data items in N -dimensional space into K clusters. Like `bayes`, we see significant deviations in execution times for the multi-threaded versions. On average, RTM performs better than TinySTM. The short transactions experience low contention, and the `kmeans` working set size is small with high locality, all of which work together to give RTM a performance advantage over TinySTM. Even though both TM systems show speedups over the sequential run, only RTM saves energy: synchronizing the `kmeans` algorithm in TinySTM expends more energy at all thread counts.

`labyrinth` routes a path in a three-dimensional maze, where each thread grabs a start and an end point and connects them through adjacent grid points. The results in Fig. 10 show that `labyrinth` does not scale in RTM. This is because each thread makes a copy of global grid inside the transaction, triggering capacity aborts that eventually cause the transaction to fall back to using a lock. Energy expenditure increases for the RTM multi-threaded runs because the threads try to execute the transaction in parallel but eventually fail, wasting many instructions while increasing cache and bus activity.

`ssca2` has short transactions, a small read-write set, and low contention, and thus even though it has a large working set, it scales well to higher thread counts. Performance for eight threads is good for both RTM and TinySTM. In general, RTM performs better (with respect to both execution time and energy expenditure) but not by a big margin, as expected for very short transactions.

`vacation` has low to medium contention among threads and medium working set size. The transactions are of medium length, locality is medium, and contention is low. Like `genome`, `vacation` scales well up to four threads, but the performance degrades for eight threads since its read-write set size is not small enough to avoid resource limitation

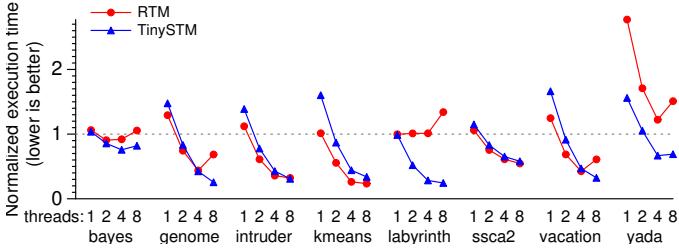


Figure 10. RTM vs TinySTM performance for STAMP benchmarks

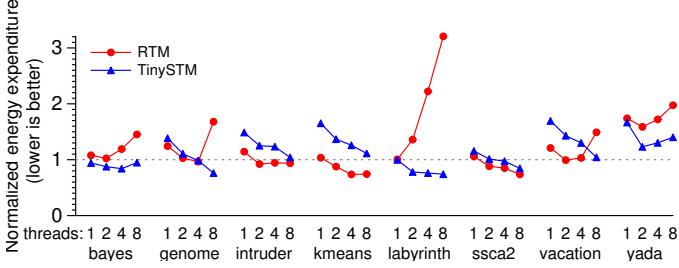


Figure 11. RTM vs TinySTM Energy Expenditure for STAMP benchmarks

issues arising from cache sharing.

yada has big working set, medium transaction length, large read-write set, and medium contention. All these conditions give TinySTM a consistent performance advantage over RTM at all thread counts.

A common observation from the STAMP results is that for the applications that have large read-write set size and large working-set size (bayes, labyrinth and yada), the energy trends do not follow performance trends. These applications expend more energy as they are scaled to more threads — possibly due to increased cache and bus activity — even when the performance remains constant or falls.

Fig. 12 shows the overall abort rates for all benchmarks, including the contributions of different abort types. As per our observations of hardware counter values, the current RTM implementation does not seem to distinguish between data-conflict aborts and aborts caused by read-set evictions from L3 cache, and thus both phenomena are reported as conflict aborts. When a thread incurs the maximum number of failed transactions and acquires the lock in the fallback path, it forces all currently running transactions to abort. We term the resulting abort a *lock* abort. These aborts are reported either as conflict aborts, *explicit* aborts (i.e., deliberately triggered by the application code), or both (i.e., the machine increments multiple counters). Such lock aborts are specific to the fallback mechanism we use in our experiments. Other fallback mechanisms that do not use serialization locks *within transactions* (they can be employed in non-transactional code) do not suffer such aborts. Note that avoiding lock aborts does not necessarily result in better performance since the lock aborts mask other type of aborts.

Abort Type	Description
Data-conflict/ Read-capacity	Conflict aborts and read-set capacity aborts
Write-capacity	Write-set capacity aborts
Lock	Conflict and explicit aborts caused by serialization locks
Misc3	Unsupported instruction aborts ^a
Misc5	Aborts due to none of the previous categories ^b

^aincludes explicit aborts and aborts due to page fault/page table modification
^binterrupts, etc.

Table III
INTEL RTM ABORT TYPES

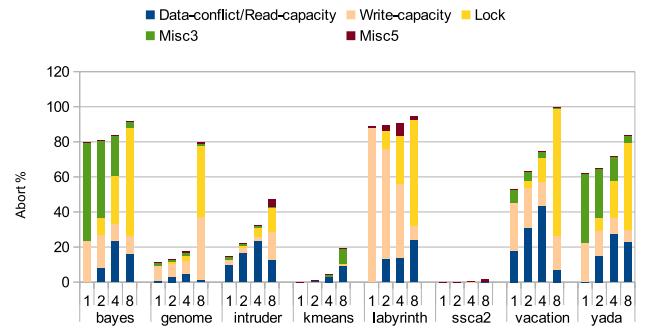


Figure 12. RTM Abort Distribution for STAMP benchmarks

This can be seen in abort contributions shown in Fig. 12. As the application is scaled, the fraction of aborts caused by lock increases. This is because every lock acquisition by a thread would potentially cause $N - 1$ lock aborts where N is the number of threads. If however, the lock was read outside transaction, the transactions that were aborted due to lock acquisition, would be allowed to continue and may abort due to data conflict or capacity overflow later.

The RTM_RETired:ABORTED_MISC3 performance counter reports aborts due to unsupported instructions, page faults, page table entry modifications, etc. The RTM_RETired:ABORTED_MISC5 counter includes miscellaneous aborts not categorized elsewhere, for example, aborts caused by interrupts. Table. III gives an overview of these abort types. In addition to these counters, three more performance counters represent categorized abort numbers: RTM_RETired:ABORTED_MISC1 counts aborts due to memory events like data conflicts and capacity overflows; RTM_RETired:ABORTED_MISC2 counts aborts due to uncommon conditions (values for which we always observed to be zero value in our experiments); and RTM_RETired:ABORTED_MISC4 counts aborts due to incompatible memory types (e.g., due to cache bypass or I/O access). Our observed values for the latter counter were always less than 20; we attribute this to hardware error, as per the Intel specification update [13].

V. IMPACT OF PROGRAMMING STYLE ON RTM PERFORMANCE

Transactions are intended to simplify synchronization in multithreaded programs. Unfortunately, the truth is that programmers who overlook the best-effort nature of the hardware where their applications run, may often find that their applications do not perform as expected. Programmers may inadvertently make design decisions that greatly limit the ability of the hardware to successfully commit transactions. In this section, we examine two popular transactional applications whose performance can be significantly improved with minimal programming effort — without finer-grain synchronization — simply by keeping in mind the general constraints of the hardware. We demonstrate that it is possible to minimize the occurrence of capacity aborts, conflict aborts, and other RTM-unfriendly events.

A. Case study I: intruder

This benchmark emulates a network intrusion detection system that processes packets in parallel. Packets are first captured from a stream, then reassembled into a complete flow and finally compared against a known set of intrusion signatures. In the reassembly phase, a red-black tree is used to keep captured packets: each node of the tree contains a list of packets belonging to the same flow (session). Flows are inserted at the point the first packet is captured. For subsequent packets, *intruder* finds the correct flow in the tree and inserts them in the corresponding list. When all packets of a flow have been collected, the flow is reassembled, removed from the tree, and pushed into a decoded queue. Flows are subsequently extracted and compared against a database of attack signatures. The main transaction encloses the reassembly phase in which a captured packet is inserted into the tree of incomplete flows.

Reducing read-set size and transaction duration. The reassembly phase maintains a sorted list of fragments in a flow, so each new packet captured is inserted into position according to its sequence number. Each reassembly transaction thus traverses a potentially long list to find the correct insertion location. Since program correctness does not rely on keeping captured fragments sorted at all times, we can simply prepend the fragments onto the list in constant time and only sort the list prior to reassembly. We thus reduce both the transaction footprint and the duration for the common case of inserting into an existing flow. The benefits are twofold: on the one hand, shorter-running reassembly transactions reduce the likelihood of conflicts with concurrent tree operations on other flows; and on the other hand, accessing fewer cache lines minimizes the likelihood of suffering capacity-induced aborts, which allows more transactions to commit in a best-effort HTM implementation.

Table IV shows results for experiments running the baseline and our optimized versions of *intruder* with the recommended large input set. Our simple optimization

Table IV
INTRUDER: KEY STATISTICS FOR BASELINE VERSUS OPTIMIZED CODE

intruder	Threads	Overall				TID1			
		Exec. Time	% Reduc	Speedup	Cycles/Tx	Abort Rate	Abort Rate	% Capac	% Confl
Base	1	15,5	-	1,00	1847	0,13	0,31	0,23	0,63
	2	8,5	-	1,82	1830	0,19	0,40	0,17	0,64
	4	4,9	-	3,16	1699	0,28	0,51	0,11	0,60
Opt	1	7,9	49	1,00	944	0,05	0,13	0,17	0,19
	2	4,4	48	1,80	959	0,08	0,17	0,10	0,37
	4	2,7	45	2,93	894	0,14	0,22	0,05	0,48

reduces execution time (*Exec. Time*, shown in seconds) by almost 50% in all thread configurations (*% Reduc* column), and it reduces the abort rate from 28 to 14% in runs with four threads. Transaction duration is halved, going from around 1800 to 900 cycles, on average. For the single-threaded configuration, we observe that memory-induced aborts (capacity+conflict) for the main transaction (TID1) decline from 86% to 36%. Note that the hardware implementation of RTM may interpret capacity aborts as being of conflict type when passing the abort status to the abort handler.

B. Case study II: vacation

vacation emulates a travel reservation system implemented as an online transaction processing system similar in design to SPECjbb2000 [14]. The database consists of four tables implemented as red-black trees. The customer tree associates customers with their list of reserved travel items, and the three remaining trees contain the reservable travel items, the associated price, and the available quantity. Client threads interact with the database in three types of sessions: reservations, cancellations, and updates. Each session is enclosed in a coarse-grain transaction to maintain the validity of the database. Reservation sessions query the item tables to search for the price and availability of a given item and then add reservations to the customer's list (decreasing the number of available instances appropriately).

In our experiments, we scale the database size down to 64K relations to reduce capacity aborts to reasonable levels, and we run only user sessions (-u 100) to reduce conflict-induced aborts as much as possible. We execute a total of 32M transactions. This workload allows us to better observe the effects of our optimizations compared to a baseline that is as RTM-friendly as possible.

Reducing read-set size and transaction duration. We find that the programming style used in *vacation* creates transactions of unnecessarily long duration due to redundant tree lookups. For example, in a reservation session, the benchmark searches the tree to check for an item's existence and then again looks up the item to find its price. Similarly, when item reservations are added to a customer reservation list, items queried in the previous step are looked up again to update their availability. With little extra effort, programmers could merge the queries for availability and price, both of which return references to the found item. The

reservation step can use this pointer to directly access items to be booked, obtaining the price and updating availability while avoiding redundant tree searches. Extra customer table lookups will thus be avoided.

The way elements are placed in data structures can have lengthen transactions and increase the size of their read-write set footprints. In vacation, the customer reservation list is kept sorted by type of item and id, and every new reservation needs traverses the list. The list is never searched for a specific reservation, though, and cancellation sessions do not require any specific ordering of the elements (since they simply iterate through the list to return each booking to the system). We simply change the code to always make insertions at the head of the list, avoiding traversals in the common case (reservation sessions).

Reducing aborts due to page faults. After the transaction has performed all queries in a reservation session, it allocates new memory to insert new reservations into the customer list. Accesses to this newly allocated memory can cause page faults that cause expensive misc3 aborts. If aware of this problem, programmers can improve RTM performance by triggering those page faults before the transaction. It suffices to modify the the STAMP thread-local memory allocator to touch new memory locations before returning..

Table V

VACATION: KEY STATISTICS FOR BASELINE VERSUS OPTIMIZED CODE.

vacation		Overall				Abort distribution				
	Threads	Exec. Time	% Reduc	Speedup	Cycles/Tx	Abort Rate	% Mem	% HLE-unfr	% Other	
Base	1	38,8	-		1,00	3360	0,11	0,21	0,76	0,03
	2	20,1	-		1,93	3284	0,14	0,31	0,64	0,05
	4	10,6	-		3,66	3095	0,21	0,43	0,51	0,06
Opt	1	29,4	24,23%		1,00	2725	0,02	0,89	0,01	0,10
	2	14,9	25,87%		1,97	2720	0,03	0,66	0,02	0,32
	4	7,9	25,47%		3,72	2711	0,07	0,13	0,01	0,86

Table V shows the results of our comparison between the baseline benchmark against an optimized version of vacation, which includes the changes described above, applied accumulatively. As can be seen in the table, with rather straightforward changes in the code a programmer can reduce execution time by approximately 25% for all thread configurations. Abort rates drop from 21 to 7% in 4-threaded runs, thanks to the elimination of virtually all aborts caused by page faults (which generally fall under the *HLE-unfriendly instruction* category or misc3). Transactions are also around 10% shorter. Note that after applying the optimizations, aborts of type misc5 (e.g., interrupts) become more important as we increase the number of threads.

VI. RELATED WORK

Hardware transactional memory systems must track memory updates within transactions and detect conflicts (read-write, write-read, or write-write conflicts across concurrent transactions or non-transactional writes to active locations

within transactions) at the time of access. The choice of where to buffer speculative memory modifications has microarchitectural ramifications, and commercial implementations naturally strive to minimize modifications to the cores and on-chip memory hierarchies on which they are based. For instance, Blue Gene/Q [8] tracks updates in the 32MB L2 cache, and the IBM System z [9] series and the cancelled Sun Rock [6] track updates in their store queues. Like the Haswell RTM implementation that we study here, the Vega Azul Java compute appliance [7] uses the L1 cache to record speculative writes. The size of transactions that can benefit from such hardware TM support depends on the capacity of the chosen buffering scheme. Like us, others have found that rewriting software to be more transaction-friendly improves hardware TM effectiveness [7]

Previous studies have investigated the characteristics of hardware transactional memory systems. Wang et al. [8] use the STAMP benchmarks to evaluate hardware transactional memory support on Blue Gene/Q, finding that the largest source of TM overhead is loss of cache locality from bypassing or flushing the L1 cache. Yoo et al. [15] use the STAMP benchmarks to compare Haswell RTM with the TL2 software transactional memory system [16], finding significant performance differences between TL2 and RTM. We performed a similar study and found that TinySTM consistently outperforms TL2, and thus we choose the former as our STM point of comparison. Our RTM scaling results for STAMP benchmark concur with those of Wang et al. [8].

Others have also studied power/performance tradeoffs. For instance, Gaona et al. [17] perform a simulation-based energy characterization study of two HTM systems: the LogTM-SE *Eager-Eager* system [2] and the Scalable TCC *Lazy-Lazy* system [18]. Ferri et al.[19] estimate the performance and energy implications of using TM in an embedded multiprocessor system-on-chips (MPSoCs), providing detailed energy distribution figures from their energy models.

In contrast to the work presented here, none of these studies analyzes energy expenditure for a commercial hardware implementation.

VII. CONCLUSIONS

The emergence of Hardware Transactional Memory (HTM) support in commonly available microprocessors is making it attractive for large numbers of programmers to use transactional memory. However, the best-effort nature of these early TM systems can create performance issues. HTM transactions are expected to be used with runtime systems that guarantee forward progress when transactions fail to commit. Carefully avoiding unnecessary serialization in such systems is essential for performance and energy efficiency. In this study, we measure the performance and energy costs of several schemes for shared memory synchronization. We show that HTM does not perform well across the board: Software Transactional Memory (STM)

implementations like tinySTM deliver highly competitive performance for certain workloads. We also characterize RTM performance along several dimensions. This should help programmers know what to expect when designing or porting applications to run on RTM. Our case studies highlight common optimizations that programmers should be mindful of to avoid unexpected performance losses.

REFERENCES

- [1] M. Herlihy and J. E. B. Moss, “Transactional memory: Architectural support for lock-free data structures,” in *Proceedings of the 20th International Symposium on Computer Architecture*, 1993, pp. 289–300.
- [2] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, “LogTM-SE: Decoupling hardware transactional memory from caches,” in *Proceedings of the 13th Symposium on High-Performance Computer Architecture*, 2007, pp. 261–272.
- [3] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, “Transactional memory coherence and consistency,” in *Proceedings of the 31st International Symposium on Computer Architecture*, 2004, pp. 102–113.
- [4] M. Lupon, G. Magklis, and A. González, “A dynamically adaptable hardware transactional memory,” in *Proceedings of the 43rd International Symposium on Microarchitecture*, 2010, pp. 27–38.
- [5] R. Titos-Gil, A. Negi, M. E. Acacio, J. M. Garcia, and P. Stenstrom, “Zebra : A data-centric, hybrid-policy hardware transactional memory design,” in *Proceedings of the 25th International Conference of Supercomputing*, 2011, pp. 53–62.
- [6] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, “Early experience with a commercial hardware transactional memory implementation,” *SIGPLAN Not.*, vol. 44, no. 3, pp. 157–168, Mar. 2009.
- [7] C. Click, “Azul’s experiences with hardware transactional memory,” 2009, http://sss.cs.purdue.edu/projects/tm/tmw2010/talks/Click-2010_TMW.pdf.
- [8] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael, “Evaluation of Blue Gene/Q hardware support for transactional memories,” in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, ser. PACT ’12. New York, NY, USA: ACM, 2012, pp. 127–136.
- [9] C. Jacobi, T. Slegel, and D. Greiner, “Transactional memory architecture and implementation for IBM System z,” in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 25–36.
- [10] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, “STAMP: Stanford transactional applications for multi-processing,” in *Proceedings of the IEEE International Symposium on Workload Characterization*, 2008, pp. 35–46.
- [11] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun, “Eigenbench: A simple exploration tool for orthogonal TM characteristics,” in *Proceedings of the IEEE International Symposium on Workload Characterization*, ser. IISWC ’10. IEEE Computer Society, 2010, pp. 1–11.
- [12] P. Felber, C. Fetzer, P. Marlier, and T. Riegel, “Time-based software transactional memory,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 12, pp. 1793–1807, 2010.
- [13] Desktop 4th Generation Intel Core™ Processor Family Specification Update, Intel, August 2013.
- [14] S. P. E. Corporation, “SPECjbb2000 benchmark,” 2000. [Online]. Available: <http://www.spec.org/osg/jbb2000/>
- [15] R. Yoo, C. Hughes, K. Lai, and R. Rajwar, “Performance evaluation of Intel transactional synchronization extensions for high performance computing,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2013.
- [16] D. Dice, O. Shalev, and N. Shavit, “Transactional Locking II,” in *Proceedings of the 19th International Symposium on Distributed Computing*, 2006.
- [17] E. Gaona-Ramirez, R. Titos-Gil, J. Fernandez, and M. Acacio, “Characterizing energy consumption in hardware transactional memory systems,” in *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*, 2010, pp. 9–16.
- [18] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun, “A scalable, non-blocking approach to transactional memory,” in *Proceedings of the 13th Symposium on High-Performance Computer Architecture*, 2007, pp. 97–108.
- [19] C. Ferri, R. Bahar, A. Marongiu, L. Benini, M. Herlihy, B. Lipton, and T. Moreshet, “SoC-TM: Integrated HW/SW support for transactional memory programming on embedded MPSoCs,” in *2011 Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis*, 2011, pp. 39–48.

DATA PARALLEL ALGORITHMS

Parallel computers with tens of thousands of processors are typically programmed in a data parallel style, as opposed to the control parallel style used in multiprocessing. The success of data parallel algorithms—even on problems that at first glance seem inherently serial—suggests that this style of programming has much wider applicability than was previously thought.

W. DANIEL HILLIS and GUY L. STEELE, JR.

In this article we describe a series of algorithms appropriate for fine-grained parallel computers with general communications. We call these algorithms *data parallel* algorithms because their parallelism comes from simultaneous operations across large sets of data, rather than from multiple threads of control. The intent is not so much to present new algorithms (most have been described earlier in other contexts), but rather to demonstrate a style of programming that is appropriate for a machine with tens of thousands or even millions of processors. The algorithms described all use $O(N)$ processors to solve problems of size N , typically in $O(\log N)$ time. Some of the examples solve problems that at first sight seem inherently serial, such as parsing strings and finding the end of a linked list. In many cases, we include actual run times measured on the 65,536-processor Connection Machine® system. A key feature of this hardware is a general-purpose communications network connecting the processors that frees the programmer from detailed concerns of the mapping between data and hardware.

MODEL OF THE MACHINE

Our model of a fine-grained parallel machine with general-purpose communication is based on the

Connection Machine system [14]. Current Connection Machine systems have either 16,384 or 65,536 processors, each with 4,096 bits of memory. There is nothing special about these particular numbers, other than being powers of two, but they are indicative of the scale of the machine. (The model is more sensible with thousands rather than tens of processors.) The system has two parts: a front-end computer of the usual von Neumann style, and an array of Connection Machine processors. Each processor in the array has a small amount of local memory, and to the front end, the processor array looks like a memory. A typical front-end processor is a VAX® or a Symbolics 3600®.

The processor array is connected to the memory bus of the front end so that the local processor memories can be random accessed directly by the front end, one word at a time, just as if it were any other memory. This section of the memory on the front end, however, is "smart" in the sense that the front end can issue special commands that cause many parts of the memory to be operated upon simultaneously, or cause data to move around within the memory. The processor array therefore effectively extends the instruction set of the front-end processor to include instructions that operate on large

Connection Machine is a registered trademark of Thinking Machines Corporation.

© 1986 ACM 0001-0782/86/1200-1170 75¢

VAX is a trademark of Digital Equipment Corporation.

Symbolics 3600 is a trademark of Symbolics, Inc.

amounts of data simultaneously. In this way, the processor array serves a function similar to a floating-point accelerator unit, except that it accelerates general parallel computation and not just floating-point arithmetic.

The control structure of a program running on a Connection Machine system is executed by the front end in the usual way. An important practical benefit of this approach is that the program is developed and executed within the already-familiar programming environment of the front end. The program can perform computations in the usual serial way on the front end and also issue commands to the processor array.

The processor array executes commands in SIMD fashion. There is a single instruction stream coming from the front end; these instructions act on multiple data items, on the order of one (or a few) per processor. Most instructions are executed conditionally: That is, each processor has state bits that determine which instructions the processor will execute. A processor whose state bit is set is said to be *selected*. The state bit is called the *context flag* because the set of selected processors is often referred to as the *context* within which instructions are executed. For example, the front end might arrange for all odd-numbered processors to have their context flags set, and even-numbered processors to have their context flags cleared; issuing an **ADD** instruction would then cause each of the selected processors (the odd-numbered ones) to add one word of local memory into another word. The deselected (even-numbered) processors would do nothing, and their local memories would remain unchanged.

Contexts may be saved in memory and later restored, with each processor saving or restoring its own bit in parallel with the others. There are a few instructions that are unconditional: They are executed by every processor regardless of the value of the context flag. Such instructions are required for saving and restoring contexts.

A context, or a set of values for all the context flags, represents a set: namely, a set of selected processors. Forming the intersection, union, or complement of such sets is simple and fast; it requires only a one-bit logical **AND**, **OR**, or **NOT** operation issued to the processors. Locally viewed, each processor performs just one logical operation on one or two single-bit operands; viewed globally, an operation on sets is performed. (On the current Connection Machine hardware, such an operation takes about a microsecond.)

The processors can individually perform all the usual operations on logical, integer, and floating-

point operands: add, subtract, multiply, divide, compare, max, min, not, and, or, exclusive or, shift, square root, and so on. In addition, single values computed in the front end can be broadcast from the front end to all processors at once (essentially by including them as immediate data in the instruction stream).

A number of other computing systems have been constructed with the characteristics we have already described, namely, a large number of parallel processors, each of which has local memory and the ability to execute instructions of more or less the usual sort, as broadcast from a master controller. These include ILLIAC IV [8], the Goodyear MPP [3], the Non-Von [23]; and the ICL DAP [12]; among others [13]. There are two additional characteristics of the Connection Machine programming model, however, which distinguish it from these other systems: *general, pointer-based communication*, and *virtual processors*.

Previous parallel computing systems of this fine-grained SIMD style have restricted interprocessor communication to certain patterns wired into the hardware; typically this pattern is a two-dimensional rectangular grid, or a tree. The Connection Machine model allows any processor to communicate directly with any other processor in unit time, while other processors also communicate concurrently. Communication is implemented via a **SEND** instruction.

Within each processor, the **SEND** instruction takes two operands: One addresses—within the processor—the field that contains the data to be sent; the other addresses a processor pointer (i.e., the number of the processor to which the datum is to be sent and the destination field within that processor, into which the data will be placed). The communications system is very much like a postal system, where you can send a message to anyone else directly, provided you know the address, and where many letters can be routed at the same time. The **SEND** instruction can also be viewed as a parallel “store indirect” instruction that allows each processor to store anywhere in the entire memory, not just in its own local memory.

The **SEND** instruction can also take one additional operand that specifies what happens if two or more messages are sent to the same destination. The options are to deliver to the destination the sum, maximum, minimum, bitwise **AND**, or bitwise **OR** of the messages; to deliver one message and discard all others; or to produce an error.

From a global point of view, the **SEND** instruction performs something like an arbitrary permutation on an array of items, although it is actually more

general than a permutation because more than one item may be sent to the same destination. The pattern is not wired into the hardware, but is encoded as an array of pointers, and is completely under software control; the same encoding, once constructed, can be used over and over again to transfer data repeatedly in the same pattern. To implement a regular communication pattern, such as a two-dimensional grid, the pointers are typically computed when needed rather than stored.

The Connection Machine programming model is carefully abstracted from the details of the hardware that supports it, and, in particular, the number and size of its hardware processors. Programs are described in terms of virtual processors. In actual implementations, hardware processors are multiplexed as necessary to support this abstraction; indeed, the abstraction is supported at a very low level by a microcoded controller interposed between the front end and the processor array, so that the front end always deals in virtual processors.

The benefits of the virtual processor abstraction are twofold. The first is that the same program can be run unchanged on different sizes of the Connection Machine system, notwithstanding the linear trade-off between the number of hardware processors and execution time. For example, a program that requires 2^{16} virtual processors can run at top speed on a system with the same number of hardware processors, but it can also be executed on one-fourth that amount of hardware (2^{14} processors) at one-fourth the speed, provided it can fit into one-fourth the amount of memory as well.

The second benefit is that for many purposes the number of processors may be regarded as expandable rather than fixed, so that it becomes natural to write programs using the Lisp, Pascal, or C style of storage allocation rather than the Fortran style. By this we mean that there is a procedure one can call to allocate a "fresh" processor as if from thin air while the program is running. In Fortran, all storage is preallocated before program execution begins, whereas Lisp has the **cons** operation to allocate a new list cell (as well as other operations for constructing other objects); Pascal has the **new** operation; and C has the **malloc** function. In each case, a new object is allocated and a pointer to this new object is returned. Of course, in the underlying implementation, the address space (physical or virtual) is actually a fixed resource pool from which all such requests are satisfied, but the point is that the language supports the abstraction of newly created storage. In the Connection Machine model, one may similarly allocate fresh storage using the operation

processor-cons; the difference is that the newly allocated storage comes with its own processor attached.

In the ensuing discussion, we shall assume that the size and number of processors are sufficient to allocate one processor for each data element in the problem being solved. This allows us to adopt a model of the machine in which the following are counted as unit-time operations:

- any conventional word-at-a-time operation;
- any such operation applied to all the data elements concurrently, or to some selected subset;
- any communications step that involves the broadcast of information to all data elements;
- any communications step that involves no more than a single message transmission from each data element.

For purposes of analysis, it is also often useful to treat **processor-cons** as a unit-time operation, although it may be implemented in terms of more primitive operations, as described in more detail on page 1176.

EXAMPLES OF PARALLEL PROGRAMMING

To show some of the possibilities of data-parallel programming, we present here several algorithms currently in use on the Connection Machine system. Most of these algorithms are not new: Some of the ideas represented here appear in the languages APL [11, 15] and FP [1], while others are based on algorithms designed for other parallel machines, in particular, the Ultracomputer [22], and still others have been developed by our coworkers on the Connection Machine [4, 5, 7, 9, 10, 19].

Beginning with some very simple examples to familiarize the reader with the model and the notation, we then proceed to more elaborate and less obvious examples.

Sum of an Array of Numbers

The sum of n numbers can be computed in time $O(\log n)$ by organizing the addends at the leaves of a binary tree and performing the sums at each level of the tree in parallel. There are several ways of organizing an array into a binary tree. Figure 1 illustrates one such method on an array of 16 elements named x_0 through x_{15} . In this algorithm, for purposes of simplicity, the number of elements to be summed is assumed to be an integral power of two. There are as many processors as elements, and the statement **for all k in parallel do s od** causes all processors to execute the same statement *s* in synchrony, but the variable *k* has a different value for each processor,

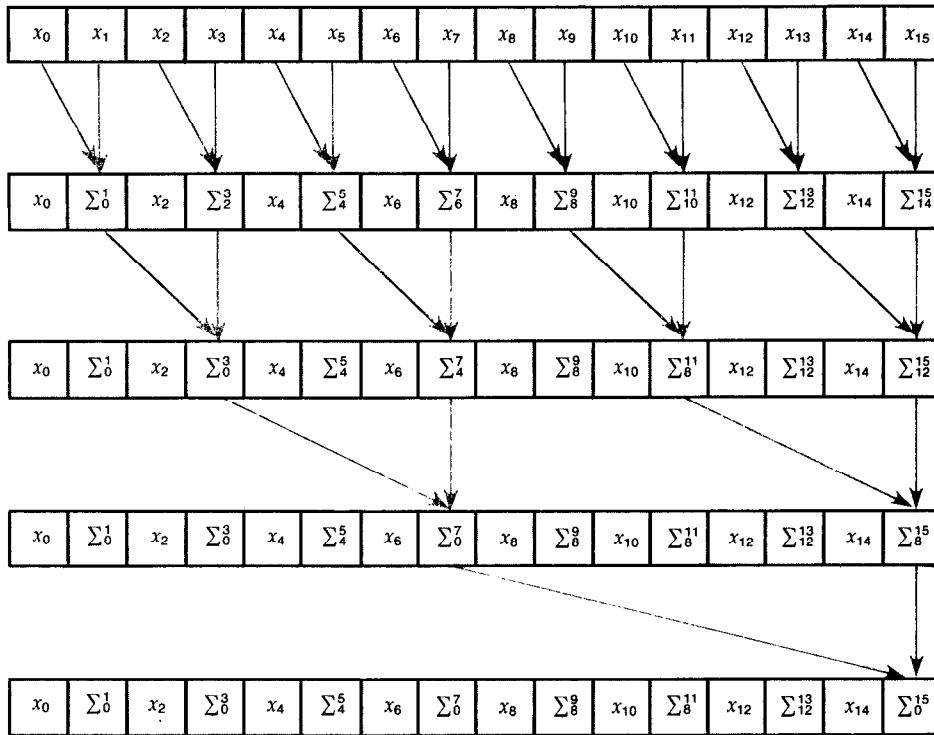


FIGURE 1. Computing the Sum of an Array of 16 Elements

namely, the index of that processor within the array.

```

for  $j := 1$  to  $\log_2 n$  do
  for all  $k$  in parallel do
    if  $((k + 1) \bmod 2^j) = 0$  then
       $x[k] := x[k - 2^{j-1}] + x[k]$ 
    fi
  od
od

```

At the end of the process, x_{n-1} contains the sum of the n elements. On the Connection Machine, an optimized version of this algorithm for 65,536 elements takes about 200 microseconds.

All Partial Sums of an Array

A frequently useful operation is computing all partial sums of an array of numbers. In APL, this computation is called a plus-scan; in other contexts, it is called the "sum-prefix" operation because it computes sums over all prefixes of the array. For example, if you put into an array your initial checkbook balance, followed by the amounts of the checks you have written as negative numbers and deposits as positive numbers, then computing the partial sums produces all the intermediate and final balances.

It might seem that computing such partial sums is an inherently serial process, because one must add up the first k elements before adding in element

$k + 1$. Indeed, with only one processor, one might as well do it that way, but with many processors one can do better, essentially because in $\log n$ time with n processors one can do $n \log n$ individual additions; serialization is avoided by performing logically redundant additions.

Looking again at the simple summation algorithm given on the facing page, we see that most of the processors are idle most of the time: During iteration j , only $n/2^j$ processors are active, and, indeed, half of the processors are never used. However, by putting the idle processors to good use by allowing more processors to operate, the summation algorithm can compute all partial sums of the array in the same amount of time it took to compute the single sum. In defining Σ_j^k to mean $\sum_{i=j}^k x_i$, note that $\Sigma_j^k + \Sigma_{k+1}^m = \Sigma_j^m$. The partial-sums algorithm replaces each x_k by Σ_0^k ; that is, the sum of all elements preceding and including x_k . In Figure 2 (on the following page), this process is illustrated for an array of 16 elements.

```

for  $j := 1$  to  $\log_2 n$  do
  for all  $k$  in parallel do
    if  $k \geq 2^j$  then
       $x[k] := x[k - 2^{j-1}] + x[k]$ 
    fi
  od
od

```

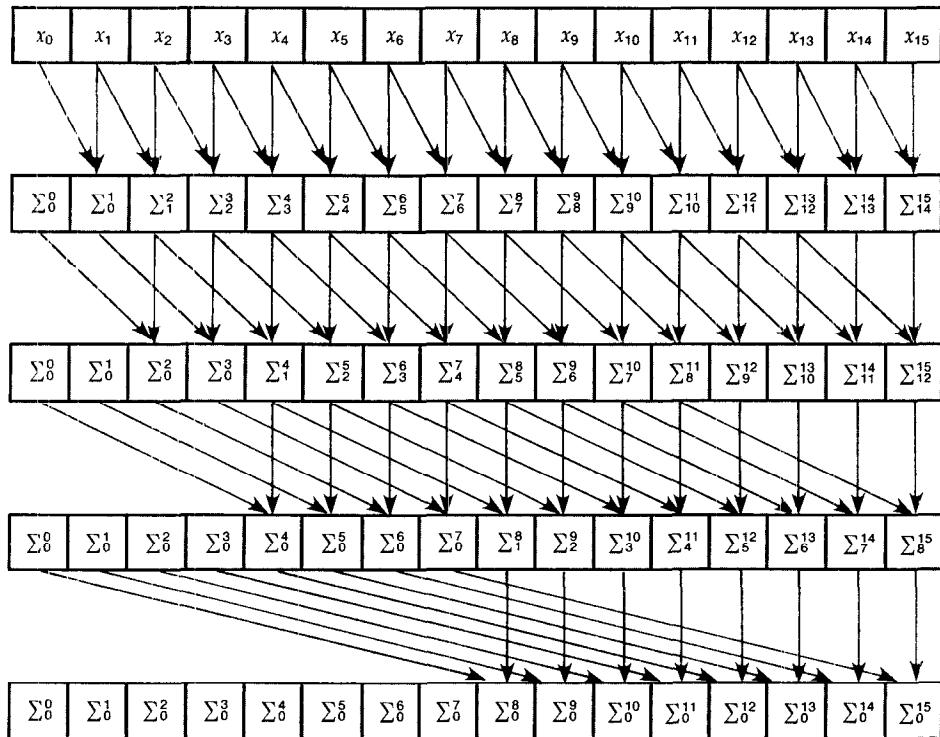


FIGURE 2. Computing Partial Sums of an Array of 16 Elements

The only difference between this algorithm and the earlier one is the test in the **if** statement in the partial-sums algorithm that determines whether a processor will perform the assignment. This algorithm keeps more processors active: During step j , $n - 2^{j-1}$ processors are in use; after step j , element number k has become \sum_a^k where $a = \max(0, k - 2^j + 1)$.

This technique can be generalized from summation to any associative combining operation. Some obvious choices are product, maximum, minimum, and logical **AND**, **OR**, and **EXCLUSIVE OR**. Some programming languages provide such reduction and parallel-prefix operations on arrays as primitives. The current proposal for Fortran 8x, for example, provides reduction operations called **SUM**, **PRODUCT**, **MAXVAL**, **MINVAL**, **ANY**, and **ALL**. The corresponding reduction functions in APL are $+/$, $\times/$, $\lceil /$, $\lfloor /$, $\vee/$, and $\wedge/$; APL also provides other reduction operations and all the corresponding scan (prefix) operations. The combining functions for all these operations happen to be commutative as well, but the algorithm does not depend on commutativity. This was no accident; we took care to write

$$x[k] := x[k - 2^{j-1}] + x[k]$$

instead of the more usual

$$x[k] := x[k] + x[k - 2^{j-1}]$$

precisely in order to preserve the correctness of the algorithm when $+$ is replaced by an associative but noncommutative operator. Under "Parsing a Regular Language" (facing page), we discuss the use of parallel-prefix computations with a noncommutative combining function for a nonnumerical application, specifically, the division of a character string into tokens. Another associative noncommutative operator of particular practical importance is matrix multiplication. We have found this technique useful in multiplying long chains of matrices.

Counting and Enumerating Active Processors

After some subset of the processors has been selected according to some condition (i.e., by using the result of a test to set the context flags), two operations are frequently useful: determining how many processors are active, and assigning a distinct integer to each processor. We call the first operation **count** and the second **enumerate**: Both are easily implemented in terms of summation and sum-prefix.

To **count** the active processors, we have every processor unconditionally examine its context flag and compute the integer 1 if the flag is set and 0 if it is clear. (Remember that an unconditional operation is performed by every processor regardless of whether or not its context flag is set.) We then perform an unconditional summation of these integer values.

To **enumerate** the active processors, we have every processor unconditionally compute a 1 or 0 in the same manner, but then we perform an unconditional sum-prefix calculation with the result that every processor receives a count of the number of active processors that precede it (including itself) in the ordering. We then revert to conditional operation; in effect, the selected processors have received distinct integers, and values computed for the deselected processors are henceforth simply ignored. Finally, it is technically convenient to have every selected processor subtract one from its value, so that the n selected processors will receive values from 0 to $n - 1$ rather than from 1 to n .

These operations each take about 200 microseconds on a Connection Machine of 65,536 elements. Because these operations are so fast, programming models of the Connection Machine have been suggested that treat counting and enumeration as unit-time operations [6, 7].

Radix Sort

Sorting is a communications-intensive operation. In parallel computers with fixed patterns of communication, the pattern of physical connections usually suggests the use of a particular sorting algorithm. For example, Batcher's bitonic sort [2] fits nicely on processors connected in a perfect shuffle pattern, bubble sorts [16] work well on one-dimensionally connected processing, and so on. In the Connection Machine model, the ability to access data in parallel in any pattern often eliminates the need to sort data. When it is necessary to sort, however, the generality of communications provides an embarrassment of riches.

Upon implementing several sorting algorithms on the Connection Machine and comparing them for speed, we found that, for the current hardware implementation, Batcher's method has good performance for large sort keys, whereas for small sort keys a version of radix sort is usually faster. (The break-even point is for keys about 25 to 32 bits in length. With either algorithm, sorting 65,536 32-bit numbers on the Connection Machine takes about 30 milliseconds.)

To illustrate the use of **count** and **enumerate**, we present here the radix sort algorithm. In the interest of simplicity, we will assume that all processors (n) are active, that sort keys are unsigned integers, and that *maxint* is the value of the largest representable key value.

```

for  $j := 1$  to  $1 + \lceil \log_2 maxint \rceil$  do
  for all  $k$  in parallel do
    if  $(x[k] \bmod 2^j) < 2^{j-1}$  then
      comment The bit with weight  $2^{j-1}$  is zero.
      tnemmoc
       $y[k] := \text{enumerate}$ 
       $c := \text{count}$ 
    fi
    if  $(x[k] \bmod 2^j) \geq 2^{j-1}$  then
      comment The bit with weight  $2^{j-1}$  is one.
      tnemmoc
       $y[k] := \text{enumerate} + c$ 
    fi
     $x[y[k]] := x[k]$ 
  od
od

```

At this point, an explanation of a fine point concerning the intended semantics of our algorithmic notation is in order. An **if** statement that is executed for all processors is always assumed to execute its **then** part, even if no processors are selected, because some front-end computations (such as **count**) might be included. When the **then** part is executed, the active processors are those previously active processors that computed **true** for the condition.

The radix sort requires a logarithmic number of passes, where each pass essentially examines one bit of each key. All keys that have a 0 in that bit are counted (call the count c) and then enumerated in order to assign them distinct integers y_k ranging from 0 to $c - 1$. All keys that have a 1 in that bit are then enumerated, and c is added to the result, thereby assigning these keys distinct integers y_k ranging from c to $n - 1$. The values y_k are then used to permute the keys so that all keys with a 0 bit precede all keys with a 1 bit. (This is the step that takes particular advantage of general communication.) This permutation is stable: The order of any two keys that both have 0 bits or both 1 bits is preserved. This stability property is important because the keys are sorted by least significant bit first and most significant bit last.

Parsing a Regular Language

To illustrate the use of a parallel-prefix computation in a nonnumerical application, consider the problem of parsing a regular language. For a concrete practical instance, let us consider the problem of breaking

up a long string of characters into tokens, which is usually the first thing a compiler does when processing a program. A string of characters such as

```
if x <=n then print("x = ", x);
```

must be broken up into the following tokens, with redundant white space eliminated:

```
if [x] [<=] [n] then print [ ( ] ["x = " , [x] ) [ ; ]
```

This process is sometimes called *lexing a string*.

Any regular language of this type can be parsed by a finite-state automaton that begins in a certain state and makes a transition from one state to another (possibly the same one) as each character is read. Such an automaton can be represented as a two-dimensional array that maps the old state and the character read to the new state. Some of the states correspond to the start of a token; if the automaton is in one of those states just after reading a character, then that character is the first character of a token. Some characters may not be part of any token; White-space characters, for example, are typically not part of a token unless they occur within a string; such delimiting characters may also be identified by the automaton state just after the character is read. To divide a string up into tokens, then, means merely determining the state of the automaton after each character has been processed.

Table I shows the automaton array for a simple language in which a token may be one of three things: a sequence of alphabetic characters, a string surrounded by double quotes (where an embedded double quote is represented by two consecutive double quotes), or any of +, -, *, =, <, >, <=, and >=. Spaces and newlines delimit tokens, but are not part of any token except quoted strings. The automaton has nine states: N is the initial state; A is the start of an alphabetic token; Z is the continuation of an alphabetic token; * is a single-special-character token; < is a < or > character; = is an = that follows a < or > character (an = that does not follow < or > will produce state *); Q is the double quote that starts a string; S is a character within a string; and E is the double quote that ends a string, or the first of two that indicate an embedded double quote. The states A , *, <, and Q indicate that the character just read is the first character of a token.

Although, like the computation of partial sums, this may appear at first glance to be an inherently serial process, it too can be put into the form of a parallel-prefix computation. Rather than regarding

the lexing automaton as a monolithic process, let us regard the individual characters of the string as unary functions that map an automaton state onto another state. By indicating the application of the character Y to state N as NY , we may then write $NY = A$. By extension, it is also possible to regard a string as a function that maps a state p to another state q ; q is the state you end up in if you start the automaton in state p and then let the automaton read the entire string one character at a time. The result of applying the string Y^+ to the state Z may be written as $ZY^+ = ((ZY)^+)^+ = (Z^+)^+ = Q^+ = S$. It is not too hard to see that the function corresponding to a string is simply the composition of the functions for the individual characters.

A function from a state to a state can be represented as a one-dimensional array indexed by states whose elements are states. The columns of the array in Table I are in fact exactly such representations for the functions for individual characters. Composing the columns for two characters or strings to produce a new column for the concatenation of the strings is fairly straightforward: You simply replace every entry of one column with the result of using that entry to index into the other column.

Since this composition operation is associative, we may compute the automaton state after every character in a string as follows:

1. Replace every character in the string with the array representation of its state-to-state function.
2. Perform a parallel-prefix operation. The combining function is the composition of arrays as described above. The net effect is that, after this step, every character c of the original string has been replaced by an array representing the state-to-state function for that prefix of the original string that ends at (and includes) c .
3. Use the initial automaton state (N in our example) to index into all these arrays. Now every character has been replaced by the state the automaton would have after that character.

If we implement this algorithm on a Connection Machine system and allot one processor per character, the first and third steps will take constant time, and the second step will take time logarithmic in the length of the string. Naturally, this algorithm performs much more computation per character than the straightforward serial algorithm using the two-dimensional array, but, for sufficiently large amounts of text, the parallel algorithm will be faster because its time complexity is logarithmic instead of linear. An implementation of this algorithm in Connection Machine Lisp can be found in [24].

PARALLEL PROCESSING OF POINTERS

Processor-cons

To illustrate pointer manipulation algorithms, we will consider the implementation of the **processor-cons** primitive, which allows a set of processors to establish pointers to a set of new processors allocated from free storage. In a serial computer, the equivalent problem is usually solved by keeping the free storage in an ordered list and allocating new storage from the beginning of the list. In the Connection Machine, this would not suffice since we wish to allocate many elements concurrently. Instead, the **processor-cons** primitive is implemented in terms of **enumerate** by using a rendezvous technique: Two sets of m processors are brought into one-to-one communication by using the processors numbered 0 through $m - 1$ as rendezvous points.

Assuming that every processor has a Boolean variable called *free*, $free_k$ becomes **true** if processor k is available for allocation and **false** otherwise. Every selected processor is to receive, in a variable called *new-processor*, the number of a distinct free processor. Free processors that are so allocated have their *free* bits reset to **false** in the process. If there are fewer free processors than selected processors, then as many requests are satisfied as possible, and some selected processors will have their *new-processor* variables set to *null* instead of the number of a free processor, as shown below.

```

for all k in parallel do
  required := count
  unconditionally
    if free[k] then
      available := count
      free-processor[k] := enumerate
      if free-processor[k] < required then
        free[k] := false
      fi
      rendezvous[free-processor[k]] := k
      requestor[k] := enumerate
      fi
  yllanoitidnocnu
  if requestor[k] < available then
    new-processor := rendezvous[requestor[k]]
  else
    new-processor := null
  fi
od
```

In this way, the total number of processors is managed as a finite resource, but with an interface that presents the illusion of creating new processors on demand. (Of course, we have not indicated how

TABLE I. A Finite-State Automaton for Recognizing Tokens

Old State	Character Read													New line
	A	B	...	Y	Z	+	-	*	<	>	=	"	Space	
N	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
A	Z	Z	...	Z	Z	*	*	*	<	<	*	Q	N	N
Z	Z	Z	...	Z	Z	*	*	*	<	<	*	Q	N	N
*	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
<	A	A	...	A	A	*	*	*	<	<	=	Q	N	N
=	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
Q	S	S	...	S	S	S	S	S	S	S	E	S	S	
S	S	S	...	S	S	S	S	S	S	S	E	S	S	
E	E	E	...	E	E	*	*	*	<	<	*	S	N	N

processors are returned to the pool of free processors. Some technique such as reference counting or garbage collection must also be designed and coded.) Other algorithms for **processor-cons** are described in [9, 10].

Parallel Combinator Reduction

A topic of much current interest in the area of functional programming is parallel combinator reduction [25]. It is also particularly interesting in this context because it shows how data parallel algorithms can be used to simulate control parallelism, or, equivalently, how SIMD machines with general communication can simulate MIMD machines.

Combinators are a way of encoding an applicative language. Their appeal lies in the fact that a program can be executed simply by performing successive local transformations on a tree structure, moreover, it is possible to perform many independent transformations simultaneously in the same tree. A combinator tree is made up of pairs, where each of the *left* and *right* components of a pair may point to another pair or else be an atom, the name of a combinator. Standard names for combinators include S, K, I, B, and C. Figure 3 (next page) shows one possible set of four transformations that suffices for program interpretation. When a subtree is transformed, the root pair of the subtree is used as the root pair of the result (by altering its components), but it is not permissible to alter any of the other pairs involved; therefore, the transformation involving the S combinator requires the allocation of fresh pairs. For our purposes, we ignore the semantics of the combinators and simply observe that such graph transformations can easily be carried out in parallel by a Connection Machine system by letting each processor contain one pair, and using **processor-cons** to allocate new pair-processors as needed.

```

while want or need to reduce some more do
  for all k in parallel do
    lf := left[k]
    if pair(lf) then
      if left[lf] = 'K' then
        left[k] := 'T'
      fi
      if left[lf] = 'I' then
        left[k] := right[lf]
      fi
    if pair(left[lf]) and left[left[lf]] = 'S' then
      p := processor-cons
      q := processor-cons
      if p ≠ null and q ≠ null then
        left[p] := right[left[lf]]
        right[p] := right[lf]
        left[q] := right[left[lf]]
        right[q] := right[k]
        left[k] := p
        right[k] := q
      fi
    fi
    rt := right[k]
    if pair(rt) and left[rt] = 'I' then
      right[k] := right[rt]
    fi
  od
  possibly perform garbage collection
od

```

It is easy to write such parallel code as a Connection Machine program. However, there are some difficult resource-management issues that have been glossed over, such as when a garbage collection should occur; whether, at any given time, one should prefer to reduce S combinators or K combinators; and, given that S combinators are to be reduced, which ones should be given preference. (The issues are that there may be more of one kind of combinator than the other at any instant. One idea is to process whichever kind is preponderant, but S combinators consume pairs and K combinators may release pairs, so the best processing strategy may need to take the number of free processors available for allocation into account. Furthermore, if there are not enough free processors to satisfy all outstanding S combinators at once, then the computation may diverge—even if normal-order serial reduction would converge—if preference is consistently given to the wrong ones.)

Finding the End of a Linked List

When we first began to work with pointer structures in the Connection Machine model, we believed that balanced trees would be important because informa-

tion can be propagated from the root of a tree to its leaves—or from the leaves to the root—by parallel methods that take time logarithmic in the number of leaves. This was correct. However, our intuition also told us that linear linked lists would be useless. We could understand how to process an array in logarithmic time, because one can use address arithmetic to compute the number of any processor and then communicate with it directly, but it seemed to us that a linked list must be processed serially because in general one cannot compute the address of the ninth processor in a list without following all eight of the intervening pointers.

As is so often true in computer science, intuition was misleading: Essentially, we overlooked the power of having many processors working on the problem at once. It is true that one must follow all eight pointers, but by using many processors one can still achieve this in time logarithmic in the number of pointers to be traversed. Although we found this algorithm surprising, it had been discovered in other contexts several times before (e.g., see chapter 9 of [20]).

As a simple example, consider finding the last cell of a linearly linked list. Imagine each cell to have a *next* pointer that points to the next cell in the list,

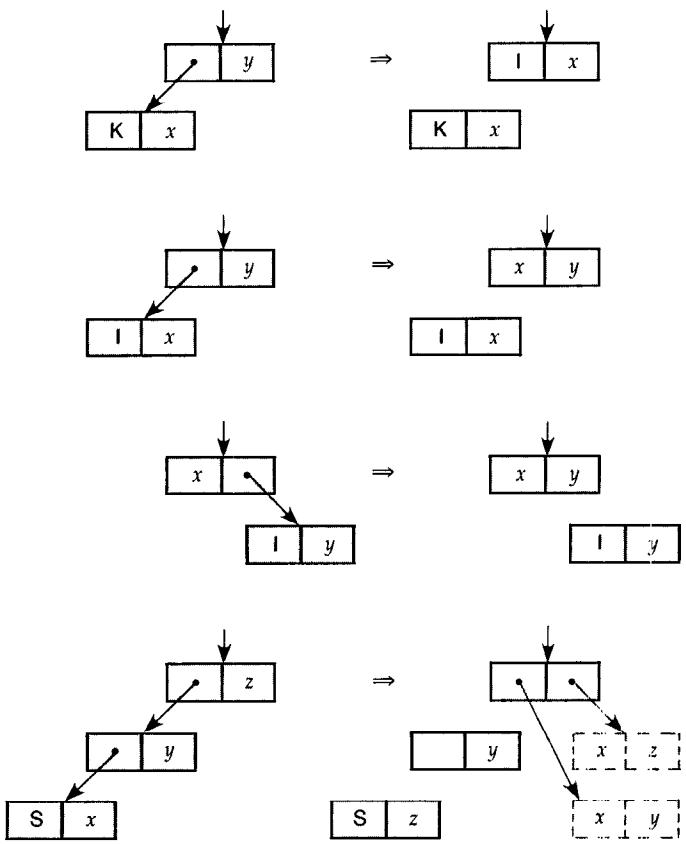


FIGURE 3. Patterns of Combinator Reduction

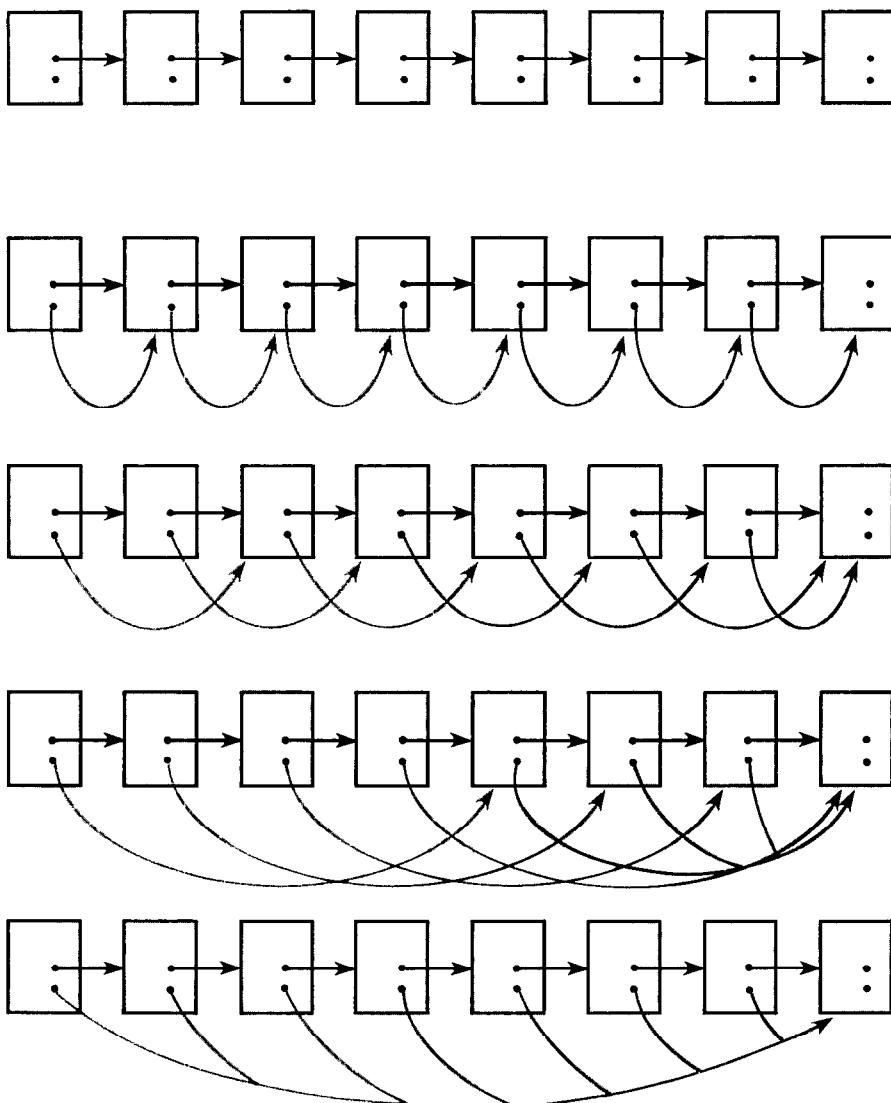


FIGURE 4. Finding the End of a Serially Linked List

while the last cell has the special value *null* in its *next* component. To accommodate other information as well, we will assume that in each cell there is another pointer component called *chum* that may be used for temporary purposes.

The basic idea is as follows: Each processor sets its *chum* component equal to a copy of its *next* component, so *chum* points to the next cell in the list. Each processor then repeatedly replaces its *chum* by its *chum's chum*. However, if its *chum* is *null*, then it remains *null*.) Initially, the *chum* of a processor is the next cell in the list; after the first step, its *chum* is the second cell following; after the second step, its *chum* is the fourth cell following; after the third step, its *chum* is the eighth cell following; and so on.

To ensure that the first cell of a list finds the last cell of a list, we carry out this procedure with the modification that a processor does not change its

chum if its *chum's chum* is *null*, as shown below. The process is illustrated graphically in Figure 4.

```
for all k in parallel do
  chum[k] := next[k]
  while chum[k] ≠ null and chum[chum[k]] ≠ null do
    chum[k] := chum[chum[k]]
  od
```

The meaning of the **while** loop is that at each iteration a processor becomes deselected if it computes **false** for the test expression; the loop terminates when all processors have become deselected (whereupon the original context, as of the beginning of the loop, is restored). When this process terminates, *every* cell of the list except the last will have the last cell as its *chum*. If there are many lists in the machine, they can all be processed simultaneously,

and the total processing time will be proportional to the logarithm of the length of the longest such list.

All Partial Sums of a Linked List

The partial sums of a linked list may be computed by the same technique

for all k in parallel do

```
chum[k] := next[k]
while chum[k] ≠ null do
    value[chum[k]] := value[k] + value[chum[k]]
    chum[k] := chum[chum[k]]
od
od
```

as illustrated in Figure 5. Comparing Figure 5 to Figure 2 (computing partial sums), we see that the

same patterns of pointers among elements are constructed on the fly by using address arithmetic in the case of an array and by following pointer chains in the case of a linked list. An advantage of the linked-list representation is that it can simultaneously process many linked lists of different lengths without any change to the code.

Matching Up Elements of Two Linked Lists

An even more exotic effect is obtained by the following algorithm, which matches up corresponding elements in two linked lists. If we will call corresponding elements of a list "friends", this algorithm assigns to each list cell a pointer to its friend in the other list; of course, the result is obtained in logarithmic time.

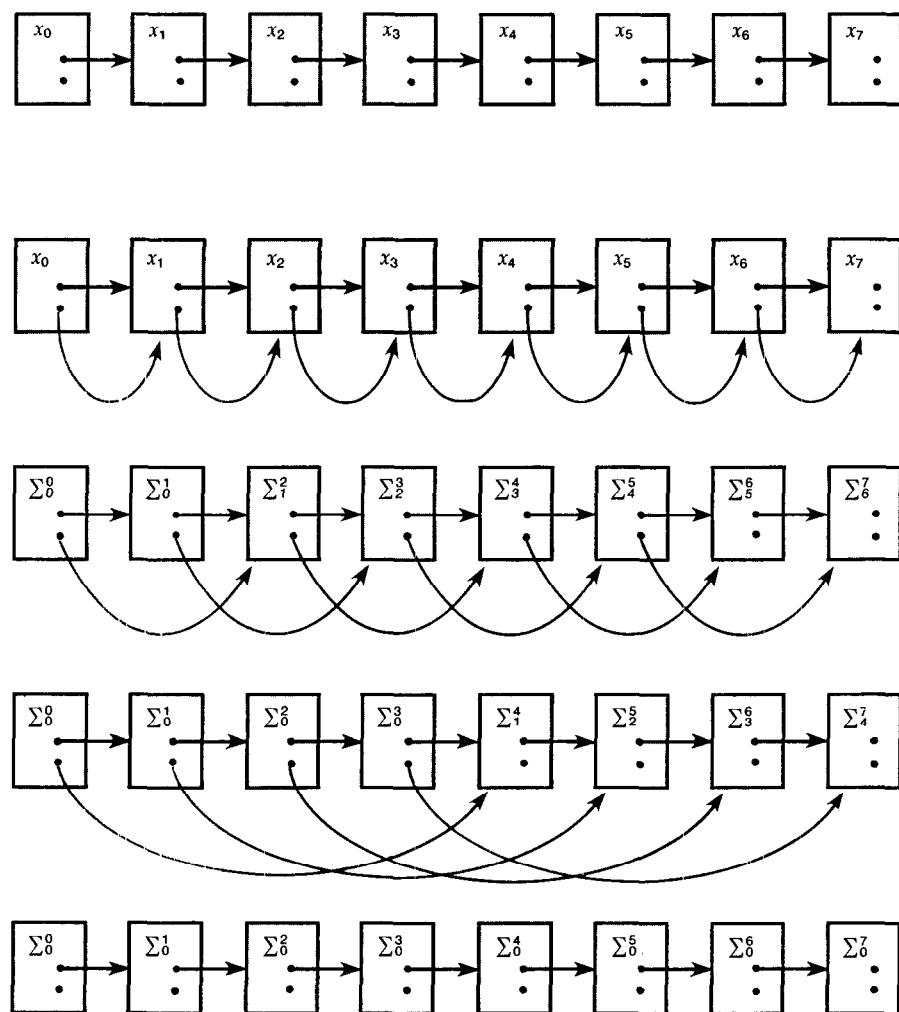


FIGURE 5. Computing Prefix Sums of a Serially Linked List

```

for all k in parallel do
  friend[k] := null
od
friend[list1] := list2
friend[list2] := list1
for all k in parallel
  chum[k] := next[k]
  while chum[k] ≠ null do
    if friend[k] ≠ null then
      friend[chum[k]] := chum[friend[k]]
      chum[k] := chum[chum[k]]
    fi
  od

```

The first part of the above algorithm is initialization: The component named *friend* is initialized to *null* in every cell; then the first cells of the two lists are introduced, so that they become *friends*. The second part plays the familiar logarithmic *chums* game, but at every iteration, a cell that has both a *chum* and a *friend* will cause its *friend's chum* to become its *chum's friend*. Believe it or not, when the dust has finally settled, the desired result does appear.

This algorithm has three additional interesting properties: First, it is possible to match up two lists of unequal length; the algorithm simply causes each extra cell at the end of the longer list to have no *friend* (that is, a *null friend*) (see Figure 6). Second, if, in the initialization, one makes the first cell of *list2* the *friend* of the first cell of *list1*, but not vice versa, then at the end all the cells of *list1* will have pointers to their *friends*, but the cells of *list2* are unaffected (their *friend* components remain *null*). Third, like the other linked-list algorithms, this one can process many lists (or pairs of lists) simultaneously.

With this primitive, one can efficiently perform such operations as componentwise addition of two vectors represented as linked lists.

Region Labeling

How are linked-list operations used in practical applications? One such application is region labeling, where, given a two-dimensional image (a grid of pixel values), one must identify and uniquely label all regions. A *region* is a maximal set of pixels that are connected and all have the same value. Each region in the image must be assigned a different label, and this label should be distributed to every pixel in the region.

An obvious approach is to let each processor of a parallel computer hold one pixel. On a parallel computer with *N* processors communicating in a fixed two-dimensional pattern, so that each processor can

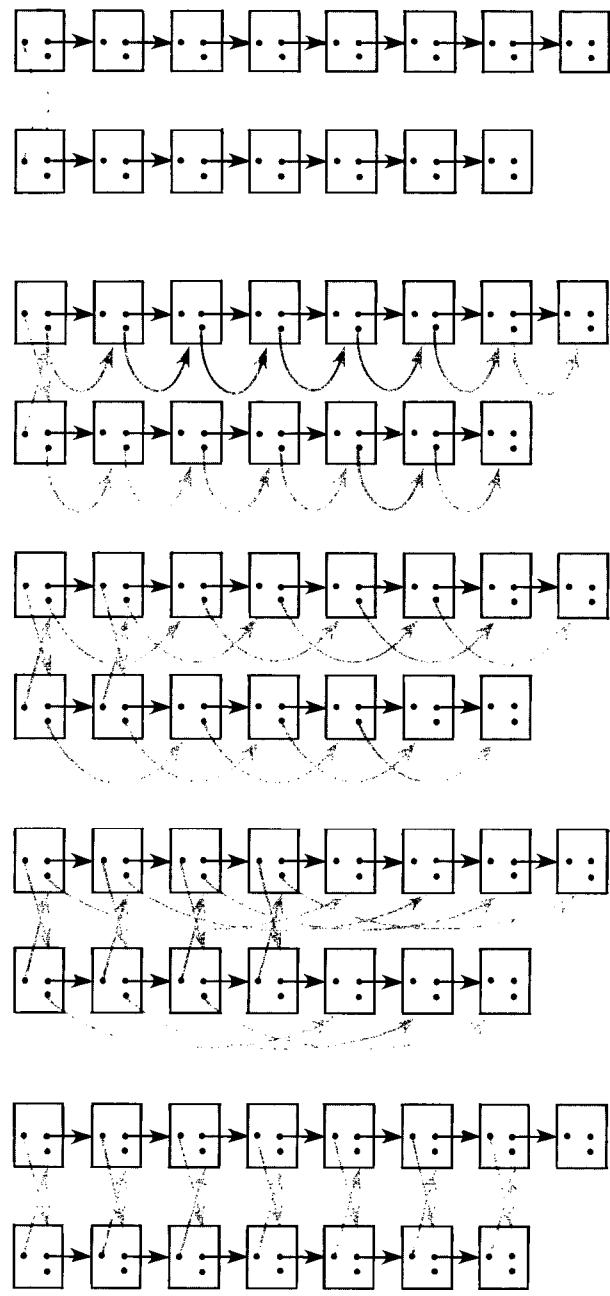


FIGURE 6. Matching Up Components of Two Lists

communicate directly only with its four neighbors, this problem can be solved simply for an *N*-pixel image in the following manner: Since every processor has an address and knows its own address, a region will be labeled with the largest address of any processor in that region. To begin with, let each processor have a variable called *largest*, initialized to its own address, and then repeat the following step until there is no overall change of state. Each

processor trades *largest* values with all neighbors that have the same pixel value, and replaces its own *largest* value with the maximum of its previous value and any values received from neighbors. The address of the largest processor in a region therefore spreads out to fill the region.

Although the idea is simple, the algorithm takes time $O(\sqrt{N})$ in simple cases, and time $O(N)$ in the worst case (for images that contain a long "snake" that fills the picture). Lim [19] has devised algorithms for the Connection Machine that use linked-list techniques to solve the problem in time $O(\log N)$. In one of these algorithms, the basic idea is that every pixel can determine by communication with its two-dimensional neighbors whether it is on the boundary of its region, and, if so, which of its neighbors are also on the boundary. Each boundary pixel creates a pointer to each neighbor that is also a boundary pixel, and voilà: Every boundary has become a linked list (actually, a doubly linked list) of pixels. If the processor addresses are of the obvious form $x + Wy$, where x and y are two-dimensional coordinates and W is the width of the image, then the processor with the largest address for any region will be on its boundary. Using logarithmic-time linked-list algorithms, all the processors on a region boundary can agree on what the label for the region should be (by performing a maximum reduction on the linked list and then spreading the result back over the list). Since all the boundaries can be processed in parallel, it is then simply a matter of propagating the labels inward from boundaries to interior pixels. This is accomplished by a process similar to a parallel-prefix computation on the rows of the image. (There are many nasty details having to do with orienting the boundaries so that each boundary pixel knows which side is the interior and handling the possibility that regions may be nested within other regions, but these details can also be handled in logarithmic time.)

This application has the further property that the linked-list structure is not preexistent; rather, it is constructed dynamically as a function of the content of the image being processed. There is therefore no way to cleverly allocate or encode the structure ahead of time (e.g., as an array). The general communication facility of the Connection Machine model is therefore essential to the efficient execution of the algorithm.

Recursive Data Parallelism

We have often found situations where data parallelism can be applied recursively to achieve multiplicative effects. To multiply together a long chain of

large matrices (a commonplace calculation in the study of systems modeled by Markov processes), we can use the associative scan operation to multiply together N matrices with $\log N$ matrix multiplications. In each matrix multiplication, the opportunity for parallelism is obvious, since matrix multiplication is defined in terms of operations on vectors. Another possibility would be to multiply the matrices using a systolic array-type algorithm [18], which will always run efficiently on a computer of the Connection Machine type. If the matrices are sparse, then we use the Pan-Reif algorithm [21], a data parallel algorithm that multiplies sparse matrices represented as trees. This algorithm fits well on a fine-grained parallel computer as long as it has capabilities for general communications. If the entries of the matrices contain high-precision numbers, there is yet another opportunity for parallelism within the arithmetic operations themselves. For example, using a scan-type algorithm for carry propagation, we can add two n -digit numbers in $O(\log n)$ time. Using a pipelined carry-save addition scheme [17], we can multiply in linear time, again by performing operations on all the data elements (digits) in parallel.

Summary and Conclusions

In discussing what kinds of computations are appropriate for data parallel algorithms, we initially assumed—when we began our work with the Connection Machine—that data parallel algorithms amounted to very regular calculations in simulation and search. Our current view of the applicability of data parallelism is somewhat broader. That is, we are beginning to suspect that this is an appropriate style wherever the amount of data to be operated upon is very large. Perhaps, in retrospect, this is a trivial observation in the sense that, if the number of lines of code is fixed and the amount of data is allowed to grow arbitrarily, then the ratio of code to data will necessarily approach zero. The parallelism to be gained by concurrently operating on multiple data elements will therefore be greater than the parallelism to be gained by concurrently executing lines of code.

One potentially productive line of research in this area is searching for counterexamples to this rule: that is, computations involving arbitrarily large data sets that can be more efficiently implemented in terms of control parallelism involving multiple streams of control. Several of the examples presented in this article first caught our attention as proposed counterexamples.

It is important to recognize that this question of

programming style is not synonymous with the hardware design issue of MIMD versus SIMD computers. MIMD computers can be well suited for executing data parallel programs: In fact, depending on engineering details like the cost of synchronization versus the cost of duplication, they may be the best means of executing data parallel programs. Similarly, SIMD computers with general communication can execute control-style parallelism by interpretation. Whether such interpretation is practical depends on the details of costs and requirements.

While interesting and important in their own right, these questions are largely independent of the data parallel versus control parallel programming styles.

Having one processor per data element changes the way one thinks. We found that our serial intuitions did not always serve us well in parallel contexts. For example, when sorting is fast enough, the order in which things are stored is often unimportant. Then again, if searching is fast, then sorting may be unimportant. In a more general sense, it seems that the selection of one data representation over another is less critical on a highly parallel machine than on a conventional machine since converting all the memory from one representation to another does not take a large amount of time. One case where our serial intuitions misled us was our expectation that parallel machines would dictate the use of binary trees [14]. It turns out that linear linked lists serve almost as well, since they can be easily converted to balanced binary trees whenever necessary and are far more convenient for other purposes.

Our own transition from serial to parallel thinkers is far from complete, and we would be by no means surprised if some of the algorithms described in this article begin to look quite "old-fashioned" in the years to come.

REFERENCES

1. Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs (1977 ACM Turing Award Lecture). *Commun. ACM* 21, 8 (Aug. 1978), 613–641.
2. Batcher, K.E. Sorting networks and their applications. In *Proceedings of the 1968 Spring Joint Computer Conference* (Reston, Va., Apr.) AFIPS, Reston, Va., 1968, pp. 307–314.
3. Batcher, K.E. Design of a massively parallel processor. *IEEE Trans. Comput.* C-29, 9 (Sept. 1980), 836–840.
4. Bawden, A. A programming language for massively parallel computers. Master's thesis, Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., Sept. 1984.
5. Bawden, A. Connection graphs. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*. ACM, (Cambridge, Mass., Aug. 4–6). New York, 1986, pp. 258–265.
6. Blelloch, G. AFL-I: A programming language for massively concurrent computers. Master's thesis, Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., June 1986.
7. Blelloch, G. Parallel prefix versus concurrent memory access. Tech. Rep., Thinking Machines Corp., Cambridge, Mass., 1986.

8. Bouknight, W.J., Denenberg, S.A., McIntyre, D.E., Randall, J.M., Sameh, A.H., and Slotnick, D.L. The ILLIAC IV system. *Proc. IEEE* 60, 4 (Apr. 1972), 369–388.
9. Christman, D.P. Programming the Connection Machine. Master's thesis, Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., Jan. 1983.
10. Christman, D.P. Programming the Connection Machine. Tech. Rep. ISL-84-3, Xerox Palo Alto Research Center, Palo Alto, Calif., Apr. 1984. (Reprint of the author's master's thesis at MIT.)
11. Falkoff, A.D., and Orth, D.L. Development of an APL standard. In *APL 79 Conference Proceedings* (Rochester, N.Y., June). ACM, New York, pp. 409–453. Published as *APL Quote Quad* 9, 4 (June 1979).
12. Flanders, P.M., et al. Efficient high speed computing with the distributed array processor. In *High Speed Computer and Algorithm Organization*, Kuch, Lawrie, and Sameh, Eds. Academic Press, New York, 1977, pp. 113–127.
13. Haynes, L.S., Lau, R.L., Siewiorek, D.P., and Mizell, D.W. A survey of highly parallel computing. *Computer* (Jan. 1982), 9–24.
14. Hillis, W.D. *The Connection Machine*. MIT Press, Cambridge, Mass., 1985.
15. Iverson, K.E. *A Programming Language*. Wiley, New York, 1962.
16. Knuth, D.E. *The Art of Computer Programming*. Vol. 3. *Sorting and Searching*. Addison-Wesley, Reading, Mass., 1973.
17. Knuth, D. E. *The Art of Computer Programming*. Vol. 2, *Seminumerical Algorithms* (Second Edition). Addison-Wesley, Reading, Mass., 1981.
18. Kung, H.T., and Lieserson, C.E. Algorithms for VLSI processor arrays. In *Introduction to VLSI Systems*, L. Carver and L. Conway, Eds. Addison-Wesley, New York, 1980, pp. 271–292.
19. Lim, W. Fast algorithms for labeling connected components in 2-D arrays. Tech. Rep. 86.22, Thinking Machines Corp., Cambridge, Mass., July 1986.
20. Minsky, M., and Papert, S. *Perceptrons*. 2nd ed. MIT Press, Cambridge, Mass., 1979.
21. Pan, V., and Reif, J. Efficient parallel solution of linear systems. Tech. Rep. TR-02-85, Aiken Computation Laboratory, Harvard Univ., Cambridge, Mass., 1985.
22. Schwartz, J.T. Ultracomputers. *ACM Trans. Program. Lang. Syst.* 2, 4 (Oct. 1980), 484–521.
23. Shaw, D.E. *The NON-VON Supercomputer*. Tech. Rep., Dept. of Computer Science, Columbia Univ., New York, Aug. 1982.
24. Steele, G.L., Jr., and Hillis, W.D. Connection machine Lisp: Fine-grained parallel symbolic processing. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming* (Cambridge, Mass., Aug. 4–6). ACM, New York, 1986, pp. 279–297.
25. Turner, D.A. A new implementation technique for applicative languages. *Softw. Pract. Exper.* 9 (1979), 31–49.

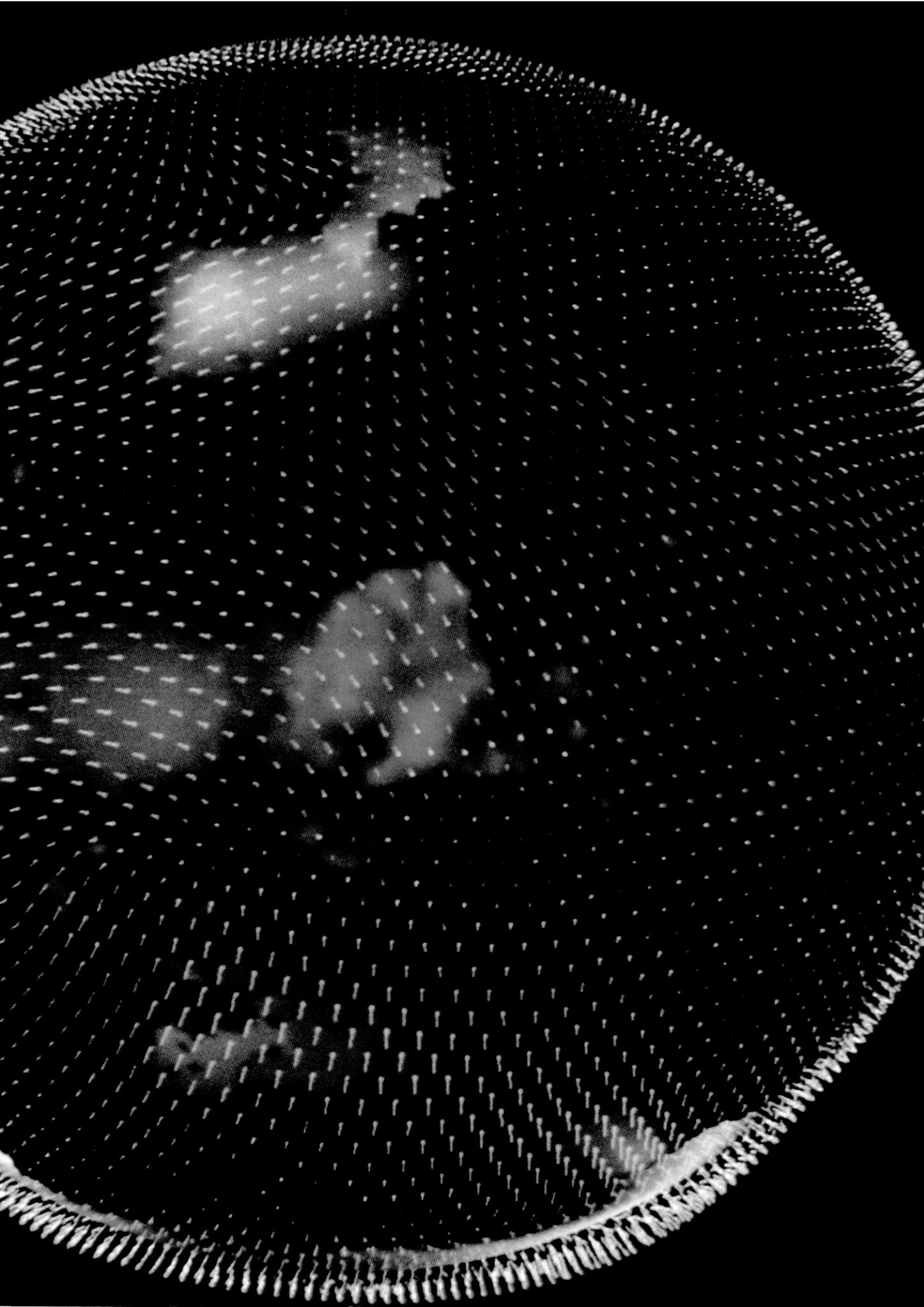
CR Categories and Subject Descriptors: B.2.1 [Arithmetic and Logic Structures]: Design Styles—parallel; C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors)—parallel processors; D.1.3 [Programming Techniques]: Concurrent Programming; D.3.3 [Programming Languages]: Language Constructs—concurrent programming structures; E.2 [Data Storage Representations]: linked representations; F.1.2 [Computation by Abstract Devices]: Modes of Computation—parallelism; G.1.0 [Numerical Analysis]: General—parallel algorithms

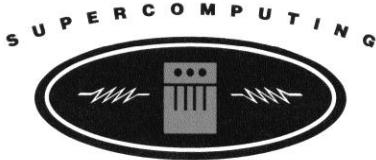
General Terms: Algorithms

Additional Key Words and Phrases: Combinator reduction, combinator, Connection Machine computer system, log-linked lists, parallel prefix, SIMD, sorting, Ultracomputer

Authors' Present Address: W. Daniel Hillis and Guy L. Steele, Jr., Thinking Machines Corporation, 245 First Street, Cambridge, MA 02142-1214.

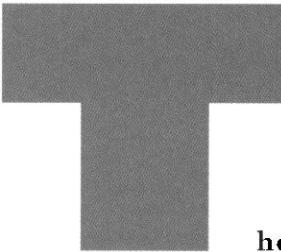
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.





W. Daniel Hillis and Lewis W. Tucker

THE CM-5 CONNECTION MACHINE: A SCALABLE SUPERCOMPUTER



The CM-5 Connection Machine is a scalable homogeneous multiprocessor designed for large-scale scientific and business applications. In this article we describe its architecture and implementation from the standpoint of the programmer or user of parallel machines. In particular, we emphasize three features of the Connection Machine architecture: scalability, distributed memory/global addressing, and distributed execution/global synchronization. We believe that architectures of this type will replace most other forms of supercomputing in the foreseeable future. • Examples of the current applications of the machine are included, focusing particularly on the machine's ability to support a variety of programming models. The article is intended to be a general overview appropriate to a user or programmer of parallel machines, as opposed to a hardware designer. The latter may prefer more detailed descriptions elsewhere [16, 23, 24].

Architecture vs. Implementation

In describing the CM-5, it is useful to distinguish between issues of architecture and implementation. While there is no hard line between the two, the architectural features of the machine are those features that are intended to remain constant across multiple implementations. In a microprocessor, for example, one of the most important features of the architecture is usually the instruction

set. Object-code compatibility from one generation of the machine to the next is an important issue for many microprocessor applications. Supercomputer applications, on the other hand, are usually written in higher-level languages, so the instruction set is not a necessary part of the architectural specification of the CM-5. The architecture is defined by other issues, such as the addressability of memory, the *user-visible* mechanisms of synchronization, and the function-

Zuse Mounted on
86% /u19



ability of the communication and I/O systems.

We will describe both the architecture of the CM-5 and its current implementation. Figure 1, for example, is the architectural block diagram of the machine that is unlikely to change. The specifications shown in Table 1, on the other hand, are descriptions of the current implementation, which are likely to change from year to year as new versions of the machine are introduced and faster, denser chips become available.

Architectural Overview

The CM-5 is a coordinated homogeneous array of RISC microprocessors. We note that the acronym for this description is CHARM. CHARM architectures take advantage of high-volume microprocessors and memory components to build high-performance supercomputers that are capable of supporting a wide range of applications and programming styles. By CHARM we mean any homogeneous collection of RISC microprocessors that has the following coordination mechanisms:

1. A low-latency, high-bandwidth communications mechanism that allows each processor to access data stored in other processors
2. A fast global synchronization mechanism that allows the entire machine, including the network, to be brought to a defined state at specific points in the course of a computation

These coordinating mechanisms support the efficient execution of sequentially deterministic parallel programs. Both mechanisms are important for supporting the data-parallel model of programming [14]. The CM-5 is a CHARM architecture and we believe most of the massively parallel machines currently on the drawing boards in the U.S., Europe, and Japan are also CHARMS.

The CM-5 consists of a large number of processing nodes connected by a pair of networks, implementing the coordination functions described earlier. The number of processors may vary from a few dozen to tens of thousands. The number of pro-

sors in the machines currently installed ranges from 32 to 1,024. The networks are similar to the backplanes of conventional computers in that the machine's I/O and processing can be expanded by plugging more modules into the networks. Unlike a bus in a conventional computer, the bandwidths of the networks increase in proportion to the number of processors (see Figure 1). The memory of the machine is locally distributed across the machine for fast local access. Each processor accesses data stored in remote memories of the processors via the network.

One of the most important features of the CM-5 is its scalability. The current implementation, including networks, clocking, I/O system, and software are designed to scale reliably up to 16,384 processors. The same application software runs all of these machines. Because the machine is constructed from modular components, the number of processors, amount of memory, and I/O capacity can be expanded on-site, without requiring any change in application software.

The Processor Node

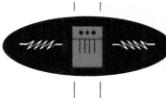
Microprocessor technology changes very rapidly and the CM-5 was designed with the assumption that the processing node would therefore change from implementation to implementation. This implies that the system software, including compilers and operating system, be written in a way that is largely processor-independent. It also requires, at the hardware level, that the clocking system of the network is decoupled from that of the processor, so the processor and network can be upgraded independently.

The current implementation of the CM-5 uses a SPARC-based processing node, operating at a 32- or 40MHz clock rate, with 32MB of memory per node. The SPARC is augmented with auxiliary chips that increase floating-point execution rate and memory bandwidth. The result is a 5-chip microprocessor with performance characteristics similar to that of single-chip microprocessors that will be widely available in a

few more years. The SPARC was chosen primarily for its wide base of existing software. One drawback in using a standard microprocessor, SPARC included, is that standard RISC processors generally depend on a data cache to increase the effective memory bandwidth. These caching techniques do not perform as well in a large parallel machine, because many applications reference most of the data in memory on each pass of an iteration. This results in relatively little reuse of data cache references, reducing the effectiveness of the cache. In a parallel machine, the absolute uncached bandwidth between the memory and processor is an important parameter for performance.

In the current implementation of the CM-5 we addressed the memory bandwidth problem by adding four floating-point data paths, each with direct parallel access to main memory, to the standard SPARC microprocessor. The four data paths have an aggregate bandwidth to main memory of 500MB/sec per processing node, and 128MFLOPS peak floating-point rate. This is enough bandwidth to allow a single memory reference for each multiply and add when all of the floating-point units are operating at peak rate. A block diagram of the node is shown in Figure 2. The data paths are capable of performing scalar or vector operations on IEEE single- or double-precision floating-point numbers, including 64-bit integers. They also support specialized operations such as bit count. While the use of additional floating-point units with their own paths to memory is an important implementation feature of the current CM-5, it is not generally visible to users at the programming level.

The measured rate in a single node is currently 64MFLOPS for matrix multiply and 100MFLOPS on matrix-vector multiplication. The rate for the 500×500 Linpack benchmark is 50MFLOPS. The sustained rate for the node 8K-point FFTs is 90MFLOPS. In a typical application, since much of the time is spent in interprocessor communications, the peak execution rate of the node is not usually a reliable predic-



tor of average performance on applications. In data parallel applications, where the opportunities for parallelism increase with the size of data, performance increases almost linearly with an increasing number of processors and dataset size. This near-linear scaling is possible because network bandwidth scales in proportion to the number of processing

nodes. Benchmarks such as the large matrix Linpack [5], in which the problem size increases with the amount of available memory, as shown in Figure 3, are an example of a problem with near-linear speedup. Measurements on various sizes of CM-5's show Linpack benchmark running with near-linear speedups at rates of 45MFLOPS/node. A 16,000

processor CM-5 is expected to exceed 700GFLOP/sec in performance on this benchmark.

Choosing the right balance between memory bandwidth, communication bandwidth, and floating-point capacity is one of the major design issues of a supercomputer. In the past, when floating point units were very expensive, designers mea-

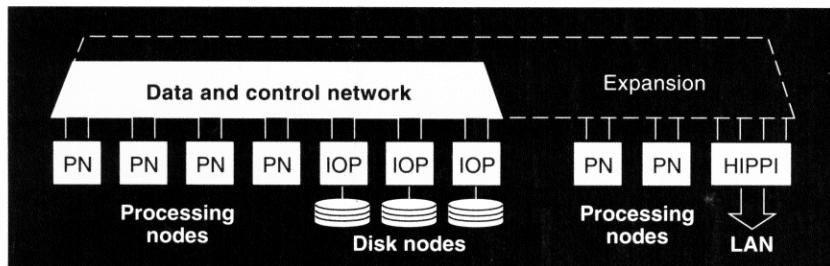


Figure 1.

CM-5 Communications networks

The CM-5 consists of processing, control, and I/O nodes connected by two scalable networks which handle communication of data and control information. Unlike a bus on a conventional computer, the communications network is scalable in the sense that the bandwidth of the network grows in proportion to the number of nodes. Shown is the expansion of the network that results when additional processing and I/O nodes are added.

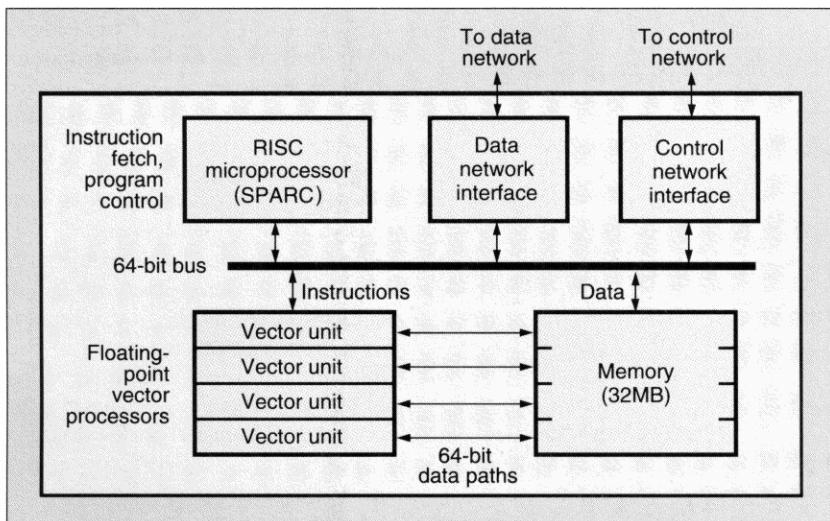


Figure 2.

CM-5 processing node with vector units

Each processing node of the current implementation of the CM-5 consists of a RISC microprocessor, 32MB of memory, and an interface to the control and data interconnection networks. The processor also includes four independent vector units, each with a direct 64-bit path to an 8MB bank of memory. This gives a processing node with a double-precision floating-point rate of 128MFLOPS, and a memory bandwidth of 512MB/sec.

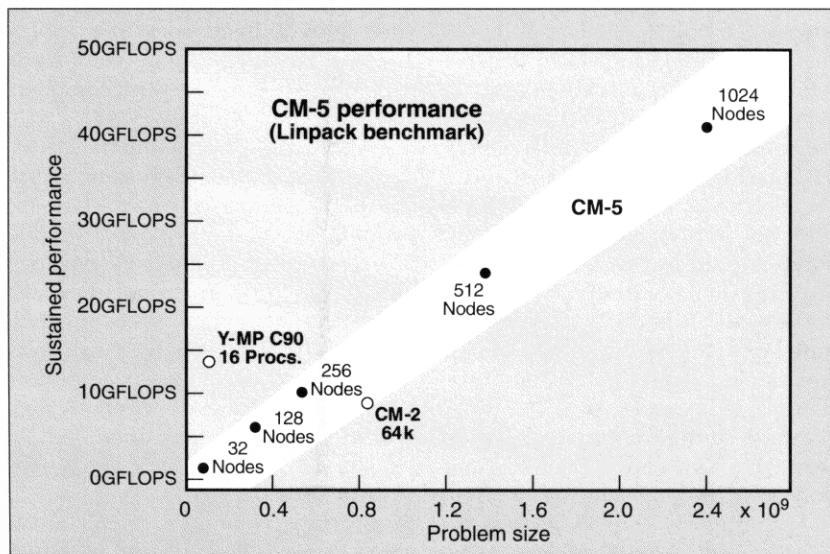


Figure 3.

Scalable performance

The performance of data parallel applications increases with the size of data and available processing resources. Shown is the measured performance of the Linpack benchmark on the CM-5 for increasing problem size and number of processors.

sured the efficiency of their machine by the percentage utilization of the floating-point units in a typical application. This sometimes led to designs that were inefficient at using an even more precious resource, such as memory bandwidth. Since applications vary in their floating-point-to-memory reference ratio, no architecture uses both of these resources at 100% capacity in every application. Today, floating-point units represent a relatively small portion of the total cost, so optimizing the design around their utilization makes little sense. Instead, the design point chosen in

the hypercubes or grids. Specifically, the network expands naturally as new nodes are added, and the capacity of network traffic across the machine (bisection bandwidth) is proportional to the number of nodes in the machine. This is not the case with grid-based networks. The fat-tree topology also has good fault tolerance properties.

The data network uses a packet-switched router which guarantees deadlock-free delivery. The message-routing system uses an adaptive load-balancing algorithm to minimize network congestion. Packets are

programming model.

The Control Network

The CM-5's control network supports the broadcast, synchronization, and data reduction operations necessary to coordinate the processors. The network can broadcast information, such as blocks of instructions, to all processors simultaneously or within programmable segments. This control network also implements global reduction operations, such as computing the total of numbers posted by all processors, computing global logical AND, global MIN, and so forth. Also, the control network implements global and segmented reduction and parallel prefix functions that have been shown to be important building blocks for parallel programming [3, 14].

Unlike the CM-2, the operations of the individual CM-5 instructions are not necessarily synchronized. Each processor has its own program counter, and fetches instructions from a local instruction cache. Rapid synchronization, however, is provided by the control network employing a barrier synchronization mechanism. Each processor may elect to wait for all the other processors to reach a defined point in the program before proceeding. This mechanism allows processors to detect whether data network messages are still in transit between processors before establishing synchronization. The details of this mechanism are not normally visible to the application programmer, since the appropriate synchronization commands are typically generated by compilers and run-time libraries.

From the programmer's standpoint, the fast synchronization supports the sequential control flow required by the data parallel model. The automatically inserted synchronization barriers guarantee that each data parallel operation is completed, before the next begins. Such synchronization events occur whenever there are potential dependencies between the two parallel operations.

Input/Output

Many supercomputers spend much of their time accessing and updating

Table 1. CM-5 specs

The specifications of the current implementation of the CM-5 are shown here. The largest machine shipped to date is 1,024 processors, but the software networks clocking, power distribution, and so forth are all designed to scale to the 16,000 processor system.

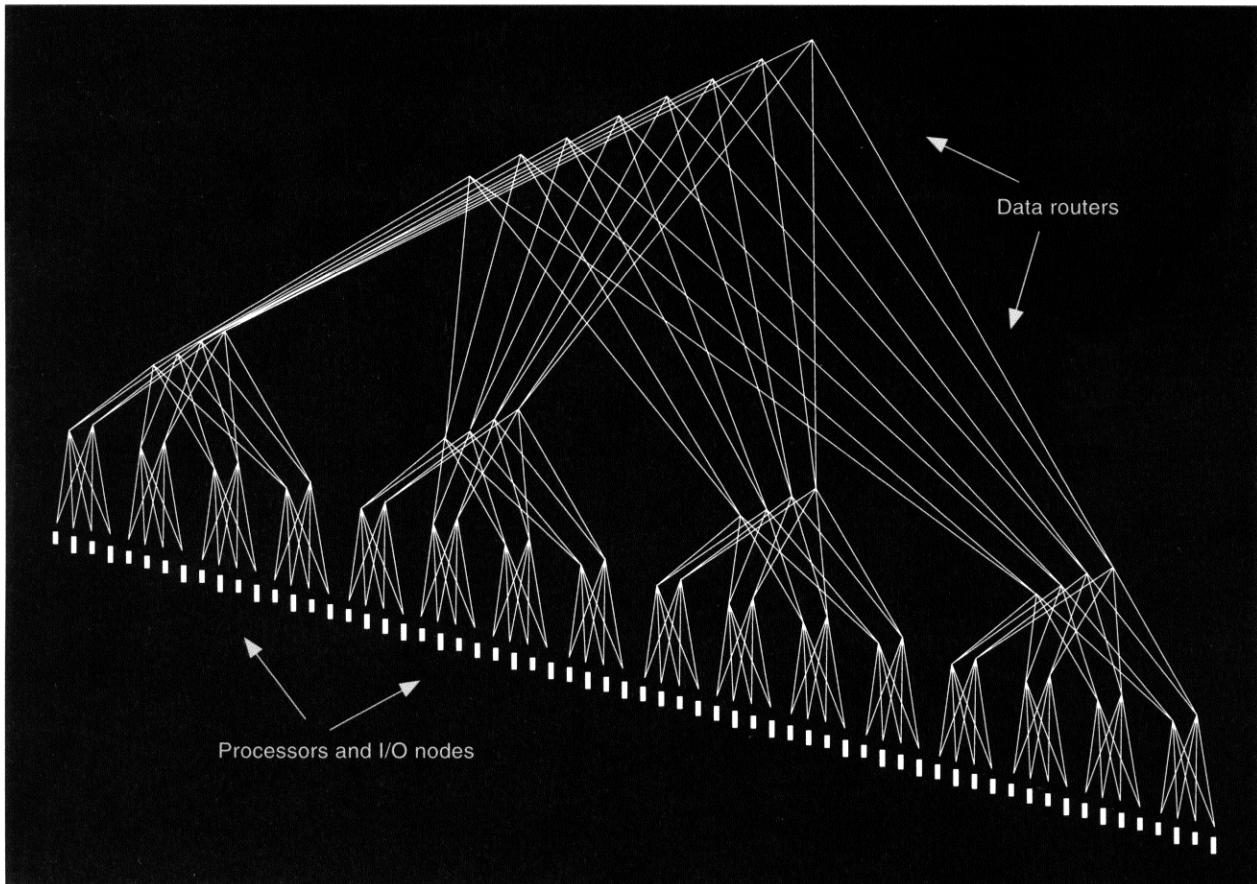
Processors	32	1024	16384
Network address	64	2048	32768
Number of data paths	128	4096	65536
Peak MFLOPS	4GFLOPS	128GFLOPS	2048GFLOPS
Memory	1GB	32GB	512GB
Memory bandwidth	16GB/sec	512GB/sec	8192GB/sec
Bisection bandwidth	320MB/sec	10GB/sec	160GB/sec
I/O bandwidth	320MB/sec	10GB/sec	160GB/sec
Global synch time	1.5 microsecs	3 microsecs	4 microsecs

the CM-5 was based on the balance point calculated by the equal ratios method [12], which takes the relative cost and performance aspect into account.

The Data Network

Like the CM-2, the CM-5 incorporates two important types of communication networks: one for point-to-point communication of data, and a second for broadcast and synchronization. As in the CM-2, the data network is implemented as a packet-switched logarithmic network. The CM-5 uses a fat-tree topology (see Figure 4) [15], which is a more general and flexible structure than the grid and hypercube used in the CM-1 and CM-2. This topology offers better scalability properties than

delivered at a rate of 20MB/sec to nearby nodes. The bisection bandwidth of a 1,024-node CM-5, that is, the minimum bandwidth between one-half of the machine and the other, is 5GB/sec in each direction. Network latencies across a 1k-processor machine are between 3 and 7 microseconds. This bandwidth is achieved on regular or irregular communications patterns. Bounds checking hardware, error checking, end-to-end counting of messages, and rapid context switching of network state protects users from interference in a time-shared, multiple-partition configuration. As discussed later, the data network, in conjunction with the run-time system software, supports either a message-passing or a global address space



data stored externally to the processor. The most challenging part of input/output is generally the transfer of data to and from disk subsystems. The CM-5 was designed with the assumption that mass storage devices, such as disk arrays, will continue to increase in capacity and bandwidth. One implication is that no fixed I/O channel capacity will remain appropriate for very long. The CM-5 I/O system is constructed in such a way as to be connected directly into the processor's communication network. The network bandwidth available to a given I/O device is determined by the number of taps into the network. For each I/O device, multiple network taps are combined to form a single I/O channel that operates at whatever rate the device will allow.

On CM-5 systems, in addition to processing nodes, there will typically be a number of independent nodes with attached disk drive units. These "disk nodes" are configured by software to form a completely scalable, high-bandwidth RAID file system—

limited only by the aggregate bandwidth of its individual drives. To an application, the disks appear as a single very-high-bandwidth mass storage subsystem.

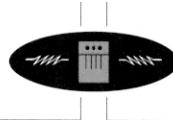
Unix Operating System

The CM-5 is normally operated as a multiuser machine. The system supports a Unix-based operating system capable of time-sharing multiple users and a priority-based job queuing system, compatible with NQS. Tasks are fully protected from one another through memory management units and bounds checking hardware in the network. The network state is always accessible to the operating system so messages in transit can be "swapped out" during the process switch from user to user.

In addition to normal time-sharing, the CM-5 supports a form of space sharing, called "partitioning." The CM-5 can be partitioned under software control into several independent sections. Each partition effectively works as an independent,

Figure 4.
Data network topology

The data network uses a fat tree topology. Each interior node connects four children to two or more parents (only two parents are shown). Processing nodes form the leaf nodes of the tree. As more processing nodes are added, the number of levels of the tree increases, allowing the total bandwidth across the network to increase proportionally to the number of processors.



For a 1-D array, compute the average of nearby neighbors.

Fortran 77 with automatic parallelization

```
integer size, i
parameter (size=1000)
double precision x(size), y(size)

do i = 2, size-1
    y(i) = (x(i-1)+x(i+1))/2.0
enddo
```

High-performance Fortran (Fortran 90)

```
forall (i=2:size-1)
    y(i) = (x(i-1)+x(i+1))/2.0
```

C*

```
where (pcoord(0)>0 && pcoord(0)<SIZE-1)
    y = ([-1]x+[+1]x)/2.0;
```

Message passing

```
int nodesize = size/nproc;

if (myself > 0)
    CMMD_send_noblock (myself-1,tag, &x[0], len);
if (myself < (nproc-1) {
    CMMD_send_noblock (myself+1,tag, &x[nodesize-1], len);
    CMMD_receive (myself+1,tag,&right,len);
}
if (myself > 0)
    CMMD_receive(myself-1,tag,&left, len);

for (i=1; i < nodesize; i++) y[i] = (x [i-1] + x[i+1])/2.0;
y[0] = (left + x[1])/2.0;
y[nodesize-1] = (x[nodesize-2] + right)/2.0
```

***Lisp**

```
(*if (and (> (self-address!!) 0)
           (< (self-address!!) x size))
    (*set y (/ (+ (news!! x -1) (news!! = 1)) 2)))
```

Id

```
def relax x =
{ 1,u = bounds x;
  typeof x = vector *0;
  in { array(1,u) of
    | [i] = (x[i-1]+x[i+1])/2.0 || i <- 1+1 to u-1}
  };
```

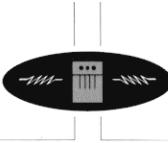


Figure 5. Rosetta Stone

This example program fragment shows the same function written in different programming styles. All of these programs compile and run efficiently on the CM-5. The program computes the average of neighboring elements in a nondimensional array.

time-shared, machine. Hardware in the communication network isolates these partitions in such a way that the computations in one partition of the machine have no effect on the computations in another. Each partition appears to the software as a single linear address space, although it may map into physically discontinuous sections of the machine. It is even possible to power down a partition of the machine for maintenance while other partitions continue to operate.

Because the CM-5 is used in large-scale, long-running applications requiring very large datasets, the standard Unix file system has been extended to handle files with terabytes of data. Checkpointing facilities, invoked either automatically or under user control, have been added to the operating system [22].

Relation of CM-5 to CM-2

The CM-5 is the successor to the CM-2 and CM-200. The CM-5 shares with the CM-2 the two-network organization, the global address local memory structure, the ability to broadcast and synchronize instructions, and special hardware support for data parallel execution. Its architecture differs most dramatically from the CM-2 in the ability of the individual nodes to execute instructions independently and in the organization of the I/O system. Also, the CM-5 uses a larger-grained 64-bit processor, instead of the single-bit processors and 32-bit floating-point paths of the CM-2. Most programs written for the CM-2 only require recompilation to run on the CM-5.

The most important lesson learned from the CM-2 was the importance of the scalable data parallel model which is used in most commercial and scientific applications of massively parallel machines. As it turned out, the strict per-instruction

synchronization of the CM-2 was not required to support this model. The CM-5 is a MIMD machine (multiple instruction/multiple data), although it incorporates many features that are normally found only in SIMD (single instruction/multiple data) machines. The machine incorporates several important architectural features from message-passing machines such as the Caltech Hypercube [21]. The data network of fat trees is based on work from MIT [15] and the New York University Ultracomputer [20].

Support for Multiple Programming Models

The CM-5 is particularly effective at executing data parallel programs, but it was designed with the assumption that no one model of parallel programming will solve all users' needs. The data parallel model, which is embodied in languages such as Fortran-90, *Lisp, and C*, has proved extremely successful for broad classes of scientific and engineering applications. On the other hand, this model does not seem to be the best way to express a control structured algorithm such as playing chess by searching the game tree [8]. In this case, a message-passing model is often more appropriate. Other applications may be better suited by process-based models, in which tasks communicate via pipes or shared variables, or by object-based models in which each processor executes a different program, or even dataflow models, such as those supported by the language Id. Figure 5 shows a "rosetta stone" of program fragments performing a similar operation in many different programming languages. All of these languages compile efficiently for the CM-5. For this particular example, the data parallel languages offer the most concise formulation, but in other cases message-passing-based approaches may offer a more natural expression of an algorithm.

Since the actual hardware required to support all of these programming models is very similar, it is likely that most parallel machines of the future will be designed to support all of these programming mod-

els. All that is required is a good communication system, an efficient global synchronization mechanism, and a fast hardware broadcast and reduction. With today's technology, this is relatively inexpensive. This is part of the charm of CHARM architectures.

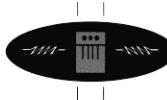
Shared Memory vs. Shared Address Space

Some of the most interesting issues in computer architecture arise in resolving the conflicts between the needs of the programmer and the constraints of the hardware. Sometimes apparent conflicts are resolved by careful separation of the actual requirements. For example, consider the shared vs. distributed memory issue. Programmers would like all of the data to be accessible from every processor, which suggests the use of a single central memory that is accessible by all the processors. Hardware constraints, on the other hand, make this central memory approach costly and inefficient. Local memory, distributed among the processors, is much faster and less expensive.

It would seem at first glance that the conflict is irresolvable, and that the user will be forced to choose between convenience and efficiency. Fortunately, the choice is unnecessary. It is possible to address both the hardware and the software needs simultaneously by distributing memories for fast local access and providing an additional mechanism for accessing all of the memory through a network. This allows the compiler and operating system to provide the user with a shared address space, without sacrificing the cost-effectiveness, speed and efficiency of distributed memory. The CM-5 adopts this combination of shared address space and distributed memory. Hennessy and Patterson have published a discussion of these two dimensions of architecture [10]. Figure 6 shows one of their tables with the addition of the CM-5.

SIMD vs. MIMD

A similar analysis can be applied to the SIMD/MIMD issue. Programmers often find it most convenient to specify programs in terms of a single



flow of control [2], using a sequential or a data-parallel programming style. This frees the programmer from issues of synchronization, deadlock, stale access, nondeterministic ordering, and so forth. Parallel computers, on the other hand, can operate more efficiently executing independent instruction streams in each processor, since a sequential program overspecifies the ordering and synchronization. These two issues are resolved by providing a machine with both independent instruction streams, and the synchronization and broadcast mechanisms necessary to support a sequential programming style. The program is guaranteed to compute exactly the same results as it would operating on a single processor, although the result is generally computed more quickly.

If we take the classical SIMD machine to be one that is synchronized on every instruction, and the classical MIMD machine to be one in which synchronization is left to the programmer, then the CM-5 normally operates in the middle ground; that is, synchronization occurs only as often as necessary to guarantee getting the same answer as the globally synchronized machine. These invocations of the hardware synchronization operations are generated automatically by the compiler. The processors are free to operate independently whenever such synchronization is unnecessary. Even applications that were originally written for traditional MIMD machines have been able to take good advantage of these hardware synchronization mechanisms. Of course, it is also possible to program the machine as a standard MIMD machine, or for that matter to force a synchronization on every instruction execution, in which case it operates as a SIMD machine. This is illustrated in Figure 7.

Applications

When massively parallel computers were first introduced, many expected that they would only achieve good performance on a narrow range of applications. Experience has shown that, contrary to initial expectations, most applications involving large amounts of data can

take advantage of the computational power of massively parallel machines [9]. This is not to say that most existing applications, as currently written, run well on massively parallel machines. They do not. Most existing supercomputer applications were designed with much smaller, non-parallel machines in mind. The average application on Cray Y-MP at the NCSA Supercomputer Center, for example, runs at 70MFLOP/sec in 25MB of memory. Such small-scale applications rarely have the inherent parallelism to take advantage of massive parallelism. On the other hand, large-scale applications that involve many GBs of data can almost always be written in a way that takes good advantage of large-scale parallelism. In a recent survey of the world's most powerful supercomputers, four out of the top five systems reported were Connection Machines [6].

Current applications of the Connection Machine system include high-energy physics, global climate modeling and geophysics, astrophysics, linear and nonlinear optimization, computational fluid dynamics and magnetohydrodynamics, electromagnetism, computational chemistry, computational electromagnetics, computational structural mechanics, materials modeling, evolutionary modeling, neural modeling, and database search [19]. There are many other examples. These applications use a variety of numerical methods, including finite difference and finite element schemes, direct methods, Monte Carlo calculations, particle-in-cell methods, *n*-body calculations, and spectral methods [4].

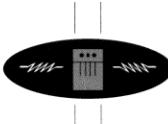
Many applications that were originally developed for the CM-2 achieve high rates of sustained performance on the CM-5. For example, a rarefied gas dynamics simulation, for modeling situations such as spacecraft reentry [17, 18], which runs at 8GFLOP/sec on the largest CM-2, runs at a sustained rate of 64GFLOP/sec on a 1,024 processor CM-5 (1GFLOP/sec is about the peak rate of a single processor on a Y-MP C90). Another example is the Global Climate Modeling software developed by Los Alamos National Laboratory

(see Figure 8). This simulation is designed to model global climate warming due to the buildup of greenhouse gases. The algorithm relies on a dynamically varying mesh of over 250,000 points and 1,200,000 tetrahedra to cover the Earth's atmosphere. The application consists of over 30,000 lines of Fortran. Dynamic rearrangement of the mesh automatically adjusts local resolution to resolve important details of the flow, such as storm formations. Simulation of a six-week time period requires about one hour of simulation time on a 64K CM-2. When run on a 1,024 processor CM-5, this six-week simulation completes in less than five minutes.

Massively parallel machines are also becoming increasingly important in applications outside of science and engineering. Many of these commercial applications take advantage of the scalable input/output capability of the CM-5. For example, American Express corporation is using two CM-5's to process its large credit card databases. Another example is the oil industry, where companies such as Mobil and Schlumberger use CM-5's for large-scale seismic processing.

Because the CM-5 provides direct user-level access to the network hardware, it is also used as an experimental platform for testing new architectures. There is no operating system software overhead between the user and the network. Computer scientists at the University of California, Berkeley are using this property of the CM-5 to investigate a model of parallel computation based on what they term a "Threaded Abstract Machine" (TAM). This abstract machine is the compilation target for parallel languages such as ID90. This language, originally designed for dataflow computers, relies on a fine-grained, message-driven execution model. They have taken advantage of direct access to the data network to implement an ultralight form of remote procedure call, called active messages [7].

At the University of Wisconsin, investigators are also using their CM-5 for virtual prototyping; that is, they use the machine to simulate other architectures to better under-



stand alternative architectural choices. The CM-5 Wisconsin Wind Tunnel prototyping system mimics the operations of a large-scale, cache-coherent, shared-memory machine. Almost all application instructions are directly executed by the hardware. References to shared data that

are not locally cached, invoke communication to retrieve memory pages from the global address space. The scalability and flexibility of the CM-5 has allowed these researchers to run full-sized applications on a simulated architecture and study the effects of various cache coherency

schemes [11].

Some of the applications of the CM-5 really require even greater computing power than is available today. For example, a consortium of particle physicists has formed to construct a special version of the CM-5 for teraflops calculations in quantum

Physical memory location	Addressing	
	Multiple	Shared
Distributed	Cosmic Cube IPSC/i860 nCUBE 2	CM-2, CM-5 IBM RP3 Cedar
Centralized		IBM 3090-600 Multimax YMP-8

	Instruction stream	
	Single	Multiple
Globally synchronous	Classic SIMD	CM-5
Pair-wise synchronous		Classic MIMD

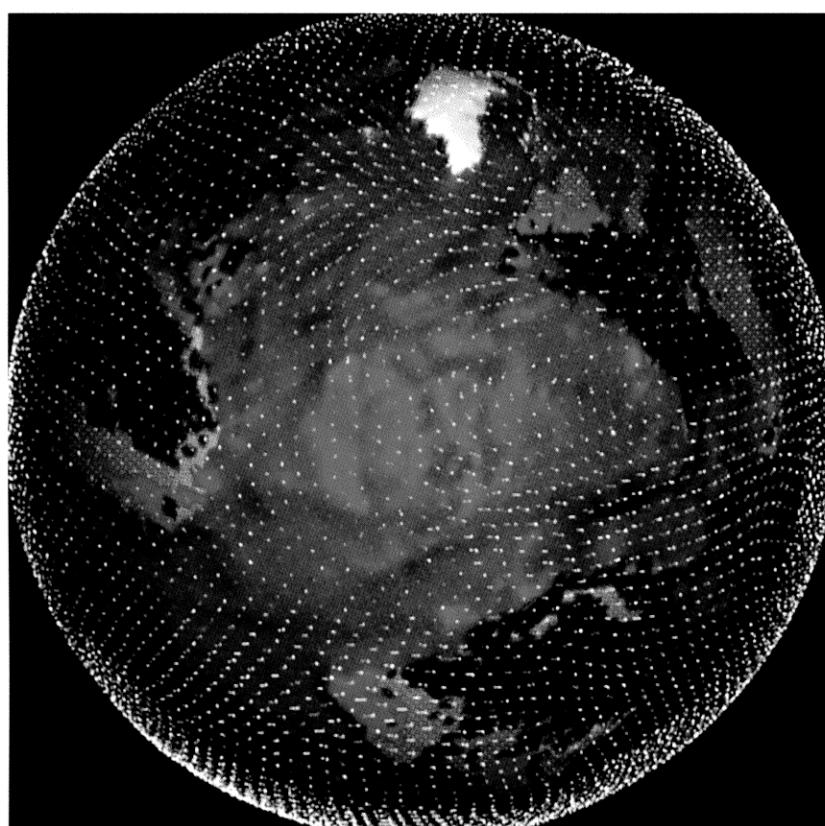


Figure 6.

Parallel processors according to centralized versus distributed memory and shared vs. multiple addressing

Source: Patterson and Hennessy

The hardware notion of shared memory is often confused with the software notion of shared addressing. Machines with locally distributed memory can be designed to support a program model on which the user sees all data in a single address space. This allows the machine to combine the hardware efficiency of distributed memory with the software convenience of a single address space.

Figure 7.

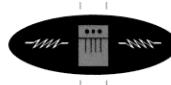
Instruction stream

In a classical SIMD machine, global synchronization is provided by a single broadcast instruction stream. In a classical MIMD machine, multiple instruction streams are synchronized via pairwise interactions; for example, through messages or shared memory references. The CM-5 is an example of a coordinated MIMD machine, in which multiple instruction streams are synchronized whenever necessary through global synchronization hardware.

Figure 8.

X3D—Massively parallel adaptive advanced climate model

This global climate model simulation is an example of a scientific application of the CM-5. Lines represent an irregular computational mesh of approximately 1,200,000 tetrahedra and is used to compute Lagrangian hydrodynamics, Monte Carlo Transport, and thermal heat diffusion. Land masses are shown in yellow. The goal of this application is to better answer fundamental questions about changes in the earth's global climate through computer simulation. Credit: Harold Trease (Los Alamos National Laboratory)



chromodynamics. These calculations could potentially advance our understanding of the structure of atomic particles [1]. Protein structure prediction and global climate modeling are other examples of problems that will ultimately require teraflops performance. These calculations have tremendously large potential economic impact. As a final example of an application that will require even greater performance, one of us (W.D. Hillis) is using a Connection Machine to design computer algorithms by a process analogous to biological evolution [13]. Perhaps some future version of the Connection Machine will use circuits designed by these evolutionary methods.

Summary

In summary, we believe the most important features of the Connection Machine are its scalability, its global access of locally distributed memory, and its global coordination of local executing processes. Together, these features allow the CM-5 to support a variety of parallel programming models and a wide range of applications with good performance. Because the cost of this universality is relatively low, we predict that the architectures of this type, using coordinated homogeneous arrays of RISC microprocessors (CHARM), will be the dominant form of large-scale computing in the 1990s. 

References

1. Aoki, A., et al. Physics goals of the QCD Teraflop project. *J. Mod. Phys. C*, 2, 4 (1991), 892–947.
2. Backus, J. Can programming be liberated from the von Neumann style? *Commun. ACM* 8 (1978).
3. Blelloch, G. Scan primitives and parallel vector models. Ph.D. thesis, Carnegie-Mellon University, Jan. 1989.
4. Boghosian, B. Computational physics on the Connection Machine. *Comput. Phys.* 4, 1 (1990).
5. Dongarra, J.J. Performance of various computers using standard linear equations software. University of Tennessee, Sept. 1992.
6. Dongarra, J.J. TOP500. Tech. Rep. 37831, Computer Science dept., University of Tennessee, July 1993.
7. Eicken, T., Culler, D., Goldstein, S., and Schausler, K. Active Messages: A mechanism for integrated communication and computation. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*. ACM Press (May 1992).
8. Fox, G.C. Parallel computing comes of age: Supercomputer level parallel computations at Caltech. *Concurrency*, 1, 1 (Sept. 1982).
9. Fox, G.C. Achievements and prospects for parallel computing. *Concurrency: Pract. Exp.*, 3, (1991), 725.
10. Hennessy, J.L. and Patterson, D.A. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1990.
11. Hill, M., Larus, J., Reinhardt, S. and Wood, D. Cooperative shared memory: Software and hardware for scalable multiprocessors. Tech. Rep. 1096, Computer Science Dept., University of Wisconsin-Madison, July 1992.
12. Hillis, W.D. Balancing a Design. *IEEE Spectrum* (May 1987).
13. Hillis, W.D. Co-Evolving parasites improve simulated evolution as an optimization procedure. *Physica D* 42 (1990).
14. Hillis, W.D. and Steele, G.L. Data parallel algorithms. *Commun. ACM* 29, 12 (Dec. 1986).
15. Leiserson, C.E. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput. C-34*, 10 (Oct. 1985).
16. Leiserson, C.E., et al. The network architecture of the connection machine CM-5. In the *Fourth Annual ACM Symposium on Parallel Algorithms and Architecture* (June 1992).
17. Long, L.N., Kamon, M., Chyczewski, T.S. and Myczkowski, J. A deterministic parallel algorithm to solve a model Boltzmann equation (BGK). *Comput. Syst. Eng.* 3, 1–4, (Dec. 1992), 337–345.
18. Long, L.N., Kamon, M., Myczkowski, J. A massively parallel algorithm to solve the Boltzmann (BGK) equation. AIAA Rep. 92-0563, Jan. 1992.
19. Sabot, G., Tennies, L., Vasilevsky, A. and Shapiro, R. In *Scientific Applications of the Connection Machine*, H.D. Simon, Ed. World Scientific, River Edge, N.J., Second ed., 1992, pp. 364–378.
20. Schwartz, J. Ultracomputers. *ACM Trans. Program. Lang. Syst.* 2, 4 (Oct. 1980).
21. Seitz, C.L. The cosmic cube. *Commun. ACM* 28, 1 (Jan. 1985).
22. Thinking Machines Corporation. CM-5 Software Sum., CMost Version 7.1, Jan. 1992.
23. Thinking Machines Corporation.

The Connection Machine CM-5 Tech. Sum. (Oct. 1991).

24. Wade, J. The vector coprocessor unit (VU) for the CM-5. Symposium Record: Hot Chips IV, Stanford University, IEEE Computer Society, August 1992.

CR Categories and Subject Descriptors: C.1.2 [Processor Architectures]: Multiple Data Stream Architectures—connection machines; C.5.1 [Computer System Implementation]: Large and Medium (“Mainframe”) Computers—super (very large) computers

General Terms: Design, Performance

Additional Key Words and Phrases: CM-5, Massively Parallel Systems

About the Authors:

W. DANIEL HILLIS is chief scientist and cofounder of Thinking Machines Corp. He is the architect of the Connection Machine computer. His research interests include parallel programming, evolutionary biology, computer architecture, and the study of complex dynamical systems with emergent behavior. His current research is on evolution and parallel learning algorithms.

LEWIS W. TUCKER is director of programming models at Thinking Machines Corp., and is responsible for the CM-5's message passing and scientific visualization library development. His research interests include parallel architecture design, scientific visualization, and computer vision.

Authors' Present Address: Thinking Machines Corp., 245 First Street, Cambridge, MA 02142; email: {danny, tucker}@think.com

Connection Machine is a registered trademark of Thinking Machines Corp.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

A PROCESSOR ARCHITECTURE FOR HORIZON

Mark R. Thistle

Institute for Defense Analyses
Supercomputing Research Center
Lanham, Maryland 20706
thistle@super.org

Burton J. Smith †

Tera Computer Company
P. O. Box 25418
Washington, DC 20007-8418
burton@ll-crg.llnl.gov

ABSTRACT

Horizon is a scalable shared-memory Multiple Instruction stream - Multiple Data stream (MIMD) computer architecture independently under study at the Supercomputing Research Center (SRC) and Tera Computer Company. It is composed of a few hundred identical scalar processors and a comparable number of memories, sparsely embedded in a three-dimensional nearest-neighbor network. Each processor has a horizontal instruction set that can issue up to three floating point operations per cycle without resorting to vector operations. Processors will each be capable of performing several hundred Million Floating Point Operations Per Second (FLOPS) in order to achieve an overall system performance target of 100 Billion (10^{11}) FLOPS.

This paper describes the architecture of the processor in the Horizon system. In the fashion of the Denelcor HEP, the processor maintains a variable number of Single Instruction stream - Single Data stream (SISD) processes, which are called instruction streams. Memory latency introduced by the large shared memory is hidden by switching context (instruction stream) each machine cycle. The processor functional units are pipelined to achieve high computational throughput rates; however, pipeline dependences are hidden from user code. Hardware mechanisms manage the resources to guarantee anonymity and independence of instruction streams.

1. Introduction

Over the last decade, supercomputer performance has risen by an order of magnitude. Machine cycle times of under 10 nanoseconds have become commonplace. However, this performances comes with a cost. Maximum performance can only be achieved with heavily vectorized codes. In addition, these codes must be tuned precisely to the machine on which they will run in order to achieve this maximum performance, thereby thwarting any effort to generate portable programs. Although non-vectorizable problems may benefit from the scalar performance of supercomputers over that of the more general purpose minisupercomputers, they still do not utilize the full potential of the machine. Consequently, scalar speed is low. A fundamental limit for scalar performance is memory latency for large memories. The problem is exacerbated in multiple processor systems by the physical and logical nonlocality of memory with respect to a processor. Caching techniques have been developed to reduce effective memory latency; however, in shared memory systems, cache coherence is a significant problem [1].

The Horizon supercomputer is a shared memory, Multiple Instruction stream, Multiple Data stream (MIMD) system that bridges the gap between scalar and vector performance. Each processor in the multiple processor system uses internal multistream

parallelism to compensate for the latencies introduced by the large shared memory and the pipeline. The architecture of the processor in Horizon bears a strong resemblance to that of its predecessor: the Denelcor HEP. At the time of this writing, the SRC is completing a one-year study to demonstrate the feasibility of the Horizon architecture, its implementation, and its software approach. This paper describes the architecture of the Horizon processor and highlights many of the features which support the high performance targeted for the machine. Section 2 introduces the programmer's model of the Horizon system and of the individual processors. The details of the architecture are discussed in Section 3, with some of the major unresolved issues in Section 4.

2. Programmer Model

2.1 Global Machine Model

A Horizon machine consists of P processors, where P is currently defined in the range 256 to 1024, that share a memory with an address space of 2^{48} bits. Memory is organized into 64-bit words and is addressed through 48-bit virtual bit addresses, the most significant 42 bits of which form the word address. Loads and stores fetch or store the 64-bit word that contains the bit that was addressed. Associated with each memory word is a six-bit access state composed of a full/empty bit, an indirect bit, and four trap bits numbered 0 to 3. These access state bits can modify the behavior of memory references to the associated location. All communication among processes in the machine is through the shared memory [2]. describes the options available for memory access.

Processors and memory modules are spatially and logically distributed throughout a multidimensional, nearest-neighbor, packet-switched interconnection network. Each processor is individually connected to one node in the network and sends data to or receives data from a memory module by via this node. The current design calls for a ratio of the number of processors to memory modules from 1:1 to 1:2. The response time of the memory request is a function of the addressed module's distance from the processor and the traffic in the network. Simulation indicates that average response times of 50-80 cycles are attainable in a system containing 256 processors and 512 memory modules, with nearly all responses available within 128 cycles. Each machine cycle, the network is capable of receiving a message from every processor and memory module, and delivering a message to every processor and memory module. [3] describes the interconnection network for Horizon.

2.2 Processor Model

The Horizon processor manages a variable number of processes, as in the HEP architecture [4], called instruction streams (i-streams), each of which is an autonomous virtual processor with its own register set, program counter, and associated processor state. Each cycle, the processor issues the next instruction from an i-stream chosen from the set of active i-streams. Instructions from any single i-stream are executed in sequential order. Pipelining of instruction

† Work performed at the Supercomputing Research Center.

execution is hidden from the user to avoid the complexity of managing exposed pipelines. The maximum number of i-streams per processor is 128. There is no sharing of registers or other processor state between i-streams; the processor manages each i-stream's state independently.

The processor has three major execution units: the Memory unit (M-unit), Arithmetic unit (A-unit), and the Control unit (C-unit). The architecture is horizontal; a 64-bit instruction can initiate three operations, an M-unit operation (M-op), an A-unit operation (A-op), and a C-unit operation (C-op), through three independent fields in the instruction. Also part of each instruction is a lookahead field that is used to control instruction overlap. The user-specified lookahead value in an instruction indicates the number of subsequent instructions that may be issued without waiting for the completion of that instruction.

M-ops include loads and stores. A load/store architecture was selected to maximize performance with high-bandwidth register operations. Shorter instructions, less hardware complexity, and better reuse of values are additional benefits of a load/store architecture. Arithmetic, logical, control, test, and miscellaneous operations are performed in the A-unit, C-unit, or both. [5] specifies the complete Horizon Instruction Set.

Synchronization between instruction streams is performed through memory operations that use the full/empty bit in each memory cell. An instruction stream can suspend its execution or can be placed into execution by a single instruction. The allocation of instruction streams to processors is handled jointly by user code and the operating system. The operating system allocates instruction streams in response to a system call. Such a system call requires a significant amount of operating system code to be executed and is therefore used relatively infrequently. Thereafter, the user program activates the allocated instruction streams as they are needed. The activation of an i-stream is accomplished by a single instruction.

Each of the M-, A-, and C-fields consists of an operation code (opcode) and the register operands to be used. An instruction may specify up to 7 register sources and 3 register destinations, or 8 sources and 2 destinations for the execution unit operations. These registers are referred to symbolically as q, r, s, (M unit operation), t, u, v, w, (A unit operation), x, y, z, (C unit operation). Register q is either a source or a destination register for the M-unit operation, t is the destination register for the A-unit operation, and x is the destination register for the C-unit operation. The other registers are sources. The number of register accesses for each instruction is an important parameter in the design and will be discussed later. The several dozen program fragments that have been written for Horizon confirm that the horizontal aspect of the instruction set is well balanced and not overly aggressive. From 1-1/2 to 3 floating point operations per instruction have been achieved for the various test programs. A compiler for a small FORTRAN-like language was written to provide data on the suitability of the Horizon instruction set for code generation; this compiler has been successful in achieving high utilization of all three operation fields [6].

The data formats supported include integer, floating point, bit vector, target, and pointer. Integers are signed, 64-bit two's complement values. Floating point numbers are also 64-bit quantities with a non-standard format, chosen for its dynamic range, compatibility with the integer format for common test and compare operations, and recoverability of results from operations that produce overflowed or underflowed results. The definition of the floating point format and the rules for floating point addition, multiplication, and other operations are given in [7]. Pointers are 64-bit values containing a base address and several bits that control the synchronization, indirect addressing, and trapping properties of memory references.

For each of the 128 possible i-streams in the processor, there are 32 64-bit general-purpose registers. The general-purpose registers can each hold integers, floating-point numbers, portions of bit vectors, targets, or pointers. There are four 64-bit target registers and one Instruction-stream Status Word (ISW). Target registers are used to specify potential branch target addresses. The ISW specifies the current Program Counter (PC) as well as the trap mask and condition vector. There is also a 1024-word processor constant table, a read-only table for commonly-used constants such as pi, e, and bit masks, that is shared among all i-streams in the processor.

3. Processor Architecture

3.1 Introduction

Each Horizon processor has a pipelined architecture capable of sustaining a performance of 400 MFLOPS. The target machine cycle time is 4 nanoseconds. To achieve the targeted performance, Horizon embodies four levels of parallelism. At the top level is the MIMD model with several hundred processors. The next three levels are supported within the processor and include: pipelined MIMD instruction execution, that is concurrent execution of multiple instruction streams in a round-robin fashion; overlapped and pipelined execution of instructions within each stream; and horizontal instructions each performing multiple functions. Compilers have traditionally assumed the burden of managing the last two levels of parallelism. The targeted high performance is, in general, accomplished by effectively managing the parallelism, i.e. rapidly switching context in order to hide the memory latencies introduced by a large shared memory, including distance as well as contention and synchronization delays.

The remainder of this paper describes the architecture of the Horizon processor and examines how the various levels of parallelism are implemented. An overview of the functional block level design will be presented, followed by detailed examinations of the important components of the processor architecture, namely, the register organization, instruction issue sequence, i-stream selection policy and mechanism, and memory interface.

3.2 Functional Unit Model

Each processor has seven main functional components: the three execution units - M, A, and C units; the Instruction Fetch and Issue unit, the I-stream Selection unit, the Instruction Cache and Prefetch unit, and the register set. Figure 1 shows the internal organization of a Horizon processor. The instruction execution logic is completely pipelined so that it can begin executing a new instruction every cycle. Each of the function units is pipelined and requires multiple cycles to complete each instruction. Pipeline lengths will be fixed and will appear identical for all instructions, regardless of their complexity. Fixed-length pipelines allow a simple scheme to be used for register scheduling. A variable pipeline length that depends on the complexity of the instruction being performed is also possible. This scheme adds complexity to i-stream scheduling and creates register bandwidth problems for simultaneous completion of instructions but reduces the number of i-streams needed to keep the machine busy. Implications of these alternatives will be discussed later.

Every cycle, the i-stream selection logic selects an i-stream from which to issue the next instruction. The processor hardware and the compiler together must guarantee that an i-stream is eligible for selection only when all the flow and output dependences on previously issued instructions have been satisfied (there are no control or antidependence problems with previously issued instructions), and the instruction has been prefetched into the instruction buffer. To simplify the instruction issue decision, it is convenient to arrange

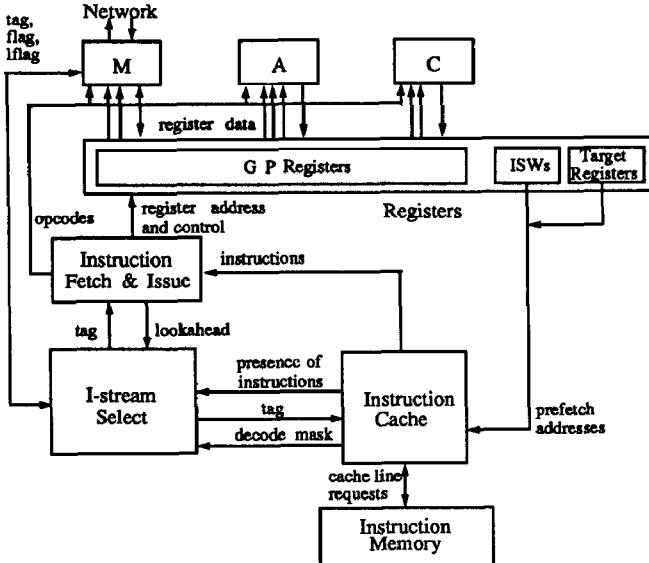


Figure 1. Horizon Processor Functional Architecture

pipeline lengths so that the earliest possible memory response (to the M-unit) is guaranteed to be later than responses from the other function units (A- and C-units). The completion of an instruction is thus made equivalent to the receipt of a response from memory. If no memory operation is specified in the instruction, instruction completion is signaled after the A- and C-units have completed.

After an i-stream has been selected for issue, its next instruction is issued to the execution units. Register operands are delivered to the execution units and execution begins. After a fixed delay, results from the A- and C- units are written back to the registers. Memory responses are essentially asynchronous events. Writing these results back to the registers will be discussed in the next section.

Every cycle the machine context is switched to another i-stream. This technique hides latencies due to memory and network response times and full/empty synchronization delays. As long as there are enough i-streams in the system to fill the pipeline in each processor, the system utilization will approach unity. The result is that the pipeline becomes totally invisible to the programmer. Instructions in a single i-stream seem to complete prior to the issue of the next instruction. The programmer need not be concerned with the number of pipeline segments in each function unit. Thus, pipelined MIMD instruction execution uses parallelism to compensate for lack of locality, and improves the programmability of horizontal (as well as conventional) processors by concealing the pipeline.

With 32 general-purpose registers, 4 target registers, 4 condition codes, the i-stream status word, and other miscellaneous state, the amount of state associated with each i-stream is large. Rather than saving and restoring the state at each instruction cycle, the processor switches context to the appropriate i-stream state. This context switching technique, taken directly from the Denelcor HEP, first appeared in the Control Data 6600 Peripheral Processor unit as the "barrel" and also in the Texas Instruments ASC I/O processor. One set of control logic is time-multiplexed among multiple i-streams. Each i-stream is served at a rate commensurate with its memory latency (or some other limiting function) while the control circuitry and arithmetic pipelines operate at a much higher rate. The maximum number of i-streams per processor 128. Technology considerations prevent it from being more than twice as large as this and

64 is too few to ensure that enough i-streams are ready to execute, given the expected network latency.

Because of the large number of i-streams in a processor, memory traffic generated by instruction fetch is considerable. Therefore access to instruction memory is made separate from data memory which is accessed by the processor through the interconnection network. Distinct processor ports for data and instruction memory operations are provided. An additional benefit of a separate instruction memory, aside from the increased bandwidth available, is that instruction security (ability to execute but not read or write instructions) may be enhanced. Depending on the speed of instruction access, an instruction cache may or may not be needed. Instruction cache issues are addressed in Section 4.

3.3 Register Organization

Each processor has one set of 32 general-purpose registers allocated to each of its i-streams. Because the number of registers per i-stream multiplied by the number of i-streams is large, it is impractical to implement the registers with individual latches or flip-flops. Instead, memory circuits must be used, thereby limiting the cycle time of the processor to some integral multiple of the register memory cycle time. This multiple depends on the number of register accesses in an instruction. Although it is possible to exchange space for time to some extent by implementing several copies of the registers behind the scenes, this approach is costly for horizontal instructions with many register references in each instruction.

The processor can achieve one instruction issue per register memory cycle while allowing a large number of register accesses per instruction by partitioning the register memory into banks. All of the registers for a subset of the processor's i-streams are resident in a single bank. In this scheme, an instruction reads and writes its registers one after another, using several cycles of the register memory bank. Other instructions that were started in the immediate past are meanwhile reading and writing their own registers which are located in other register memory banks. An i-stream becomes a candidate for execution of its next instruction based not only on whether prior instructions have finished but also on whether its register bank is available.

This register banking idea permits the implementation of a single register address space for each instruction rather than the fragmented register address spaces found in conventional horizontal processor designs. In conjunction with pipelined MIMD instruction execution register banking thus solves a problem traditionally associated with code generation for horizontal processors, namely the problem of deciding which registers should hold which values. This problem, together with the difficulty of managing an exposed pipeline with mixed function unit latencies, is largely responsible for much of the bad reputation horizontal machines have as targets for compilers.

The number of registers is a compromise stemming from the number of functional units, the number of source and destination registers needed by each function unit, the length of the instruction, and the register read/write bandwidth available. Each instruction contains up to ten references to registers requiring 50 bits of the 64-bit instruction to specify register names. The remaining 14 bits are used for the M-, A-, and C-unit opcodes and for the lookahead specification. Fewer registers, 16 for example, is not well-balanced with the number used in each instruction and would have led to under-utilized function units or worse -- register spills to memory. More registers, 64 for example, would have required too many bits (assuming fixed size instructions). Given a larger instruction width, 64 registers might be desirable.

Because every Horizon instruction requires either eight register reads and two register writes or seven register reads and three register writes, the sequential register read/write schedule would cause a long instruction cycle. To ease this problem, an identical copy of each register bank is maintained. This allows simultaneous reads to each copy, resulting in two register reads per cycle. A register write places identical data in both bank copies in one cycle. Trading space for time in this way reduces the requirement from eight to four cycles of register reads per instruction. Up to three cycles of register writes for each instruction are required.

With at most seven register cycles used by an instruction, at most seven register banks will be busy every cycle. There should be enough banks compared to the number of register references in an instruction so that there are always i-streams available to execute. If there were on the average only one or two free register banks at any cycle, the number of i-streams needed to keep the processor busy would have to be fairly large compared to the number of banks. To ensure there will always be enough free banks with ready i-streams, the number of register banks currently being considered is 16. Each bank contains the general-purpose registers for 8 i-streams.

This register organization requires a connection-intensive multiplexing scheme to route register accesses from the execution units to the correct register banks. A good implementation of this scheme is critical in order to achieve reasonably short register access latency.

3.4 Instruction Issue

The instruction issue sequence in the Horizon processor requires register bank schedule management similar to the pipeline reservation tables described by [8]. Future register bank cycles are reserved by issuing instructions for reading and writing operands, thereby affecting issue eligibility of other i-streams with registers in the same register bank. This section describes how this scheduling is performed.

When an instruction issues, four consecutive bank access cycles will be consumed reading up to eight source registers. During this time, no other instruction may issue from any i-stream whose registers are resident in that same bank. As the execution units receive their operands, their processing begins. There will be a fixed pipeline delay due to the processing time of the A- and C-units. As the results from each of the A- and C-units emerge from the pipe, they are written to the destination registers. Every cycle one instruction will issue from an i-stream in a different register bank. Instruction issue from a bank is allowed if the next four consecutive bank cycles are free (not previously reserved for register reads or writes). The pipeline schedule for a single instruction issue and execution sequence is illustrated in Figure 2.

The successful completion of a memory reference, unlike operations in the A- and C-units, is an asynchronous event. Memory references must traverse the interconnection network to reach the target address and then return to the source processor. In addition, some memory references may not be satisfied by the first access. For example, in a "wait-for-full-and-set-empty" load, unsuccessful attempts to read a full memory location require that the operation to be retried. Consequently, the register write resulting from a memory load cannot be scheduled at instruction issue time as are those for the results of the A- and C-units.

As memory references are completed, the information about that memory reference (i-stream number, destination register, 64-bit datum, etc.) is enqueued in the M-unit. The M-unit must then steal a cycle from the appropriate register bank to write the datum to the destination register. A bank is considered free for an M-unit result register write if there is no activity scheduled for that cycle. Free bank cycles arise in one of two ways: (1) Future reserved cycles could have prohibited instruction issue from the bank leaving the

next several cycles free. If the bank was eligible for instruction issue but was not selected, there would be at least one free cycle for a register write; (2) An issued instruction might not require all its register read and write slots, thereby leaving free bank cycles.

Experience with the HEP [4] and initial simulation results indicate that a sufficient number of register write cycles are available to avoid starvation. However, to utilize all unused register slots, some decoding of the instruction must be done early. The register read and write cycles that are not needed must be known early enough to steal the free cycles for register writes. Timing is critical only for the first few register read cycles, since more than enough time is available to determine the need for the A- and C-unit write cycles.

3.5 Instruction Stream Selection

The i-stream selection mechanism selects and issues an instruction every clock cycle, as long as there are i-streams ready to issue. This context switching mechanism is performed every cycle and is fundamental for the processor to achieve high performance in a shared memory environment. An i-stream is ready to issue if there are no dependences on previously-issued, unfinished instructions. These dependences are indicated in the lookahead field of the assembly language instruction. The other factors which determine an i-stream's issue eligibility are (1) whether the i-stream's register bank is free for an instruction issue (described in the previous section), and (2) whether the next instruction is available. Every clock cycle, the selection mechanism chooses from among the instruction streams that are candidates for issue. A "fair" selection algorithm will help to ensure that issue starvation does not occur. Other selection mechanisms are being considered that impose issue priorities on streams; however, as yet, none have been formally defined.

The mechanism that manages a stream's dependence information is the lookahead logic. The lookahead value associated with an instruction indicates the number of subsequent instructions that may be issued before it must complete. Instruction completion is defined to be the completion of the memory operation; an instruction with no memory operation completes after the fixed delay of the pipeline. The lookahead logic allows instructions to overlap. Consider an instruction (i) containing a memory reference and a lookahead value of (L). Before instruction (i)'s memory reference is complete, instructions (i+1) through (i+L) may issue. The lookahead field in an instruction is 3 bits wide; therefore, the maximum lookahead value is 7. Hence an i-stream may have a maximum of 8 instructions simultaneously executing.

The maximum lookahead value is related to the instruction word size, the number of destination register references in an instruction and to the size of an i-stream's register set. With 32 registers and 3 destination registers per instruction, 24 registers would be reserved for 8 outstanding memory references (maximum lookahead value of 7). Since a nominal number of registers, say 8, will be live and unusable, a maximum lookahead of 7 seems reasonable. A maximum lookahead of 15, however, would allow all registers to be reserved for memory reference returns. This change would have a

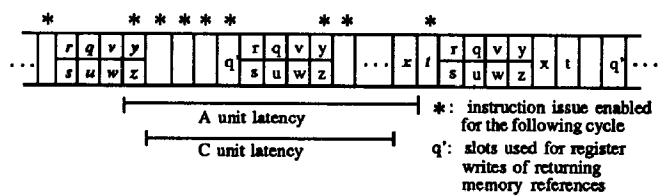


Figure 2. Instruction Execution Sequence

positive effect of reducing the number of i-streams required to keep the processor busy, since a greater memory latency could be tolerated by each i-stream. However, it has the negative effect of doubling the amount of state that must be kept by the M-unit for outstanding memory references, as well as increasing the amount of logic required to manage the additional lookahead and increasing the instruction width. Perhaps more importantly, a greater maximum lookahead also tends to increase the statically scheduled parallelism needed within each instruction stream. Experience with simulation of application codes has not yet provided insight into the realizable benefit from an extended lookahead.

The determination of whether an i-stream is a candidate for issue is performed by the lookahead logic. A functional block diagram of the lookahead mechanism and ready logic is shown in Figure 3. There are 8 locks per i-stream, each corresponding to one of 8 possible instructions awaiting completion. Since there are 128 i-streams per processor, there are a total of 1024 locks in the processor. This defines the maximum number of outstanding memory references allowed from the processor. The value of lock i indicates for how many previous instructions instruction i is waiting. An instruction may not issue until its lock is zero. The current instruction flag counter, flag, points to the lock of the instruction to issue next.

When an instruction issues, the lock associated with the future dependent instruction is incremented. The lock number, lflag, is stored in the M-unit along with the rest of the state describing the memory reference (M-unit) part of that instruction. Lflag indicates which lock is to be decremented when the memory reference completes. Each time an instruction is issued from an i-stream, flag is incremented mod 8. The i-stream is "ready" to issue if the lock addressed by flag is zero.

The inputs to the i-stream selection mechanism are the bank-free signals, the i-stream ready signals described above, and the instruction-present signals. With its current definition, the selection mechanism can be implemented entirely in combinatorial logic, as a two-level, inverted binary tree. On one side, a register bank from among the 16 possible banks is selected to issue an instruction. A register bank is "ready", i.e., is a candidate for this selection, only if there is at least one i-stream in the bank that is ready to issue an instruction and the bank is free. The bank-free bits for each register bank are generated in parallel from the bank schedules. Similarly, the ready bits for all the i-streams are produced in parallel by the lookahead logic. Bank readiness is determined for all free banks in

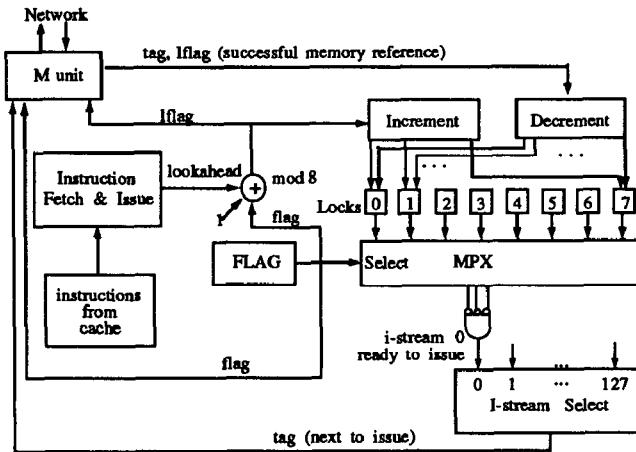


Figure 3. I-stream Lookahead Logic

parallel by OR-ing the i-stream ready bits in each bank. Bank selection is based on a "roving" round-robin algorithm, in which the highest priority rotates among the banks. The round robin selector determines the bank with the highest priority for issue each cycle. With this scheme, the maximum possible delay between instruction issues from a bank with ready i-streams is 16 cycles.

The other side of the tree is the selection of an i-stream from a bank. The same round-robin mechanism as in bank selection is used. The choice of which i-stream will be issued next from each bank is generated in parallel for each bank. The worst-case delay between issues from a particular i-stream (assuming it is ready) is 8 bank issue cycles; that is, 8 cycles in which some i-stream is selected from the particular i-stream's bank. The two sides of the tree are joined by multiplexing down the tag of the chosen i-stream in the chosen bank. This selection algorithm guarantees uniform treatment of banks and of i-streams within a bank. Given the register banking constraints, i-streams in lightly loaded banks will be favored for execution over those in heavily loaded banks. This natural priority may be useful for run time scheduling of multiple priority tasks within a processor.

3.6 Memory Interface

The M-unit directs all memory references from the processor into and out of the interconnection network. Memory requests are generated from instructions, processed after returning successfully from the network, reissued if unsuccessful, and rerouted if destined for another node. (Instruction cache lines are not fetched through the M-unit, since instructions do not travel through the network.) When a load or store instruction is issued, the M-unit receives the tag and flag (Figure 1) of the instruction from the I-stream Selection unit. This information is saved in the M-unit state table, along with the information about the request from the instruction.

The M-unit state table holds the following information: the operation to be performed (opcode), the virtual memory address, the tag and flag of instruction, destination register number, the 64-bit datum, a timestamp, and a retry value. The flag of the instruction is the address of the lock to decrement when the instruction completes. Because the arrival of a successful response from memory is an asynchronous event, some of the responses must be retained until the datum can be written to the registers during a free register bank cycle. Figure 4 shows a functional block diagram of the M-unit components. The 16 queues, one per register bank, are used to buffer completed memory requests until a register write can be performed. When the register write occurs, the datum is written to the destination register of the appropriate i-stream register set (in the case of a load). For either a load or a store, the lock addressed by lflag is decremented, indicating completion of the instruction. Assuming

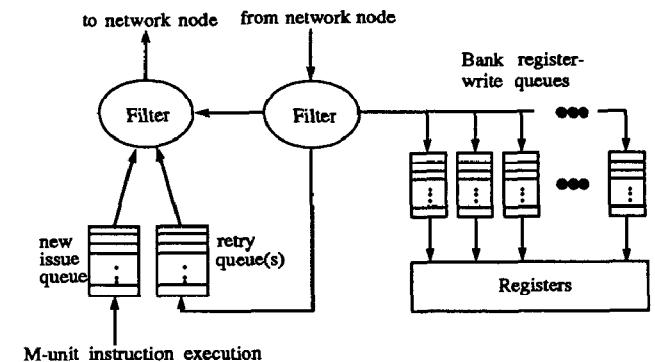


Figure 4. Memory Unit Network Interface

that each M-unit result register write queue is provided a unique data path to the registers and the i-stream selection logic can support parallel access to the locks, up to 16 M-unit result register writes could be performed in any cycle.

If the instruction contains a synchronizing load or store (e.g. wait-for-full), the M-unit must schedule a retry in the event of an unsuccessful request. Simply re-trying every unsuccessful request immediately might quickly flood the interconnection network with requests that would likely be unsuccessful again. (Flow control would limit the extent to which flooding will occur [3].) In addition retries would cause interference with issue of newly-generated requests from issued instructions since there is only a single port to the network. Hence, some back-off policy will be adopted for retry scheduling. Each successive unsuccessful attempt at a memory access will result in a greater delay before another re-try is issued. The back-off policy is further complicated by the possibility that a message may be returned without ever having gained access to its destination memory cell. Such is the case at "hot-spots" where bank conflicts overflow queues at the memory banks. Randomization policies reduce the probability of memory bank conflicts; however, hot spots will inevitably occur, whether because of poor programming practices, algorithmic constraints, or "bad" statistical epochs. Consequently, several classes of retry messages may be required.

The last responsibility of the M-unit is re-routing messages that are not addressed to it. A message may be delivered to a processor if it is not intended for if there are no other links available for that message at the node [3]. Such a message is called a misrouted message. The M-unit must then turn the message around and re-issue it to the network. The filters in Figure 4 perform this function.

As indicated in Figure 4, there are essentially two queues from which messages are inserted into the network. One queue contains messages generated by newly issued instructions and the other contains messages to be retried. An alternating scheduling mechanism of memory references delivered into the network prevents starvation of either queue. Misrouted messages must have priority over either queue for injection into the network, since the processor has no knowledge about nor any reserved storage for such messages. The size of the issue queue is 1024, the maximum number of outstanding memory references allowed from the processor. To avoid deadlock, the retry queue must also be length 1024.

4. Unresolved Issues

4.1 Introduction

As the processor architecture evolves, many new (not yet resolved) issues are identified. Some of these issues are implementation details which become more important in later stages of development. However, there are important architectural topics which have not been addressed in this paper but are noteworthy, nonetheless. Instruction stream creation, traps and exceptions, and instruction memory hierarchy are three such topics. The latter two are addressed below.

4.2 Traps and Exceptions

It is an objective of the trap mechanism in the Horizon processor is to be as lightweight as possible, requiring no operating system intervention. This means that enough information about the machine state at the time of the trap should be accessible to the user to allow identification of the cause and appropriate recovery. Moreover, the integrity of the system must not be compromised by giving unprivileged programs this level of access.

The proposed traps are listed in Table 1. Two levels of masking are provided to the user for maskable traps. Summary trap mask bits are provided in the ISW for the countdown trap, branch trap, all the floating point traps and access state violation traps so that they

Table 1. Traps

CLASS ⁽¹⁾	TRAP	DETECTING UNIT ⁽²⁾
m	Countdown	Iselect
u	Hardware Error	All
m	Single Memory Error	IF, M
u	Double Memory Error	IF, M
u	Memory Protection Violation	IF, M
u	Unimplemented Memory Address	IF, M
m	Access State Violation	M
m	Floating Point Overflow	A, C
m	Floating Point Underflow	A, C
m	Floating Point Indefinite	A, C
m	Floating Point Loss of Significance	A, C
m	Branch	C
u	Privileged Instruction Trap	IF, C
u	Unimplemented Instruction Trap	IF, A, C, M
u	Create Fault Trap	M

⁽¹⁾ m : a user-maskable trap
u : an unmaskable trap

⁽²⁾ A : A-unit
C : C-unit
M : M-unit
IF: Instruction Fetch unit

can be manipulated easily inside target registers. For masking individual types of floating point traps, access state violations, and other traps listed in Table 1, separate mask registers are accessible as part of the processor state.

Because of the pipelined nature of the machine, all instructions, once issued, flow completely through the functional units. When a trap occurs, further instructions from the same i-stream may be prevented from being issued. Some traps are detected very early, even before the offending instruction enters the pipeline. In this case, the instruction(s) already in the pipeline may be completed correctly. However, some traps are not detected until the offending instruction has nearly completed and hence not in time to prevent one or more a subsequent instruction(s) from being issued (if allowed by lookahead). Depending on the nature of the trap, the later instruction(s) are either completed correctly or else their side-effects are inhibited.

Two scenarios were considered for generating traps: trap on generate and trap on use. In the latter scenario, if an error occurred in an operation, the associated destination register would be poisoned for future references, i.e., given a value that when encountered as an operand in a subsequent instruction would cause a trap. This approach has the property that traps are put off as long as possible and in fact may never occur if the offending value is not re-used. Such a scheme is useful for handling unsafe loads, where fetching an operand at the end of a loop for the next iteration causes an invalid address for the last iteration of the loop. Such an error could be ignored if the destination register is not subsequently read before being written again with valid data. However, this mechanism is not sufficient for operations that have no destination register. Specifically, a memory protection violation occurring on a store operation has no destination register value to poison. Another argument in favor of a trap on generate scheme is that a "fatal" trap may not be evidenced until much more (wasted) computation has been done. The potentially large "distance" from the condition which caused the trap to its detection makes debugging more difficult. Consequently, the trap on generate scenario was chosen.

4.3 Instruction Cache

One possible design for an instruction cache for the Horizon processor is simply an instruction prefetch buffer. The buffer should keep at least 6 lines for each active i-stream: one for the PC, one for the PC incremented, and one for each of the 4 targets. When the PC increments or a branch instruction is processed, the next instruction is always ready in the buffer. When an event occurs that changes the set of lines needed for a particular i-stream, the relevant lock is incremented in the lookahead logic for that i-stream. The lock is then decremented when the line is brought into the instruction buffer. With this instruction prefetch mechanism, instruction issue latency is only affected if the instruction prefetch time exceeds the minimum time between successive issues from a single i-stream, which is determined by the dependence on previous instructions and the number of i-streams contending for issue slots.

5. Conclusions

The processor architecture described in this paper combines three levels of parallelism, multiple instruction streams, instruction lookahead, and horizontal instructions to achieve vector-like performance on scalar code. The processor removes the burden of managing resources, hiding the pipeline and register read/write timing, from the compiler, thereby greatly simplifying code generation. Latencies caused by contention or synchronization through the shared memory, creating serious performance degradation in most machines, are effectively hidden by the processor. Each instruction stream is served at a rate commensurate with average operation latency, while the processor sustains a utilization (instructions issued per cycle) approaching unity. There are, however, many unresolved issues to be addressed. From early simulation studies and programming experience on the Horizon simulator, it appears that the underlying processor architecture is well balanced and is capable of achieving the targeted performance.

6. Acknowledgements

The authors gratefully acknowledge the contributions of the architecture team at the Supercomputing Research Center, all of whom share primary roles in the conceptual development and design of Horizon: Paul B. Schneck, Chief Architect of Horizon at the SRC, and James T. Kuehn, both of whom contributed to and provided consultation for this writing, William E. Holmes, Daniel J. Kopetzky, Fred A. More, and David L. Smitley. Steven Melvin is recognized for the origination of the register banking scheme developed originally for the HEP-3. The authors also recognize all those in industry and academia who contributed in no small part to the genesis and evolution of the Horizon processor architecture.

7. References

- [1] M. Dubois, C. Scheurich, and F. A. Briggs, "Synchronization, Coherence, and Event Ordering in Multiprocessors," *IEEE Computer*, Vol. 21, No. 2, February 1988, pp 9-21.
- [2] J. T. Kuehn and B. J. Smith, "The Horizon Supercomputing System: Architecture and Software," *Supercomputing '88*, submitted for publication, 1988.
- [3] F. M. Pittelli and D. L. Smitley, "Analysis of a 3-D Toroidal Network for a Shared Memory Architecture," *Supercomputing '88*, submitted for publication, 1988.
- [4] B. J. Smith, "A Pipelined Shared Resource MIMD Computer," *1978 International Conference on Parallel Processing*, 1978.
- [5] B. J. Smith, "The Horizon Instruction Set," unpublished note, Supercomputing Research Center, 1987.
- [6] J. Draper, "Compiling on Horizon," *Supercomputing '88*, submitted for publication, 1988.
- [7] M. R. Thistle, "Floating Point Arithmetic on Horizon," Internal Research Note, Supercomputing Research Center, 1988.
- [8] P. M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill, NY, 1981.

Hyper-Threading Technology Architecture and Microarchitecture

Deborah T. Marr, Desktop Products Group, Intel Corp.

Frank Binns, Desktop Products Group, Intel Corp.

David L. Hill, Desktop Products Group, Intel Corp.

Glenn Hinton, Desktop Products Group, Intel Corp.

David A. Koufaty, Desktop Products Group, Intel Corp.

J. Alan Miller, Desktop Products Group, Intel Corp.

Michael Upton, CPU Architecture, Desktop Products Group, Intel Corp.

Index words: architecture, microarchitecture, Hyper-Threading Technology, simultaneous multi-threading, multiprocessor

ABSTRACT

Intel's Hyper-Threading Technology brings the concept of simultaneous multi-threading to the Intel Architecture. Hyper-Threading Technology makes a single physical processor appear as two logical processors; the physical execution resources are shared and the architecture state is duplicated for the two logical processors. From a software or architecture perspective, this means operating systems and user programs can schedule processes or threads to logical processors as they would on multiple physical processors. From a microarchitecture perspective, this means that instructions from both logical processors will persist and execute simultaneously on shared execution resources.

This paper describes the Hyper-Threading Technology architecture, and discusses the microarchitecture details of Intel's first implementation on the Intel® Xeon™ processor family. Hyper-Threading Technology is an important addition to Intel's enterprise product line and will be integrated into a wide variety of products.

INTRODUCTION

The amazing growth of the Internet and telecommunications is powered by ever-faster systems demanding increasingly higher levels of processor performance. To keep up with this demand we cannot rely entirely on traditional approaches to processor design. Microarchitecture techniques used to achieve past processor performance improvement—super-pipelining, branch prediction, super-scalar execution, out-of-order execution, caches—have made microprocessors increasingly more complex, have more transistors, and consume more power. In fact, transistor counts and power are increasing at rates greater than processor performance. Processor architects are therefore looking for ways to improve performance at a greater rate than transistor counts and power dissipation. Intel's Hyper-Threading Technology is one solution.

Processor Microarchitecture

Traditional approaches to processor design have focused on higher clock speeds, instruction-level parallelism (ILP), and caches. Techniques to achieve higher clock speeds involve pipelining the microarchitecture to finer granularities, also called super-pipelining. Higher clock frequencies can greatly improve performance by increasing the number of instructions that can be executed each second. Because there will be far more instructions in-flight in a super-pipelined microarchitecture, handling of events that disrupt the pipeline, e.g., cache misses, interrupts and branch mispredictions, can be costly.

[®]Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

[™]Xeon is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

ILP refers to techniques to increase the number of instructions executed each clock cycle. For example, a super-scalar processor has multiple parallel execution units that can process instructions simultaneously. With super-scalar execution, several instructions can be executed each clock cycle. However, with simple in-order execution, it is not enough to simply have multiple execution units. The challenge is to find enough instructions to execute. One technique is out-of-order execution where a large window of instructions is simultaneously evaluated and sent to execution units, based on instruction dependencies rather than program order.

Accesses to DRAM memory are slow compared to execution speeds of the processor. One technique to reduce this latency is to add fast caches close to the processor. Caches can provide fast memory access to frequently accessed data or instructions. However, caches can only be fast when they are small. For this reason, processors often are designed with a cache hierarchy in which fast, small caches are located and operated at access latencies very close to that of the processor core, and progressively larger caches, which handle less frequently accessed data or instructions, are implemented with longer access latencies. However, there will always be times when the data needed will not be in any processor cache. Handling such cache misses requires accessing memory, and the processor is likely to quickly run out of instructions to execute before stalling on the cache miss.

The vast majority of techniques to improve processor performance from one generation to the next is complex and often adds significant die-size and power costs. These techniques increase performance but not with 100% efficiency; i.e., doubling the number of execution units in a processor does not double the performance of the processor, due to limited parallelism in instruction flows. Similarly, simply doubling the clock rate does not double the performance due to the number of processor cycles lost to branch mispredictions.

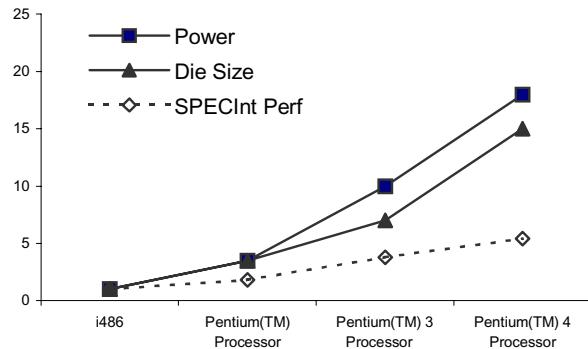


Figure 1: Single-stream performance vs. cost

Figure 1 shows the relative increase in performance and the costs, such as die size and power, over the last ten years on Intel processors¹. In order to isolate the microarchitecture impact, this comparison assumes that the four generations of processors are on the same silicon process technology and that the speed-ups are normalized to the performance of an Intel486™ processor. Although we use Intel's processor history in this example, other high-performance processor manufacturers during this time period would have similar trends. Intel's processor performance, due to microarchitecture advances alone, has improved integer performance five- or six-fold¹. Most integer applications have limited ILP and the instruction flow can be hard to predict.

Over the same period, the relative die size has gone up fifteen-fold, a three-times-higher rate than the gains in integer performance. Fortunately, advances in silicon process technology allow more transistors to be packed into a given amount of die area so that the actual measured die size of each generation microarchitecture has not increased significantly.

The relative power increased almost eighteen-fold during this period¹. Fortunately, there exist a number of known techniques to significantly reduce power consumption on processors and there is much on-going research in this area. However, current processor power dissipation is at the limit of what can be easily dealt with in desktop platforms and we must put greater emphasis on improving performance in conjunction with new technology, specifically to control power.

¹ These data are approximate and are intended only to show trends, not actual performance.

TM Intel486 is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Thread-Level Parallelism

A look at today's software trends reveals that server applications consist of multiple threads or processes that can be executed in parallel. On-line transaction processing and Web services have an abundance of software threads that can be executed simultaneously for faster performance. Even desktop applications are becoming increasingly parallel. Intel architects have been trying to leverage this so-called thread-level parallelism (TLP) to gain a better performance vs. transistor count and power ratio.

In both the high-end and mid-range server markets, multiprocessors have been commonly used to get more performance from the system. By adding more processors, applications potentially get substantial performance improvement by executing multiple threads on multiple processors at the same time. These threads might be from the same application, from different applications running simultaneously, from operating system services, or from operating system threads doing background maintenance. Multiprocessor systems have been used for many years, and high-end programmers are familiar with the techniques to exploit multiprocessors for higher performance levels.

In recent years a number of other techniques to further exploit TLP have been discussed and some products have been announced. One of these techniques is chip multiprocessing (CMP), where two processors are put on a single die. The two processors each have a full set of execution and architectural resources. The processors may or may not share a large on-chip cache. CMP is largely orthogonal to conventional multiprocessor systems, as you can have multiple CMP processors in a multiprocessor configuration. Recently announced processors incorporate two processors on each die. However, a CMP chip is significantly larger than the size of a single-core chip and therefore more expensive to manufacture; moreover, it does not begin to address the die size and power considerations.

Another approach is to allow a single processor to execute multiple threads by switching between them. Time-slice multithreading is where the processor switches between software threads after a fixed time period. Time-slice multithreading can result in wasted execution slots but can effectively minimize the effects of long latencies to memory. Switch-on-event multithreading would switch threads on long latency events such as cache misses. This approach can work well for server applications that have large numbers of cache misses and where the two threads are executing similar tasks. However, both the time-slice and the switch-on-

event multi-threading techniques do not achieve optimal overlap of many sources of inefficient resource usage, such as branch mispredictions, instruction dependencies, etc.

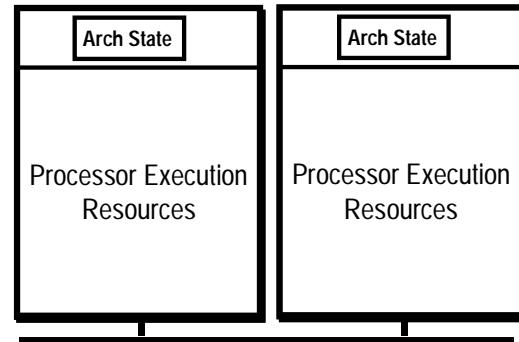
Finally, there is simultaneous multi-threading, where multiple threads can execute on a single processor without switching. The threads execute simultaneously and make much better use of the resources. This approach makes the most effective use of processor resources: it maximizes the performance vs. transistor count and power consumption.

Hyper-Threading Technology brings the simultaneous multi-threading approach to the Intel architecture. In this paper we discuss the architecture and the first implementation of Hyper-Threading Technology on the Intel® Xeon™ processor family.

HYPER-THREADING TECHNOLOGY ARCHITECTURE

Hyper-Threading Technology makes a single physical processor appear as multiple logical processors [11, 12]. To do this, there is one copy of the architecture state for each logical processor, and the logical processors share a single set of physical execution resources. From a software or architecture perspective, this means operating systems and user programs can schedule processes or threads to logical processors as they would on conventional physical processors in a multiprocessor system. From a microarchitecture perspective, this means that instructions from logical processors will persist and execute simultaneously on shared execution resources.

Figure 2: Processors without Hyper-Threading Tech



[®]Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

[™]Xeon is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

As an example, Figure 2 shows a multiprocessor system with two physical processors that are not Hyper-Threading Technology-capable. Figure 3 shows a multiprocessor system with two physical processors that are Hyper-Threading Technology-capable. With two copies of the architectural state on each physical processor, the system appears to have four logical processors.

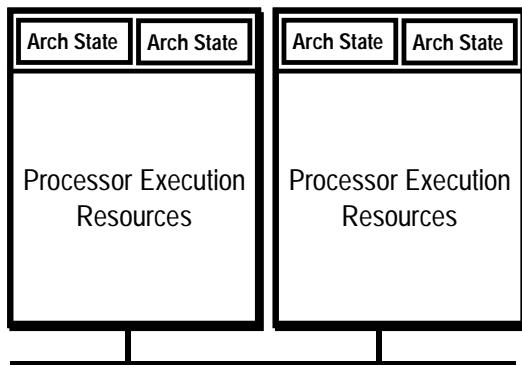


Figure 3: Processors with Hyper-Threading Technology

The first implementation of Hyper-Threading Technology is being made available on the Intel® Xeon™ processor family for dual and multiprocessor servers, with two logical processors per physical processor. By more efficiently using existing processor resources, the Intel Xeon processor family can significantly improve performance at virtually the same system cost. This implementation of Hyper-Threading Technology added less than 5% to the relative chip size and maximum power requirements, but can provide performance benefits much greater than that.

Each logical processor maintains a complete set of the architecture state. The architecture state consists of registers including the general-purpose registers, the control registers, the advanced programmable interrupt controller (APIC) registers, and some machine state registers. From a software perspective, once the architecture state is duplicated, the processor appears to be two processors. The number of transistors to store the architecture state is an extremely small fraction of the total. Logical processors share nearly all other resources on the physical processor, such as caches,

[®] Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

[™] Xeon is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

execution units, branch predictors, control logic, and buses.

Each logical processor has its own interrupt controller or APIC. Interrupts sent to a specific logical processor are handled only by that logical processor.

FIRST IMPLEMENTATION ON THE INTEL XEON PROCESSOR FAMILY

Several goals were at the heart of the microarchitecture design choices made for the Intel® Xeon™ processor MP implementation of Hyper-Threading Technology. One goal was to minimize the die area cost of implementing Hyper-Threading Technology. Since the logical processors share the vast majority of microarchitecture resources and only a few small structures were replicated, the die area cost of the first implementation was less than 5% of the total die area.

A second goal was to ensure that when one logical processor is stalled the other logical processor could continue to make forward progress. A logical processor may be temporarily stalled for a variety of reasons, including servicing cache misses, handling branch mispredictions, or waiting for the results of previous instructions. Independent forward progress was ensured by managing buffering queues such that no logical processor can use all the entries when two active software threads² were executing. This is accomplished by either partitioning or limiting the number of active entries each thread can have.

A third goal was to allow a processor running only one active software thread to run at the same speed on a processor with Hyper-Threading Technology as on a processor without this capability. This means that partitioned resources should be recombined when only one software thread is active. A high-level view of the microarchitecture pipeline is shown in Figure 4. As shown, buffering queues separate major pipeline logic blocks. The buffering queues are either partitioned or duplicated to ensure independent forward progress through each logic block.

[®] Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

[™] Xeon is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

² Active software threads include the operating system idle loop because it runs a sequence of code that continuously checks the work queue(s). The operating system idle loop can consume considerable execution resources.

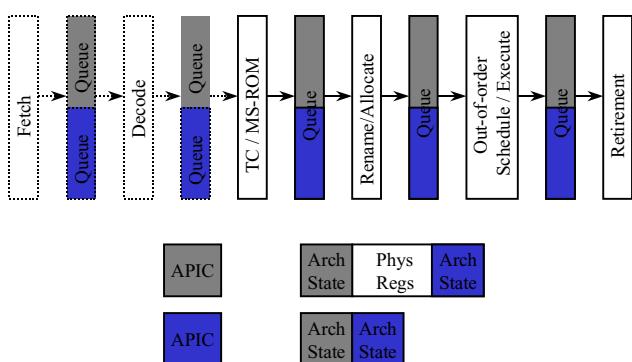


Figure 4 Intel® Xeon™ processor pipeline

In the following sections we will walk through the pipeline, discuss the implementation of major functions, and detail several ways resources are shared or replicated.

FRONT END

The front end of the pipeline is responsible for delivering instructions to the later pipe stages. As shown in Figure 5a, instructions generally come from the Execution Trace Cache (TC), which is the primary or Level 1 (L1) instruction cache. Figure 5b shows that only when there is a TC miss does the machine fetch and decode instructions from the integrated Level 2 (L2) cache. Near the TC is the Microcode ROM, which stores decoded instructions for the longer and more complex IA-32 instructions.

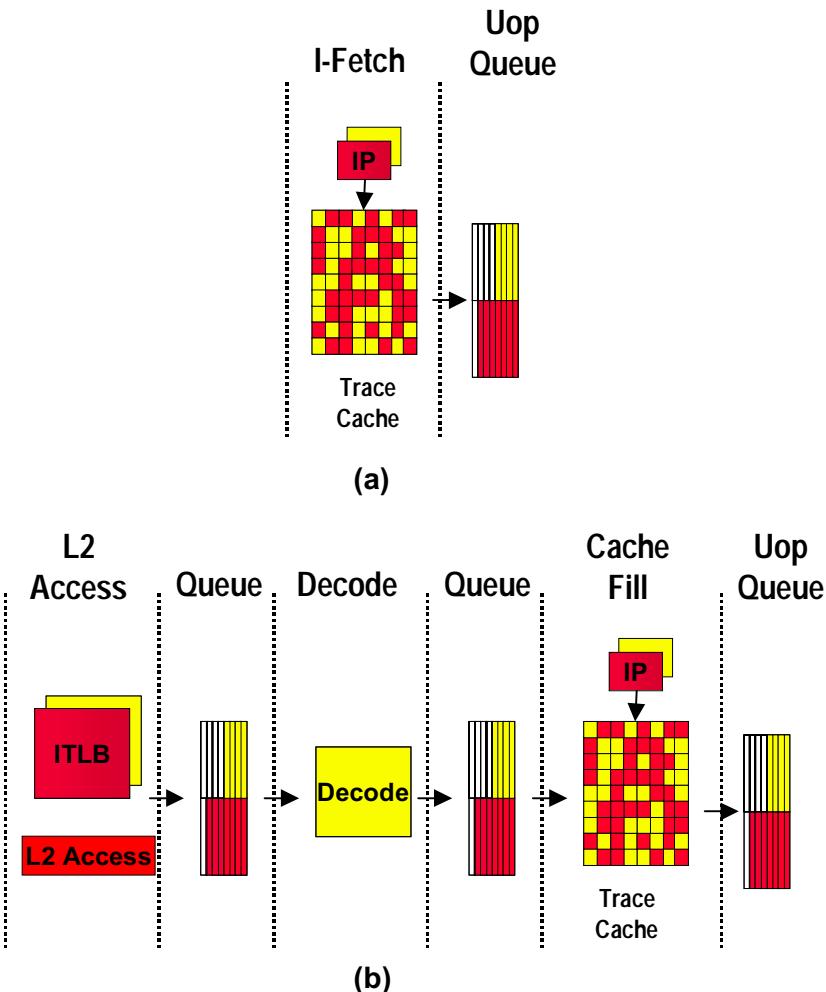


Figure 5: Front-end detailed pipeline (a) Trace Cache Hit (b) Trace Cache Miss

Execution Trace Cache (TC)

The TC stores decoded instructions, called microoperations or “uops.” Most instructions in a program are fetched and executed from the TC. Two sets of next-instruction-pointers independently track the progress of the two software threads executing. The two logical processors arbitrate access to the TC every clock cycle. If both logical processors want access to the TC at the same time, access is granted to one then the other in alternating clock cycles. For example, if one cycle is used to fetch a line for one logical processor, the next cycle would be used to fetch a line for the other logical processor, provided that both logical processors requested access to the trace cache. If one logical processor is stalled or is unable to use the TC, the other logical processor can use the full bandwidth of the trace cache, every cycle.

The TC entries are tagged with thread information and are dynamically allocated as needed. The TC is 8-way set associative, and entries are replaced based on a least-recently-used (LRU) algorithm that is based on the full 8 ways. The shared nature of the TC allows one logical processor to have more entries than the other if needed.

Microcode ROM

When a complex instruction is encountered, the TC sends a microcode-instruction pointer to the Microcode ROM. The Microcode ROM controller then fetches the uops needed and returns control to the TC. Two microcode instruction pointers are used to control the flows independently if both logical processors are executing complex IA-32 instructions.

Both logical processors share the Microcode ROM entries. Access to the Microcode ROM alternates between logical processors just as in the TC.

ITLB and Branch Prediction

If there is a TC miss, then instruction bytes need to be fetched from the L2 cache and decoded into uops to be placed in the TC. The Instruction Translation Lookaside Buffer (ITLB) receives the request from the TC to deliver new instructions, and it translates the next-instruction pointer address to a physical address. A request is sent to the L2 cache, and instruction bytes are returned. These bytes are placed into streaming buffers, which hold the bytes until they can be decoded.

The ITLBs are duplicated. Each logical processor has its own ITLB and its own set of instruction pointers to track the progress of instruction fetch for the two logical processors. The instruction fetch logic in charge of sending requests to the L2 cache arbitrates on a first-

come first-served basis, while always reserving at least one request slot for each logical processor. In this way, both logical processors can have fetches pending simultaneously.

Each logical processor has its own set of two 64-byte streaming buffers to hold instruction bytes in preparation for the instruction decode stage. The ITLBs and the streaming buffers are small structures, so the die size cost of duplicating these structures is very low.

The branch prediction structures are either duplicated or shared. The return stack buffer, which predicts the target of return instructions, is duplicated because it is a very small structure and the call/return pairs are better predicted for software threads independently. The branch history buffer used to look up the global history array is also tracked independently for each logical processor. However, the large global history array is a shared structure with entries that are tagged with a logical processor ID.

IA-32 Instruction Decode

IA-32 instructions are cumbersome to decode because the instructions have a variable number of bytes and have many different options. A significant amount of logic and intermediate state is needed to decode these instructions. Fortunately, the TC provides most of the uops, and decoding is only needed for instructions that miss the TC.

The decode logic takes instruction bytes from the streaming buffers and decodes them into uops. When both threads are decoding instructions simultaneously, the streaming buffers alternate between threads so that both threads share the same decoder logic. The decode logic has to keep two copies of all the state needed to decode IA-32 instructions for the two logical processors even though it only decodes instructions for one logical processor at a time. In general, several instructions are decoded for one logical processor before switching to the other logical processor. The decision to do a coarser level of granularity in switching between logical processors was made in the interest of die size and to reduce complexity. Of course, if only one logical processor needs the decode logic, the full decode bandwidth is dedicated to that logical processor. The decoded instructions are written into the TC and forwarded to the uop queue.

Uop Queue

After uops are fetched from the trace cache or the Microcode ROM, or forwarded from the instruction decode logic, they are placed in a “uop queue.” This queue decouples the Front End from the Out-of-order

Execution Engine in the pipeline flow. The uop queue is partitioned such that each logical processor has half the entries. This partitioning allows both logical processors to make independent forward progress regardless of front-end stalls (e.g., TC miss) or execution stalls.

OUT-OF-ORDER EXECUTION ENGINE

The out-of-order execution engine consists of the allocation, register renaming, scheduling, and execution functions, as shown in Figure 6. This part of the machine re-orders instructions and executes them as

quickly as their inputs are ready, without regard to the original program order.

Allocator

The out-of-order execution engine has several buffers to perform its re-ordering, tracing, and sequencing operations. The allocator logic takes uops from the uop queue and allocates many of the key machine buffers needed to execute each uop, including the 126 re-order buffer entries, 128 integer and 128 floating-point physical registers, 48 load and 24 store buffer entries. Some of these key buffers are partitioned such that each logical processor can use at most half the entries.

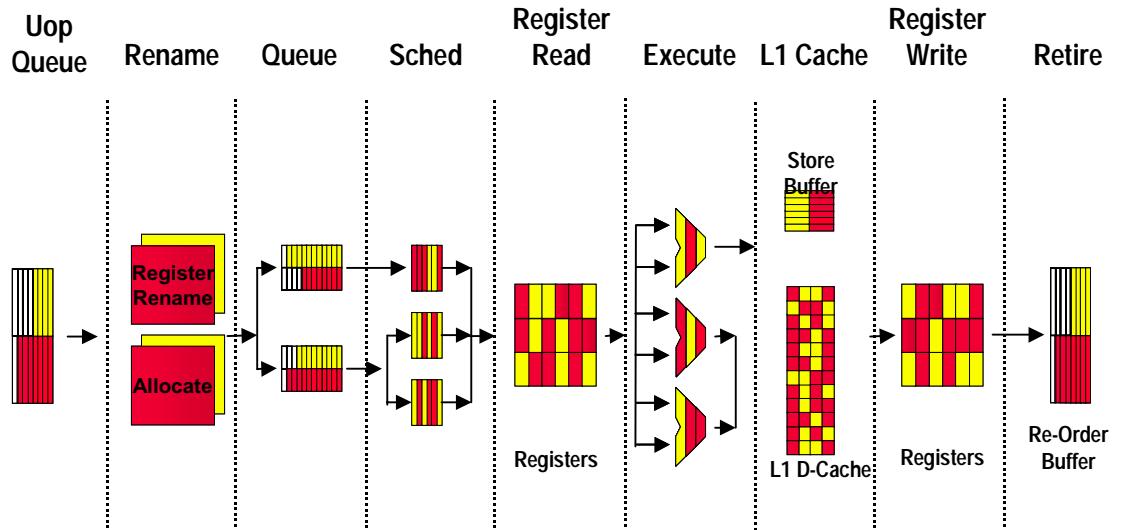


Figure 6: Out-of-order execution engine detailed pipeline

Specifically, each logical processor can use up to a maximum of 63 re-order buffer entries, 24 load buffers, and 12 store buffer entries.

If there are uops for both logical processors in the uop queue, the allocator will alternate selecting uops from the logical processors every clock cycle to assign resources. If a logical processor has used its limit of a needed resource, such as store buffer entries, the allocator will signal “stall” for that logical processor and continue to assign resources for the other logical processor. In addition, if the uop queue only contains uops for one logical processor, the allocator will try to assign resources for that logical processor every cycle to optimize allocation bandwidth, though the resource limits would still be enforced.

By limiting the maximum resource usage of key buffers, the machine helps enforce fairness and prevents deadlocks.

Register Rename

The register rename logic renames the architectural IA-32 registers onto the machine’s physical registers. This allows the 8 general-use IA-32 integer registers to be dynamically expanded to use the available 128 physical registers. The renaming logic uses a Register Alias Table (RAT) to track the latest version of each architectural register to tell the next instruction(s) where to get its input operands.

Since each logical processor must maintain and track its own complete architecture state, there are two RATs, one for each logical processor. The register renaming process is done in parallel to the allocator logic described above, so the register rename logic works on the same uops to which the allocator is assigning resources.

Once uops have completed the allocation and register rename processes, they are placed into two sets of

queues, one for memory operations (loads and stores) and another for all other operations. The two sets of queues are called the memory instruction queue and the general instruction queue, respectively. The two sets of queues are also partitioned such that uops from each logical processor can use at most half the entries.

Instruction Scheduling

The schedulers are at the heart of the out-of-order execution engine. Five uop schedulers are used to schedule different types of uops for the various execution units. Collectively, they can dispatch up to six uops each clock cycle. The schedulers determine when uops are ready to execute based on the readiness of their dependent input register operands and the availability of the execution unit resources.

The memory instruction queue and general instruction queues send uops to the five scheduler queues as fast as they can, alternating between uops for the two logical processors every clock cycle, as needed.

Each scheduler has its own scheduler queue of eight to twelve entries from which it selects uops to send to the execution units. The schedulers choose uops regardless of whether they belong to one logical processor or the other. The schedulers are effectively oblivious to logical processor distinctions. The uops are simply evaluated based on dependent inputs and availability of execution resources. For example, the schedulers could dispatch two uops from one logical processor and two uops from the other logical processor in the same clock cycle. To avoid deadlock and ensure fairness, there is a limit on the number of active entries that a logical processor can have in each scheduler's queue. This limit is dependent on the size of the scheduler queue.

Execution Units

The execution core and memory hierarchy are also largely oblivious to logical processors. Since the source and destination registers were renamed earlier to physical registers in a shared physical register pool, uops merely access the physical register file to get their destinations, and they write results back to the physical register file. Comparing physical register numbers enables the forwarding logic to forward results to other executing uops without having to understand logical processors.

After execution, the uops are placed in the re-order buffer. The re-order buffer decouples the execution stage from the retirement stage. The re-order buffer is partitioned such that each logical processor can use half the entries.

Retirement

Uop retirement logic commits the architecture state in program order. The retirement logic tracks when uops from the two logical processors are ready to be retired, then retires the uops in program order for each logical processor by alternating between the two logical processors. Retirement logic will retire uops for one logical processor, then the other, alternating back and forth. If one logical processor is not ready to retire any uops then all retirement bandwidth is dedicated to the other logical processor.

Once stores have retired, the store data needs to be written into the level-one data cache. Selection logic alternates between the two logical processors to commit store data to the cache.

MEMORY SUBSYSTEM

The memory subsystem includes the DTLB, the low-latency Level 1 (L1) data cache, the Level 2 (L2) unified cache, and the Level 3 unified cache (the Level 3 cache is only available on the Intel® Xeon™ processor MP). Access to the memory subsystem is also largely oblivious to logical processors. The schedulers send load or store uops without regard to logical processors and the memory subsystem handles them as they come.

DTLB

The DTLB translates addresses to physical addresses. It has 64 fully associative entries; each entry can map either a 4K or a 4MB page. Although the DTLB is a shared structure between the two logical processors, each entry includes a logical processor ID tag. Each logical processor also has a reservation register to ensure fairness and forward progress in processing DTLB misses.

L1 Data Cache, L2 Cache, L3 Cache

The L1 data cache is 4-way set associative with 64-byte lines. It is a write-through cache, meaning that writes are always copied to the L2 cache. The L1 data cache is virtually addressed and physically tagged.

The L2 and L3 caches are 8-way set associative with 128-byte lines. The L2 and L3 caches are physically addressed. Both logical processors, without regard to which logical processor's uops may have initially

[®]Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

[™]Xeon is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

brought the data into the cache, can share all entries in all three levels of cache.

Because logical processors can share data in the cache, there is the potential for cache conflicts, which can result in lower observed performance. However, there is also the possibility for sharing data in the cache. For example, one logical processor may prefetch instructions or data, needed by the other, into the cache; this is common in server application code. In a producer-consumer usage model, one logical processor may produce data that the other logical processor wants to use. In such cases, there is the potential for good performance benefits.

BUS

Logical processor memory requests not satisfied by the cache hierarchy are serviced by the bus logic. The bus logic includes the local APIC interrupt controller, as well as off-chip system memory and I/O space. Bus logic also deals with cacheable address coherency (snooping) of requests originated by other external bus agents, plus incoming interrupt request delivery via the local APICs.

From a service perspective, requests from the logical processors are treated on a first-come basis, with queue and buffering space appearing shared. Priority is not given to one logical processor above the other.

Distinctions between requests from the logical processors are reliably maintained in the bus queues nonetheless. Requests to the local APIC and interrupt delivery resources are unique and separate per logical processor. Bus logic also carries out portions of barrier fence and memory ordering operations, which are applied to the bus request queues on a per logical processor basis.

For debug purposes, and as an aid to forward progress mechanisms in clustered multiprocessor implementations, the logical processor ID is visibly sent onto the processor external bus in the request phase portion of a transaction. Other bus transactions, such as cache line eviction or prefetch transactions, inherit the logical processor ID of the request that generated the transaction.

SINGLE-TASK AND MULTI-TASK MODES

To optimize performance when there is one software thread to execute, there are two modes of operation referred to as single-task (ST) or multi-task (MT). In MT-mode, there are two active logical processors and some of the resources are partitioned as described

earlier. There are two flavors of ST-mode: single-task logical processor 0 (ST0) and single-task logical processor 1 (ST1). In ST0- or ST1-mode, only one logical processor is active, and resources that were partitioned in MT-mode are re-combined to give the single active logical processor use of all of the resources. The IA-32 Intel Architecture has an instruction called HALT that stops processor execution and normally allows the processor to go into a lower-power mode. HALT is a privileged instruction, meaning that only the operating system or other ring-0 processes may execute this instruction. User-level applications cannot execute HALT.

On a processor with Hyper-Threading Technology, executing HALT transitions the processor from MT-mode to ST0- or ST1-mode, depending on which logical processor executed the HALT. For example, if logical processor 0 executes HALT, only logical processor 1 would be active; the physical processor would be in ST1-mode and partitioned resources would be recombined giving logical processor 1 full use of all processor resources. If the remaining active logical processor also executes HALT, the physical processor would then be able to go to a lower-power mode.

In ST0- or ST1-modes, an interrupt sent to the HALTED processor would cause a transition to MT-mode. The operating system is responsible for managing MT-mode transitions (described in the next section).

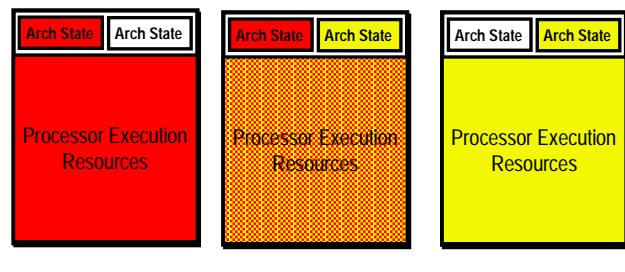


Figure 7: Resource allocation

Figure 7 summarizes this discussion. On a processor with Hyper-Threading Technology, resources are allocated to a single logical processor if the processor is in ST0- or ST1-mode. On the MT-mode, resources are shared between the two logical processors.

OPERATING SYSTEM AND APPLICATIONS

A system with processors that use Hyper-Threading Technology appears to the operating system and application software as having twice the number of processors than it physically has. Operating systems manage logical processors as they do physical

processors, scheduling runnable tasks or threads to logical processors. However, for best performance, the operating system should implement two optimizations.

The first is to use the HALT instruction if one logical processor is active and the other is not. HALT will allow the processor to transition to either the ST0- or ST1-mode. An operating system that does not use this optimization would execute on the idle logical processor a sequence of instructions that repeatedly checks for work to do. This so-called “idle loop” can consume significant execution resources that could otherwise be used to make faster progress on the other active logical processor.

The second optimization is in scheduling software threads to logical processors. In general, for best performance, the operating system should schedule threads to logical processors on different physical processors before scheduling multiple threads to the same physical processor. This optimization allows software threads to use different physical execution resources when possible.

PERFORMANCE

The Intel® Xeon™ processor family delivers the highest server system performance of any IA-32 Intel architecture processor introduced to date. Initial benchmark tests show up to a 65% performance increase on high-end server applications when compared to the previous-generation Pentium® III Xeon™ processor on 4-way server platforms. A significant portion of those gains can be attributed to Hyper-Threading Technology.

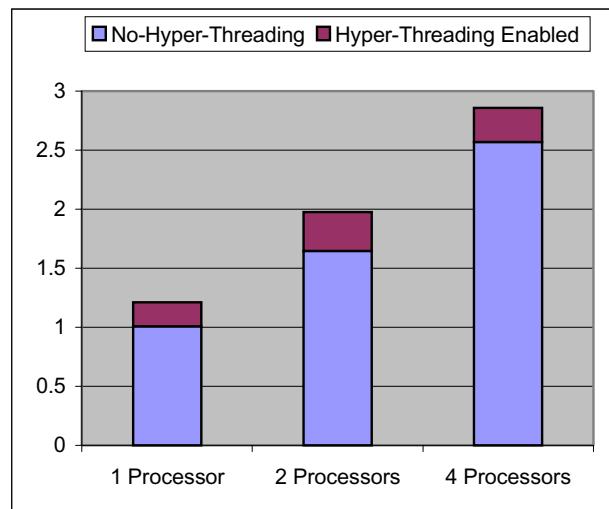


Figure 8: Performance increases from Hyper-Threading Technology on an OLTP workload

Figure 8 shows the online transaction processing performance, scaling from a single-processor configuration through to a 4-processor system with Hyper-Threading Technology enabled. This graph is normalized to the performance of the single-processor system. It can be seen that there is a significant overall performance gain attributable to Hyper-Threading Technology, 21% in the cases of the single and dual-processor systems.

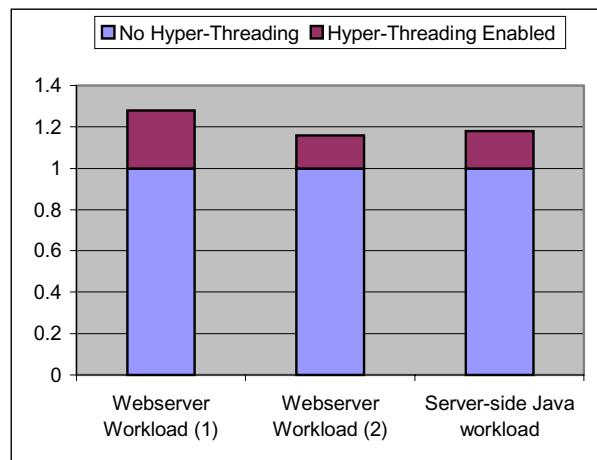


Figure 9: Web server benchmark performance

Figure 9 shows the benefit of Hyper-Threading Technology when executing other server-centric benchmarks. The workloads chosen were two different benchmarks that are designed to exercise data and Web server characteristics and a workload that focuses on exercising a server-side Java environment. In these cases the performance benefit ranged from 16 to 28%.

[®]Intel and Pentium are registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

[™]Xeon is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

All the performance results quoted above are normalized to ensure that readers focus on the relative performance and not the absolute performance.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, refer to www.intel.com/proc/perf/limits.htm or call (U.S.) 1-800-628-8686 or 1-916-356-3104

CONCLUSION

Intel's Hyper-Threading Technology brings the concept of simultaneous multi-threading to the Intel Architecture. This is a significant new technology direction for Intel's future processors. It will become increasingly important going forward as it adds a new technique for obtaining additional performance for lower transistor and power costs.

The first implementation of Hyper-Threading Technology was done on the Intel® Xeon™ processor MP. In this implementation there are two logical processors on each physical processor. The logical processors have their own independent architecture state, but they share nearly all the physical execution and hardware resources of the processor. The goal was to implement the technology at minimum cost while ensuring forward progress on logical processors, even if the other is stalled, and to deliver full performance even when there is only one active logical processor. These goals were achieved through efficient logical processor selection algorithms and the creative partitioning and recombining algorithms of many key resources.

Measured performance on the Intel Xeon processor MP with Hyper-Threading Technology shows performance gains of up to 30% on common server application benchmarks for this technology.

The potential for Hyper-Threading Technology is tremendous; our current implementation has only just

begun to tap into this potential. Hyper-Threading Technology is expected to be viable from mobile processors to servers; its introduction into market segments other than servers is only gated by the availability and prevalence of threaded applications and workloads in those markets.

ACKNOWLEDGMENTS

Making Hyper-Threading Technology a reality was the result of enormous dedication, planning, and sheer hard work from a large number of designers, validators, architects, and others. There was incredible teamwork from the operating system developers, BIOS writers, and software developers who helped with innovations and provided support for many decisions that were made during the definition process of Hyper-Threading Technology. Many dedicated engineers are continuing to work with our ISV partners to analyze application performance for this technology. Their contributions and hard work have already made and will continue to make a real difference to our customers.

REFERENCES

- A. Agarwal, B.H. Lim, D. Kranz and J. Kubiatowicz, "APRIL: A processor Architecture for Multiprocessing," in *Proceedings of the 17th Annual International Symposium on Computer Architectures*, pages 104-114, May 1990.
- R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porter, and B. Smith, "The TERA Computer System," in *International Conference on Supercomputing*, Pages 1 - 6, June 1990.
- L. A. Barroso et. al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Pages 282 - 293, June 2000.
- M. Fillo, S. Keckler, W. Dally, N. Carter, A. Chang, Y. Gurevich, and W. Lee, "The M-Machine Multicomputer," in *28th Annual International Symposium on Microarchitecture*, Nov. 1995.
- L. Hammond, B. Nayfeh, and K. Olukotun, "A Single-Chip Multiprocessor," *Computer*, 30(9), 79 - 85, September 1997.
- D. J. C. Johnson, "HP's Mako Processor," *Microprocessor Forum*, October 2001,
http://www.cpus.hp.com/technical_references/mpf_2001.pdf
- B.J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," in *SPIE Real Time Signal Processing IV*, Pages 2 241 - 248, 1981.
- J. M. Tendler, S. Dodson, and S. Fields, "POWER4 System Microarchitecture," *Technical White Paper. IBM Server Group*, October 2001.

® Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

™ Xeon is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

- D. Tullsen, S. Eggers, and H. Levy, "Simultaneous Multithreading: Maximizing On-chip Parallelism," in *22nd Annual International Symposium on Computer Architecture*, June 1995.
- D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *23rd Annual International Symposium on Computer Architecture*, May 1996.

Intel Corporation. "IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture," Order number 245472, 2001
<http://developer.intel.com/design/Pentium4/manuals>

Intel Corporation. "IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide," Order number 245472, 2001
<http://developer.intel.com/design/Pentium4/manuals>

AUTHORS' BIOGRAPHIES

Deborah T. Marr is the CPU architect responsible for Hyper-Threading Technology in the Desktop Products Group. Deborah has been at Intel for over ten years. She first joined Intel in 1988 and made significant contributions to the Intel 386SX processor, the P6 processor microarchitecture, and the Intel® Pentium® 4 Processor microarchitecture. Her interests are in high-performance microarchitecture and performance analysis. Deborah received her B.S. degree in EECS from the University of California at Berkeley in 1988, and her M.S. degree in ECE from Cornell University in 1992. Her e-mail address is debbie.marr@intel.com.

Frank Binns obtained a B.S. degree in electrical engineering from Salford University, England. He joined Intel in 1984 after holding research engineering positions with Marconi Research Laboratories and the Diamond Trading Company Research Laboratory, both of the U.K. Frank has spent the last 16 years with Intel, initially holding technical management positions in the Development Tool, Multibus Systems and PC Systems divisions. Frank's last eight years have been spent in the Desktop Processor Group in Technical Marketing and Processor Architecture roles. His e-mail is frank.binns@intel.com.

Dave L. Hill joined Intel in 1993 and was the quad pumped bus logic architect for the Pentium® 4 processor. Dave has 20 years industry experience primarily in high-performance memory system microarchitecture, logic design, and system debug. His e-mail address is david.l.hill@intel.com.

Glenn Hinton is an Intel Fellow, Desktop Platforms Group and Director of IA-32 Microarchitecture Development. He is responsible for the

microarchitecture development for the next-generation IA-32 design. He was appointed Intel Fellow in January 1999. He received bachelor's and master's degrees in Electrical Engineering from Brigham Young University in 1982 and 1983, respectively. His e-mail address is glenn.hinton@intel.com.

David A. Koufaty received B.S. and M.S. degrees from the Simon Bolivar University, Venezuela in 1988 and 1991, respectively. He then received a Ph.D. degree in Computer Science from the University of Illinois at Urbana-Champaign in 1997. For the last three years he has worked for the DPG CPU Architecture organization. His main interests are in multiprocessor architecture and software, performance, and compilation. His e-mail address is david.a.koufaty@intel.com.

John (Alan) Miller has worked at Intel for over five years. During that time, he worked on design and architecture for the Pentium® 4 processor and proliferation projects. Alan obtained his M.S. degree in Electrical and Computer Engineering from Carnegie-Mellon University. His e-mail is alan.miller@intel.com.

Michael Upton is a Principal Engineer/Architect in Intel's Desktop Platforms Group, and is one of the architects of the Intel Pentium® 4 processor. He completed B.S. and M.S. degrees in Electrical Engineering from the University of Washington in 1985 and 1990. After a number of years in IC design and CAD tool development, he entered the University of Michigan to study computer architecture. Upon completion of his Ph.D. degree in 1994, he joined Intel to work on the Pentium® Pro and Pentium 4 processors. His e-mail address is mike.upton@intel.com.

Copyright © Intel Corporation 2002.
Other names and brands may be claimed as the property of others.

This publication was downloaded from
<http://developer.intel.com/>

Legal notices at
<http://developer.intel.com/sites/corporate/tradmarx.htm>

NIAGARA: A 32-WAY MULTITHREADED SPARC PROCESSOR

THE NIAGARA PROCESSOR IMPLEMENTS A THREAD-RICH ARCHITECTURE DESIGNED TO PROVIDE A HIGH-PERFORMANCE SOLUTION FOR COMMERCIAL SERVER APPLICATIONS. THE HARDWARE SUPPORTS 32 THREADS WITH A MEMORY SUBSYSTEM CONSISTING OF AN ON-BOARD CROSSBAR, LEVEL-2 CACHE, AND MEMORY CONTROLLERS FOR A HIGHLY INTEGRATED DESIGN THAT EXPLOITS THE THREAD-LEVEL PARALLELISM INHERENT TO SERVER APPLICATIONS, WHILE TARGETING LOW LEVELS OF POWER CONSUMPTION.

Over the past two decades, microprocessor designers have focused on improving the performance of a single thread in a desktop processing environment by increasing frequencies and exploiting instruction level parallelism (ILP) using techniques such as multiple instruction issue, out-of-order issue, and aggressive branch prediction. The emphasis on single-thread performance has shown diminishing returns because of the limitations in terms of latency to main memory and the inherently low ILP of applications. This has led to an explosion in microprocessor design complexity and made power dissipation a major concern.

For these reasons, Sun Microsystems' Niagara processor takes a radically different approach to microprocessor design. Instead of focusing on the performance of single or dual threads, Sun optimized Niagara for multithreaded performance in a commercial server environment. This approach increases

application performance by improving throughput, the total amount of work done across multiple threads of execution. This is especially effective in commercial server applications such as databases¹ and Web services,² which tend to have workloads with large amounts of thread level parallelism (TLP).

In this article, we present the Niagara processor's architecture. This is an entirely new implementation of the Sparc V9 architectural specification, which exploits large amounts of on-chip parallelism to provide high throughput. Niagara supports 32 hardware threads by combining ideas from chip multiprocessors³ and fine-grained multithreading.⁴ Other studies⁵ have also indicated the significant performance gains possible using this approach on multithreaded workloads. The parallel execution of many threads effectively hides memory latency. However, having 32 threads places a heavy demand on the memory system to support high band-

**Poonacha Kongetira
Kathirgamar Aingaran
Kunle Olukotun
Sun Microsystems**

Table 1. Commercial server applications.

Benchmark	Application category	Instruction-level parallelism	Thread-level parallelism	Working set	Data sharing
Web99	Web server	Low	High	Large	Low
JBB	Java application server	Low	High	Large	Medium
TPC-C	Transaction processing	Low	High	Large	High
SAP-2T	Enterprise resource planning	Medium	High	Medium	Medium
SAP-3T	Enterprise resource planning	Low	High	Large	High
TPC-H	Decision support system	High	High	Large	Medium

width. To provide this bandwidth, a crossbar interconnects scheme routes memory references to a banked on-chip level-2 cache that all threads share. Four independent on-chip memory controllers provide in excess of 20 Gbytes/s of bandwidth to memory.

Exploiting TLP also lets us improve performance significantly without pushing the envelope on CPU clock frequency. This and the sharing of CPU pipelines among multiple threads enable an area- and power-efficient design. Designers expect Niagara to dissipate about 60 W of power, making it very attractive for high compute density environments. In data centers, for example, power supply and air conditioning costs have become very significant. Data center racks often cannot hold a complete complement of servers because this would exceed the rack's power supply envelope.

We designed Niagara to run the Solaris operating system, and existing Solaris applications will run on Niagara systems without modification. To application software, a Niagara processor will appear as 32 discrete processors with the OS layer abstracting away the hardware sharing. Many multithreaded applications currently running on symmetric multiprocessor (SMP) systems should realize performance improvements. This is consistent with observations from previous multithreaded-processor development at Sun^{6,7} and from Niagara chips and systems, which are undergoing bring-up testing in the laboratory.

Recently, the movement of many retail and business processes to the Web has triggered the increasing use of commercial server applications (Table 1). These server applications exhibit large degrees of client request-level parallelism, which servers using multiple threads can exploit. The key performance metric for

a server running these applications is the sustained throughput of client requests.

Furthermore, the deployment of servers commonly takes place in high compute density installations such as data centers, where supplying power and dissipating server-generated heat are very significant factors in the center's cost of operating. Experience at Google shows a representative power density requirement of 400 to 700 W/sq. foot for racked server clusters.² This far exceeds the typical power densities of 70 to 150 W/foot² supported by commercial data centers. It is possible to reduce power consumption by simply running the ILP processors in server clusters at lower clock frequencies, but the proportional loss in performance makes this less desirable. This situation motivates the requirement for commercial servers to improve performance per watt. These requirements have not been efficiently met using machines optimized for single-thread performance.

Commercial server applications tend to have low ILP because they have large working sets and poor locality of reference on memory access; both contribute to high cache-miss rates. In addition, data-dependent branches are difficult to predict, so the processor must discard work done on the wrong path. Load-load dependencies are also present, and are not detectable in hardware at issue time, resulting in discarded work. The combination of low available ILP and high cache-miss rates causes memory access time to limit performance. Therefore, the performance advantage of using a complex ILP processor over a single-issue processor is not significant, while the ILP processor incurs the costs of high power and complexity, as Figure 1 shows.

However, server applications tend to have large amounts of TLP. Therefore, shared-

memory machines with discrete single-threaded processors and coherent interconnect have tended to perform well because they exploit TLP. However, the use of an SMP composed of multiple processors designed to exploit ILP is neither power efficient nor cost-efficient. A more efficient approach is to build a machine using simple cores aggregated on a single die, with a shared on-chip cache and high bandwidth to large off-chip memory, thereby aggregating an SMP server on a chip. This has the added benefit of low-latency communication between the cores for efficient data sharing in commercial server applications.

Niagara overview

The Niagara approach to increasing throughput on commercial server applications involves a dramatic increase in the number of threads supported on the processor and a memory subsystem scaled for higher bandwidths. Niagara supports 32 threads of execution in hardware. The architecture organizes four threads into a thread group; the group shares a processing pipeline, referred to as the *Sparc pipe*. Niagara uses eight such thread groups, resulting in 32 threads on the CPU. Each SPARC pipe contains level-1 caches for instructions and data. The hardware hides memory and pipeline stalls on a given thread by scheduling the other threads in the group onto the SPARC pipe with a zero cycle switch penalty. Figure 1 schematically shows how reusing the shared processing pipeline results in higher throughput.

The 32 threads share a 3-Mbyte level-2 cache. This cache is 4-way banked and pipelined for bandwidth; it is 12-way set-associative to minimize conflict misses from the many threads. Commercial server code has data sharing, which can lead to high coherence miss rates. In conventional SMP systems using discrete processors with coherent system interconnects, coherence misses go out over low-frequency off-chip buses or links, and can have high latencies. The Niagara design with its shared on-chip cache eliminates these misses and replaces them with low-latency shared-cache communication.

The crossbar interconnect provides the communication link between Sparc pipes, L2 cache banks, and other shared resources on the CPU; it provides more than 200 Gbytes/s

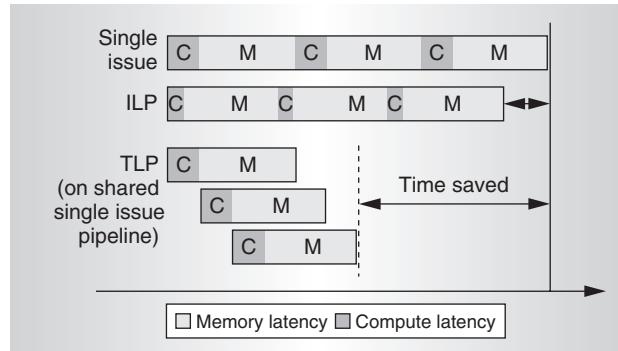


Figure 1. Behavior of processors optimized for TLP and ILP on commercial server workloads. In comparison to the single-issue machine, the ILP processor mainly reduces compute time, so memory access time dominates application performance. In the TLP case, multiple threads share a single-issue pipeline, and overlapped execution of these threads results in higher performance for a multithreaded application.

of bandwidth. A two-entry queue is available for each source-destination pair, and it can queue up to 96 transactions each way in the crossbar. The crossbar also provides a port for communication with the I/O subsystem. Arbitration for destination ports uses a simple age-based priority scheme that ensures fair scheduling across all requestors. The crossbar is also the point of memory ordering for the machine.

The memory interface is four channels of dual-data rate 2 (DDR2) DRAM, supporting a maximum bandwidth in excess of 20 Gbytes/s, and a capacity of up to 128 Gbytes. Figure 2 shows a block diagram of the Niagara processor.

Sparc pipeline

Here we describe the Sparc pipe implementation, which supports four threads. Each thread has a unique set of registers and instruction and store buffers. The thread group shares the L1 caches, translation look-aside buffers (TLBs), execution units, and most pipeline registers. We implemented a single-issue pipeline with six stages (fetch, thread select, decode, execute, memory, and write back).

In the fetch stage, the instruction cache and instruction TLB (ITLB) are accessed. The following stage completes the cache access by selecting the way. The critical path is set by the 64-entry, fully associative ITLB access. A thread-select multiplexer determines which of

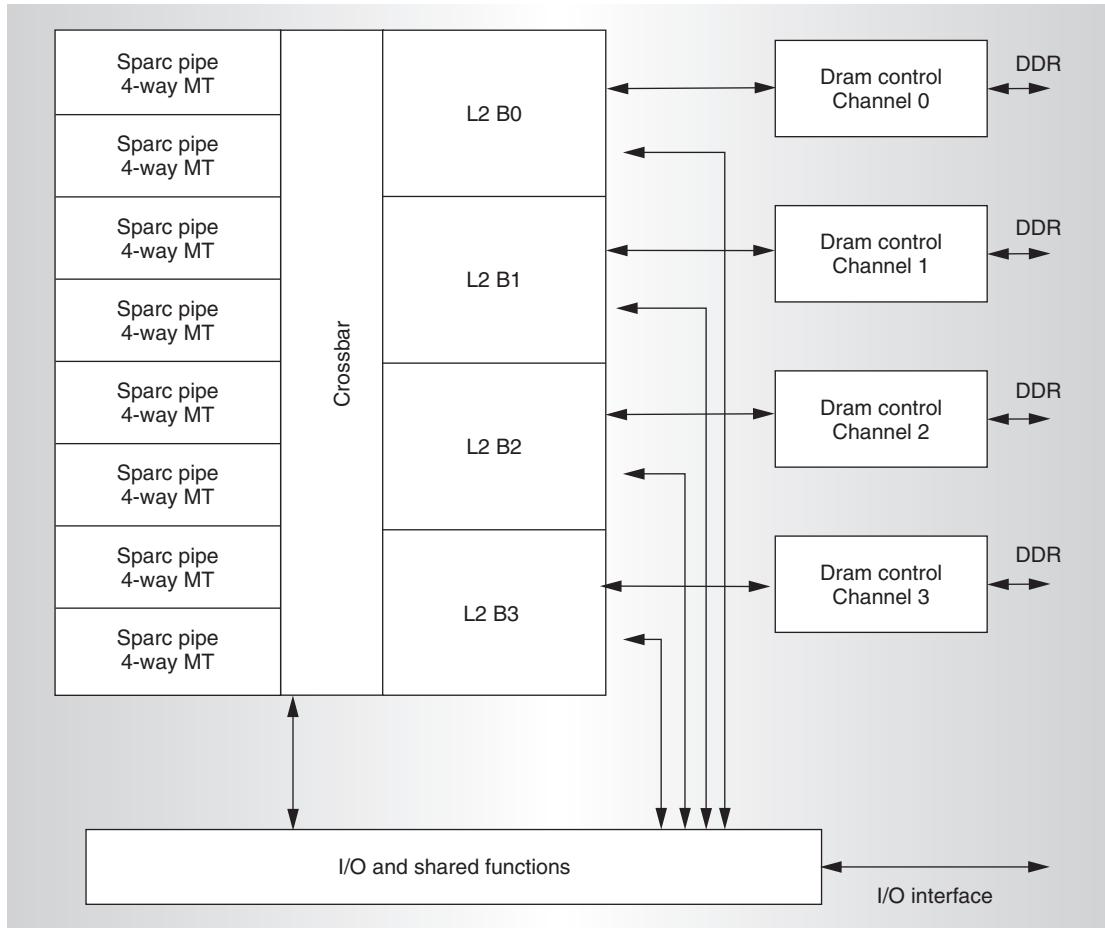


Figure 2. Niagara block diagram.

the four thread program counters (PC) should perform the access. The pipeline fetches two instructions each cycle. A predecode bit in the cache indicates long-latency instructions.

In the thread-select stage, the thread-select multiplexer chooses a thread from the available pool to issue an instruction into the downstream stages. This stage also maintains the instruction buffers. Instructions fetched from the cache in the fetch stage can be inserted into the instruction buffer for that thread if the downstream stages are not available. Pipeline registers for the first two stages are replicated for each thread.

Instructions from the selected thread go into the decode stage, which performs instruction decode and register file access. The supported execution units include an arithmetic logic unit (ALU), shifter, multiplier, and a divider. A bypass unit handles instruction results that must be passed to dependent

instructions before the register file is updated. ALU and shift instructions have single-cycle latency and generate results in the execute stage. Multiply and divide operations are long latency and cause a thread switch.

The load store unit contains the data TLB (DTLB), data cache, and store buffers. The DTLB and data cache access take place in the memory stage. Like the fetch stage, the critical path is set by the 64-entry, fully associative DTLB access. The load-store unit contains four 8-entry store buffers, one per thread. Checking the physical tags in the store buffer can indicate read after write (RAW) hazards between loads and stores. The store buffer supports the bypassing of data to a load to resolve RAW hazards. The store buffer tag check happens after the TLB access in the early part of write back stage. Load data is available for bypass to dependent instructions late in the write back stage. Single-cycle

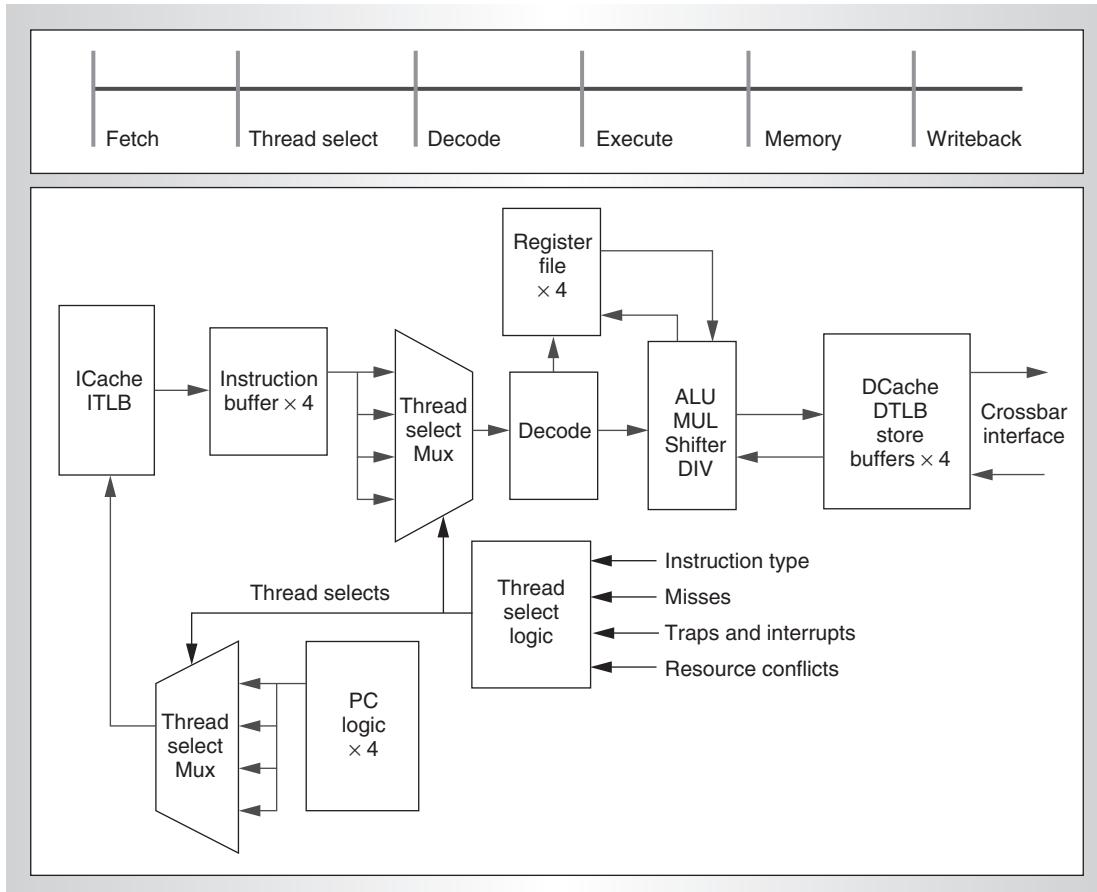


Figure 3. Sparc pipeline block diagram. Four threads share a six-stage single-issue pipeline with local instruction and data caches. Communication with the rest of the machine occurs through the crossbar interface.

instructions such as ADD will update the register file in the write back stage.

The thread-select logic decides which thread is active in a given cycle in the fetch and thread-select stages. As Figure 3 shows, the thread-select signals are common to fetch and thread-select stages. Therefore, if the thread-select stage chooses a thread to issue an instruction to the decode stage, the F stage also selects the same instruction to access the cache. The thread-select logic uses information from various pipeline stages to decide when to select or deselect a thread. For example, the thread-select stage can determine instruction type using a predecode bit in the instruction cache, while some traps are only detectable in later pipeline stages. Therefore, instruction type can cause deselection of a thread in the thread-select stage, while a late trap detected in the memory stage needs to flush all younger instructions from the thread and deselect itself during trap processing.

Pipeline interlocks and scheduling

For single-cycle instructions such as ADD, Niagara implements full bypassing to younger instructions from the same thread to resolve RAW dependencies. As mentioned before, load instructions have a three-cycle latency before the results of the load are visible to the next instruction. Such long-latency instructions can cause pipeline hazards; resolving them requires stalling the corresponding thread until the hazard clears. So, in the case of a load, the next instruction from the same thread waits for two cycles for the hazards to clear.

In a multithreaded pipeline, threads competing for shared resources also encounter structural hazards. Resources such as the ALU that have a one-instruction-per-cycle throughput present no hazards, but the divider, which has a throughput of less than one instruction per cycle, presents a scheduling problem. In this case, any thread that must execute a DIV

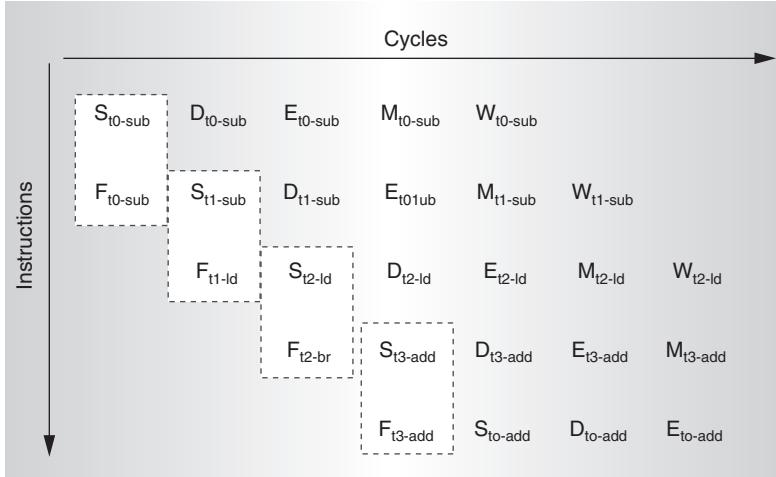
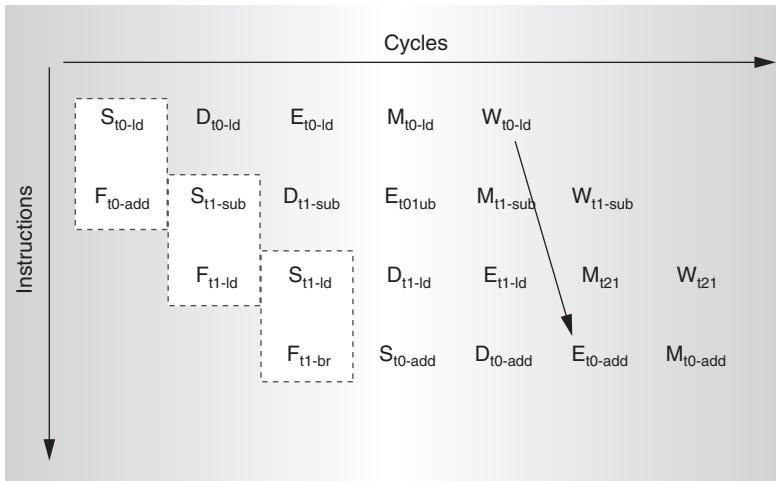


Figure 4. Thread selection: all threads available.

Figure 5. Thread selection: only two threads available. The ADD instruction from *thread0* is speculatively switched into the pipeline before the hit/miss for the load instruction has been determined.

instruction has to wait until the divider is free. The thread scheduler guarantees fairness in accessing the divider by giving priority to the least recently executed thread. Although the divider is in use, other threads can use free resources such as the ALU, load-store unit, and so on.

Thread selection policy

The thread selection policy is to switch between available threads every cycle, giving priority to the least recently used thread. Threads can become unavailable because of long-latency instructions such as loads, branches, and multiply and divide. They also

become unavailable because of pipeline stalls such as cache misses, traps, and resource conflicts. The thread scheduler assumes that loads are cache hits, and can therefore issue a dependent instruction from the same thread speculatively. However, such a speculative thread is assigned a lower priority for instruction issue as compared to a thread that can issue a non-speculative instruction.

Figure 4 indicates the operation when all threads are available. In the figure, reading from left to right indicates the progress of an instruction through the pipeline. Reading from top to bottom indicates new instructions fetched into the pipe from the instruction cache. The notation $S_{t0\text{-sub}}$ refers to a Subtract instruction from thread 0 in the S stage of the pipe. In the example, the $t0\text{-sub}$ is issued down the pipe. As the other three threads become available, the thread state machine selects *thread1* and deselects *thread0*. In the second cycle, similarly, the pipeline executes the $t1\text{-sub}$ and selects $t2\text{-ld}$ (load instruction from thread 2) for issue in the following cycle. When $t3\text{-add}$ is in the S stage, all threads have been executed, and for the next cycle the pipeline selects the least recently used thread, *thread0*. When the thread-select stage chooses a thread for execution, the fetch stage chooses the same thread for instruction cache access.

Figure 5 indicates the operation when only two threads are available. Here *thread0* and *thread1* are available, while *thread2* and *thread3* are not. The $t0\text{-ld}$ in the thread-select stage in the example is a long-latency operation. Therefore it causes the deselection of *thread0*. The $t0\text{-ld}$ itself, however, issues down the pipe. In the second cycle, since *thread1* is available, the thread scheduler switches it in. At this time, there are no other threads available and the $t1\text{-sub}$ is a single-cycle operation, so *thread1* continues to be selected for the next cycle. The subsequent instruction is a $t1\text{-ld}$ and causes the deselection of *thread1* for the fourth cycle. At this time only *thread0* is speculatively available and therefore can be selected. If the first $t0\text{-ld}$ was a hit, data can bypass to the dependent $t0\text{-add}$ in the execute stage. If the load missed, the pipeline flushes the subsequent $t0\text{-add}$ to the thread-select stage instruction buffer, and the instruction reissues when the load returns from the L2 cache.

Integer register file

The integer register file has three read and two write ports. The read ports correspond to those of a single-issue machine; the third read port handles stores and a few other three source instructions. The two write ports handle single-cycle and long-latency operations. Long-latency operations from various threads within the group (load, multiply, and divide) can generate results simultaneously. These instructions will arbitrate for the long-latency port of the register file for write backs. Sparc V9 architecture specifies the register window implementation shown in Figure 6. A single window consists of eight *in*, *local*, and *out* registers; they are all visible to a thread at a given time. The *out* registers of a window are addressed as the *in* registers of the next sequential window, but are the same physical registers. Each thread has eight register windows. Four such register sets support each of the four threads, which do not share register space among themselves. The register file consists of a total of 640 64-bit registers and is a 5.7 Kbyte structure. Supporting the many multiported registers can be a major challenge from both area and access time considerations. We have chosen an innovative implementation to maintain a compact footprint and a single-cycle access time.

Procedure calls can request a new window, in which case the visible window slides up, with the old outputs becoming the new inputs. Return from a call causes the window to slide down. We take advantage of this characteristic to implement a compact register file. The set of registers visible to a thread is the *working set*, and we implement it using fast register file cells. The complete set of registers is the *architectural set*; we implement it using compact six-transistor SRAM cells. A transfer port links the two register sets. A window changing event triggers deselection of the thread and the transfer of data between the old and new windows. Depending on the event type, the data transfer takes one or two cycles. When the transfer is complete, the thread can be selected again. This data transfer is independent of operations to registers from other threads, therefore operations on the register file from other threads can continue. In addition, the registers of all threads share the read circuitry because only one

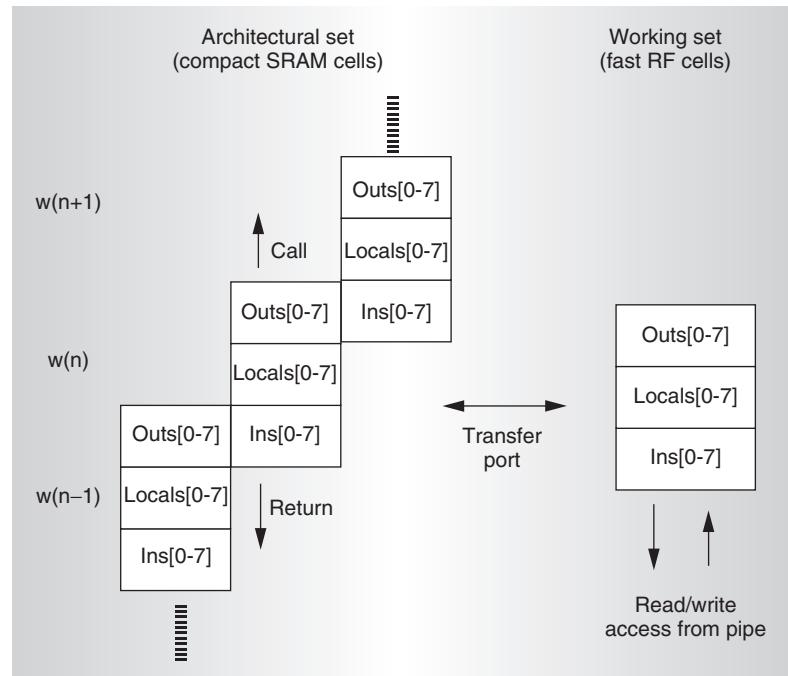


Figure 6. Windowed register file. $w(n - 1)$, $w(n)$, and $w(n + 1)$ are neighboring register windows. Pipeline accesses occur in the working set while window changing events cause data transfer between the working set and old or new architectural windows. This structure is replicated for each of four threads,

thread can read the register file in a given cycle.

Memory subsystem

The L1 instruction cache is 16 Kbyte, 4-way set-associative with a block size of 32 bytes. We implement a random replacement scheme for area savings, without incurring significant performance cost. The instruction cache fetches two instructions each cycle. If the second instruction is useful, the instruction cache has a free slot, which the pipeline can use to handle a line fill without stalling. The L1 data cache is 8 Kbytes, 4-way set-associative with a line size of 16 bytes, and implements a write-through policy. Even though the L1 caches are small, they significantly reduce the average memory access time per thread with miss rates in the range of 10 percent. Because commercial server applications tend to have large working sets, the L1 caches must be much larger to achieve significantly lower miss rates, so this sort of trade-off is not favorable for area. However, the four threads in a thread group effectively hide the latencies from L1 and L2 misses. Therefore,

Hardware multithreading primer

A multithreaded processor is one that allows more than one thread of execution to exist on the CPU at the same time. To software, a dual-threaded processor looks like two distinct CPUs, and the operating system takes advantage of this by scheduling two threads of execution on it. In most cases, though, the threads share CPU pipeline resources, and hardware uses various means to manage the allocation of resources to these threads.

The industry uses several terms to describe the variants of multithreading implemented in hardware; we show some in Figure A.

In a *single-issue, single-thread machine* (included here for reference), hardware does not control thread scheduling on the pipeline. Single-thread machines support a single context in hardware; therefore a thread switch by the operating system incurs the overhead of saving and retrieving thread states from memory.

Coarse-grained multithreading, or *switch-on-event multithreading*, is when a thread has full use of the CPU resource until a long-latency event such as miss to DRAM occurs, in which case the CPU switches to another thread. Typically, this implementation has a context switch cost associated with it, and thread switches occur when event latency exceeds a specified threshold.

Fine grained multithreading is sometimes called *interleaved multithreading*; in it, thread selection typically happens at a cycle boundary. The selection policy can be simply to allocate resources to a thread on a round-robin basis with threads becoming unavailable for longer periods of time on events like memory misses.

The preceding types time slice the processor resources, so these implementations are also called time-sliced or vertical multithreaded. An approach that schedules instructions from different threads on different functional units at the same time is called *simultaneous multithreading* (SMT) or, alternately, *horizontal multithreading*. SMT typically works on superscalar processors, which have hardware for scheduling across multiple pipes.

Another type of implementation is *chip multiprocessing*, which simply calls for instantiating single-threaded processor cores on a die, perhaps

sharing a next-level cache and system interfaces. Here, each processor core executes instructions from a thread independently of the other thread, and they interact through shared memory. This has been an attractive option for chip designers because of the availability of cores from earlier processor generations, which, when shrunk down to present-day process technologies, are small enough for aggregation onto a single die.

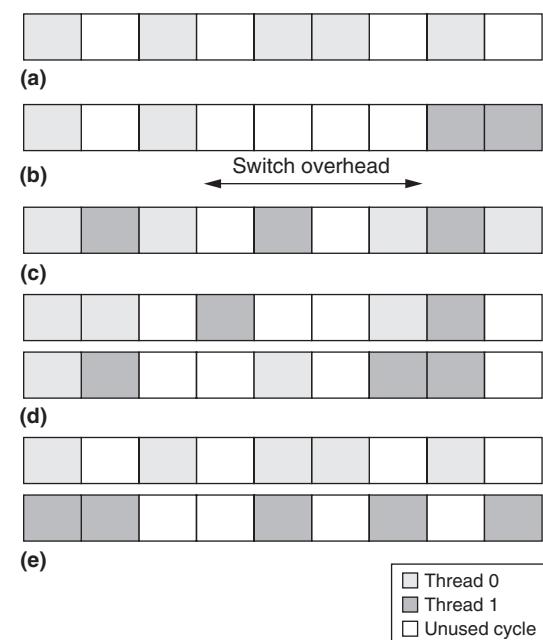


Figure A. Variants of multithreading implementation hardware: single-issue, single-thread pipeline (a); single-issue, 2 threads, coarse-grain threading (b); single-issue, 2 threads, fine-grain threading (c); dual-issue, 2 threads, simultaneous threading (d); and single-issue, 2 threads, chip multiprocessing (e).

the cache sizes are a good trade-off between miss rates, area, and the ability of other threads in the group to hide latency.

Niagara uses a simple cache coherence protocol. The L1 caches are write through, with allocate on load and no-allocate on stores. L1 lines are either in valid or invalid states. The L2 cache maintains a directory that shadows the L1 tags. The L2 cache also interleaves data across banks at a 64-byte granularity. A load that missed in an L1 cache (load miss) is delivered to the source bank of the L2 cache along with its replacement way from the L1 cache. There, the load miss address is entered in the corresponding L1 tag location of the directory, the L2 cache is accessed to get the missing

line and data is then returned to the L1 cache. The directory thus maintains a sharers list at L1-line granularity. A subsequent store from a different or same L1 cache will look up the directory and queue up invalidates to the L1 caches that have the line. Stores do not update the local caches until they have updated the L2 cache. During this time, the store can pass data to the same thread but not to other threads; therefore, a store attains global visibility in the L2 cache. The crossbar establishes memory order between transactions from the same and different L2 banks, and guarantees delivery of transactions to L1 caches in the same order. The L2 cache follows a copy-back policy, writing back dirty evicts and dropping

clean evicts. Direct memory access from I/O devices are ordered through the L2 cache. Four channels of DDR2 DRAM provide in excess of 20 Gbytes/s of memory bandwidth.

Niagara systems are presently undergoing testing and bring up. We have run existing multithreaded application software written for Solaris without modification on laboratory systems. The simple pipeline requires no special compiler optimizations for instruction scheduling. On real commercial applications, we have observed the performance in lab systems to scale almost linearly with the number of threads, demonstrating that there are no bandwidth bottlenecks in the design. The Niagara processor represents the first in a line of microprocessors that Sun designed with many hardware threads to provide high throughput and high performance per watt on commercial server applications. The availability of a thread-rich architecture opens up new avenues for developers to efficiently increase application performance. This architecture is a paradigm shift in the way that microprocessors have been designed thus far.

MICRO

Acknowledgments

This article represents the work of the very talented and dedicated Niagara development team, and we are privileged to present their work.

References

1. S.R. Kunkel et al., "A Performance Methodology for Commercial Servers," *IBM J. Research and Development*, vol. 44, no. 6, 2000, pp. 851-872.
2. L. Barroso, J. Dean, and U. Hoezle, "Web Search for a Planet: The Architecture of the Google Cluster," *IEEE Micro*, vol 23, no. 2, Mar.-Apr. 2003, pp. 22-28.
3. K. Olukotun et al., "The Case for a Single Chip Multiprocessor," *Proc. 7th Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, 1996, pp. 2-11.
4. J. Laudon, A. Gupta, and M. Horowitz, "Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations," *Proc. 6th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, ACM Press, 1994, pp. 308-316.
5. L. Barroso et al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," *Proc. 27th Ann. Int'l Symp. Computer Architecture (ISCA 00)*, IEEE CS Press, 2000, pp. 282-293.
6. S. Kapil, H. McGhan, and J. Lawrendra, "A Chip Multithreaded Processor for Network-Facing Workloads," *IEEE Micro*, vol. 24, no. 2, Mar.-Apr. 2004, pp. 20-30.
7. J. Hart et al., "Implementation of a 4th-Generation 1.8 GHz Dual Core Sparc V9 Microprocessor," *Proc. Int'l Solid-State Circuits Conf. (ISSCC 05)*, IEEE Press, 2005, <http://www.isscc.org/isscc/2005/ap/ISSCC2005AdvanceProgram.pdf>.

Poonacha Kongetira is a director of engineering at Sun Microsystems and was part of the development team for the Niagara processor. His research interests include computer architecture and methodologies for SoC design. Kongetira has a MS from Purdue University and BS from Birla Institute of Technology and Science, both in electrical engineering. He is a member of the IEEE.

Kathirgamar Aingaran is a senior staff engineer at Sun Microsystems and was part of the development team for Niagara. His research interests include architectures for low power and low design complexity. Aingaran has an MS from Stanford University and a BS from the University of Southern California, both in electrical engineering. He is a member of the IEEE.

Kunle Olukotun is an associate professor of electrical engineering and computer science at Stanford University. His research interests include computer architecture, parallel programming environments, and formal hardware verification. Olukotun has a PhD in computer science and engineering from the University of Michigan. He is a member of the IEEE and the ACM.

Direct questions and comments about this article to Poonacha Kongetira, MS: USUN05-215, 910 Hermosa Court, Sunnyvale, CA 94086; poonacha.kongetira@sun.com.

For further information on this or any other computing topic, visit our Digital Library at <http://www.computer.org/publications/dlib>.

NVIDIA TESLA: A UNIFIED GRAPHICS AND COMPUTING ARCHITECTURE

TO ENABLE FLEXIBLE, PROGRAMMABLE GRAPHICS AND HIGH-PERFORMANCE COMPUTING, NVIDIA HAS DEVELOPED THE TESLA SCALABLE UNIFIED GRAPHICS AND PARALLEL COMPUTING ARCHITECTURE. ITS SCALABLE PARALLEL ARRAY OF PROCESSORS IS MASSIVELY MULTITHREADED AND PROGRAMMABLE IN C OR VIA GRAPHICS APIs.

..... The modern 3D graphics processing unit (GPU) has evolved from a fixed-function graphics pipeline to a programmable parallel processor with computing power exceeding that of multicore CPUs. Traditional graphics pipelines consist of separate programmable stages of vertex processors executing vertex shader programs and pixel fragment processors executing pixel shader programs. (Montrym and Moreton provide additional background on the traditional graphics processor architecture.¹)

NVIDIA's Tesla architecture, introduced in November 2006 in the GeForce 8800 GPU, unifies the vertex and pixel processors and extends them, enabling high-performance parallel computing applications written in the C language using the Compute Unified Device Architecture (CUDA²⁻⁴) parallel programming model and development tools. The Tesla unified graphics and computing architecture is available in a scalable family of GeForce 8-series GPUs and Quadro GPUs for laptops, desktops, workstations, and servers. It also provides the processing architecture for the Tesla GPU computing platforms introduced in 2007 for high-performance computing.

In this article, we discuss the requirements that drove the unified graphics and parallel computing processor architecture, describe the Tesla architecture, and how it is enabling widespread deployment of parallel computing and graphics applications.

The road to unification

The first GPU was the GeForce 256, introduced in 1999. It contained a fixed-function 32-bit floating-point vertex transform and lighting processor and a fixed-function integer pixel-fragment pipeline, which were programmed with OpenGL and the Microsoft DX7 API.⁵ In 2001, the GeForce 3 introduced the first programmable vertex processor executing vertex shaders, along with a configurable 32-bit floating-point fragment pipeline, programmed with DX8⁵ and OpenGL.⁶ The Radeon 9700, introduced in 2002, featured a programmable 24-bit floating-point pixel-fragment processor programmed with DX9 and OpenGL.^{7,8} The GeForce FX added 32-bit floating-point pixel-fragment processors. The XBox 360 introduced an early unified GPU in 2005, allowing vertices and pixels to execute on the same processor.⁹

Vertex processors operate on the vertices of primitives such as points, lines, and triangles. Typical operations include transforming coordinates into screen space, which are then fed to the setup unit and the rasterizer, and setting up lighting and texture parameters to be used by the pixel-fragment processors. Pixel-fragment processors operate on rasterizer output, which fills the interior of primitives, along with the interpolated parameters.

Vertex and pixel-fragment processors have evolved at different rates: Vertex processors were designed for low-latency, high-precision math operations, whereas pixel-fragment processors were optimized for high-latency, lower-precision texture filtering. Vertex processors have traditionally supported more-complex processing, so they became programmable first. For the last six years, the two processor types have been functionally converging as the result of a need for greater programming generality. However, the increased generality also increased the design complexity, area, and cost of developing two separate processors.

Because GPUs typically must process more pixels than vertices, pixel-fragment processors traditionally outnumber vertex processors by about three to one. However, typical workloads are not well balanced, leading to inefficiency. For example, with large triangles, the vertex processors are mostly idle, while the pixel processors are fully busy. With small triangles, the opposite is true. The addition of more-complex primitive processing in DX10 makes it much harder to select a fixed processor ratio.¹⁰ All these factors influenced the decision to design a unified architecture.

A primary design objective for Tesla was to execute vertex and pixel-fragment shader programs on the same unified processor architecture. Unification would enable dynamic load balancing of varying vertex- and pixel-processing workloads and permit the introduction of new graphics shader stages, such as geometry shaders in DX10. It also let a single team focus on designing a fast and efficient processor and allowed the sharing of expensive hardware such as the

texture units. The generality required of a unified processor opened the door to a completely new GPU parallel-computing capability. The downside of this generality was the difficulty of efficient load balancing between different shader types.

Other critical hardware design requirements were architectural scalability, performance, power, and area efficiency.

The Tesla architects developed the graphics feature set in coordination with the development of the Microsoft Direct3D DirectX 10 graphics API.¹⁰ They developed the GPU's computing feature set in coordination with the development of the CUDA C parallel programming language, compiler, and development tools.

Tesla architecture

The Tesla architecture is based on a scalable processor array. Figure 1 shows a block diagram of a GeForce 8800 GPU with 128 streaming-processor (SP) cores organized as 16 streaming multiprocessors (SMs) in eight independent processing units called texture/processor clusters (TPCs). Work flows from top to bottom, starting at the host interface with the system PCI-Express bus. Because of its unified-processor design, the physical Tesla architecture doesn't resemble the logical order of graphics pipeline stages. However, we will use the logical graphics pipeline flow to explain the architecture.

At the highest level, the GPU's scalable streaming processor array (SPA) performs all the GPU's programmable calculations. The scalable memory system consists of external DRAM control and fixed-function raster operation processors (ROPs) that perform color and depth frame buffer operations directly on memory. An interconnection network carries computed pixel-fragment colors and depth values from the SPA to the ROPs. The network also routes texture memory read requests from the SPA to DRAM and read data from DRAM through a level-2 cache back to the SPA.

The remaining blocks in Figure 1 deliver input work to the SPA. The input assembler collects vertex work as directed by the input command stream. The vertex work distri-

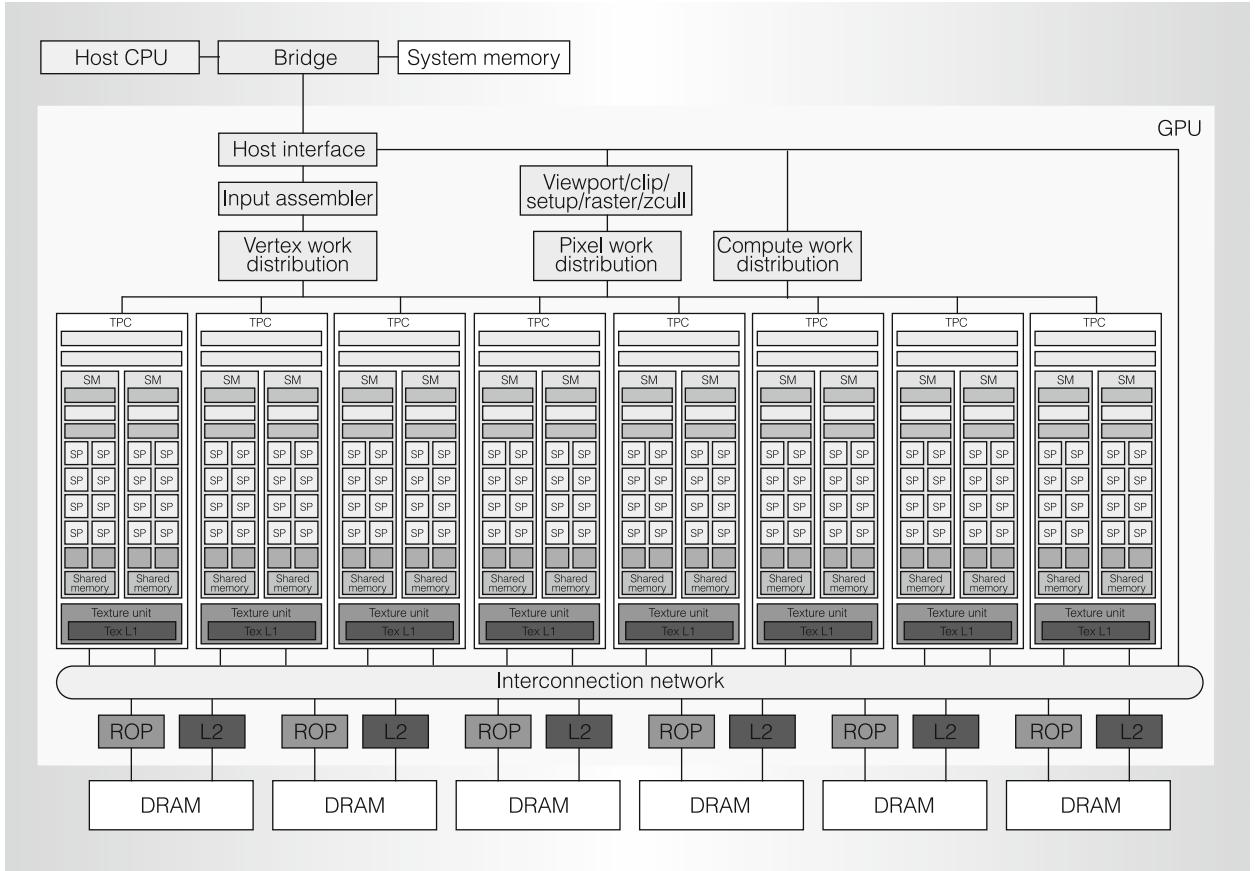


Figure 1. Tesla unified graphics and computing GPU architecture. TPC: texture/processor cluster; SM: streaming multiprocessor; SP: streaming processor; Tex: texture, ROP: raster operation processor.

bution block distributes vertex work packets to the various TPCs in the SPA. The TPCs execute vertex shader programs, and (if enabled) geometry shader programs. The resulting output data is written to on-chip buffers. These buffers then pass their results to the viewport/clip/setup/raster/zcull block to be rasterized into pixel fragments. The pixel work distribution unit distributes pixel fragments to the appropriate TPCs for pixel-fragment processing. Shaded pixel-fragments are sent across the interconnection network for processing by depth and color ROP units. The compute work distribution block dispatches compute thread arrays to the TPCs. The SPA accepts and processes work for multiple logical streams simultaneously. Multiple clock domains for GPU units, processors, DRAM, and other units allow independent power and performance optimizations.

Command processing

The GPU host interface unit communicates with the host CPU, responds to commands from the CPU, fetches data from system memory, checks command consistency, and performs context switching.

The input assembler collects geometric primitives (points, lines, triangles, line strips, and triangle strips) and fetches associated vertex input attribute data. It has peak rates of one primitive per clock and eight scalar attributes per clock at the GPU core clock, which is typically 600 MHz.

The work distribution units forward the input assembler's output stream to the array of processors, which execute vertex, geometry, and pixel shader programs, as well as computing programs. The vertex and compute work distribution units deliver work to processors in a round-robin scheme. Pixel

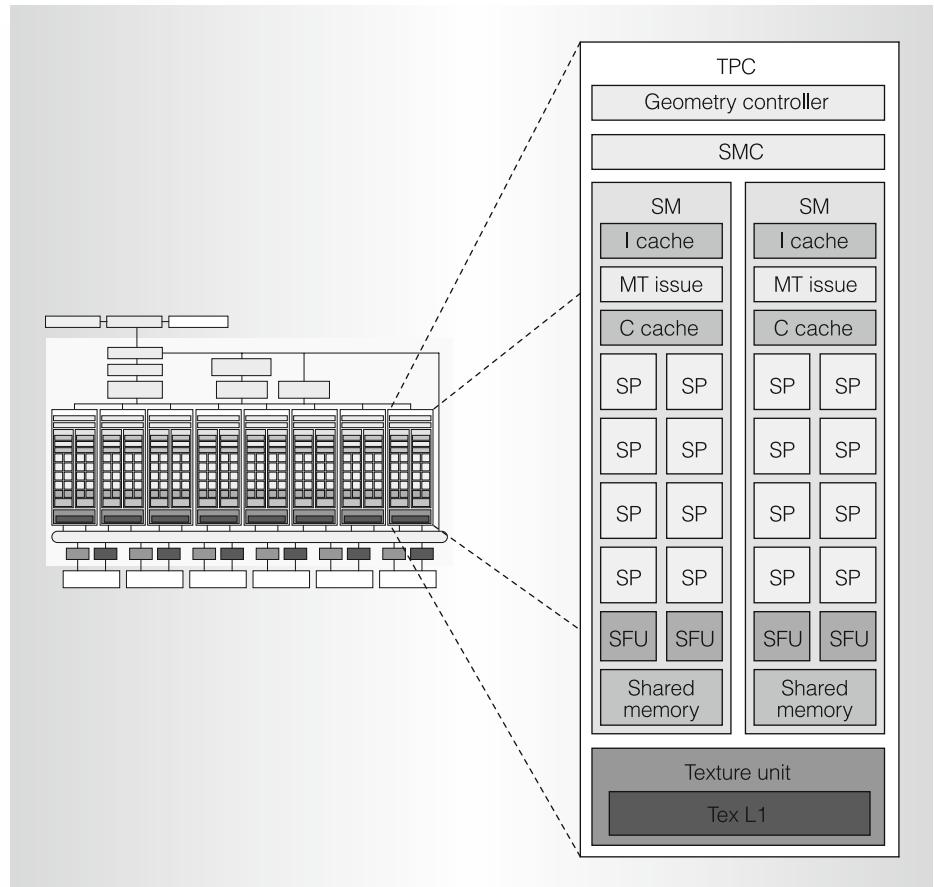


Figure 2. Texture/processor cluster (TPC).

work distribution is based on the pixel location.

Streaming processor array

The SPA executes graphics shader thread programs and GPU computing programs and provides thread control and management. Each TPC in the SPA roughly corresponds to a quad-pixel unit in previous architectures.¹ The number of TPCs determines a GPU's programmable processing performance and scales from one TPC in a small GPU to eight or more TPCs in high-performance GPUs.

Texture/processor cluster

As Figure 2 shows, each TPC contains a geometry controller, an SM controller (SMC), two streaming multiprocessors (SMs), and a texture unit. Figure 3 expands each SM to show its eight SP cores. To balance the expected ratio of math opera-

tions to texture operations, one texture unit serves two SMs. This architectural ratio can vary as needed.

Geometry controller

The geometry controller maps the logical graphics vertex pipeline into recirculation on the physical SMs by directing all primitive and vertex attribute and topology flow in the TPC. It manages dedicated on-chip input and output vertex attribute storage and forwards contents as required.

DX10 has two stages dealing with vertex and primitive processing: the vertex shader and the geometry shader. The vertex shader processes one vertex's attributes independently of other vertices. Typical operations are position space transforms and color and texture coordinate generation. The geometry shader follows the vertex shader and deals with a whole primitive and its vertices. Typical operations are edge extrusion for

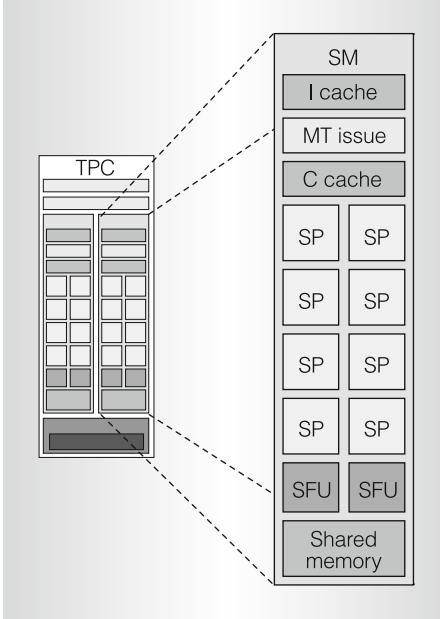


Figure 3. Streaming multiprocessor (SM).

stencil shadow generation and cube map texture generation. Geometry shader output primitives go to later stages for clipping, viewport transformation, and rasterization into pixel fragments.

Streaming multiprocessor

The SM is a unified graphics and computing multiprocessor that executes vertex, geometry, and pixel-fragment shader programs and parallel computing programs. As Figure 3 shows, the SM consists of eight streaming processor (SP) cores, two special-function units (SFUs), a multithreaded instruction fetch and issue unit (MT Issue), an instruction cache, a read-only constant cache, and a 16-Kbyte read/write shared memory.

The shared memory holds graphics input buffers or shared data for parallel computing. To pipeline graphics workloads through the SM, vertex, geometry, and pixel threads have independent input and output buffers. Workloads can arrive and depart independently of thread execution. Geometry threads, which generate variable amounts of output per thread, use separate output buffers.

Each SP core contains a scalar multiply-add (MAD) unit, giving the SM eight MAD units. The SM uses its two SFU units

for transcendental functions and attribute interpolation—the interpolation of pixel attributes from vertex attributes defining a primitive. Each SFU also contains four floating-point multipliers. The SM uses the TPC texture unit as a third execution unit and uses the SMC and ROP units to implement external memory load, store, and atomic accesses. A low-latency interconnect network between the SPs and the shared-memory banks provides shared-memory access.

The GeForce 8800 Ultra clocks the SPs and SFU units at 1.5 GHz, for a peak of 36 Gflops per SM. To optimize power and area efficiency, some SM non-data-path units operate at half the SP clock rate.

SM multithreading. A graphics vertex or pixel shader is a program for a single thread that describes how to process a vertex or a pixel. Similarly, a CUDA kernel is a C program for a single thread that describes how one thread computes a result. Graphics and computing applications instantiate many parallel threads to render complex images and compute large result arrays. To dynamically balance shifting vertex and pixel shader thread workloads, the unified SM concurrently executes different thread programs and different types of shader programs.

To efficiently execute hundreds of threads in parallel while running several different programs, the SM is hardware multithreaded. It manages and executes up to 768 concurrent threads in hardware with zero scheduling overhead.

To support the independent vertex, primitive, pixel, and thread programming model of graphics shading languages and the CUDA C/C++ language, each SM thread has its own thread execution state and can execute an independent code path. Concurrent threads of computing programs can synchronize at a barrier with a single SM instruction. Lightweight thread creation, zero-overhead thread scheduling, and fast barrier synchronization support very fine-grained parallelism efficiently.

Single-instruction, multiple-thread. To manage and execute hundreds of threads running

several different programs efficiently, the Tesla SM uses a new processor architecture we call single-instruction, multiple-thread (SIMT). The SM's SIMT multithreaded instruction unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. The term *warp* originates from weaving, the first parallel-thread technology. Figure 4 illustrates SIMT scheduling. The SIMT warp size of 32 parallel threads provides efficiency on plentiful fine-grained pixel threads and computing threads.

Each SM manages a pool of 24 warps, with a total of 768 threads. Individual threads composing a SIMT warp are of the same type and start together at the same program address, but they are otherwise free to branch and execute independently. At each instruction issue time, the SIMT multithreaded instruction unit selects a warp that is ready to execute and issues the next instruction to that warp's active threads. A SIMT instruction is broadcast synchronously to a warp's active parallel threads; individual threads can be inactive due to independent branching or predication.

The SM maps the warp threads to the SP cores, and each thread executes independently with its own instruction address and register state. A SIMT processor realizes full efficiency and performance when all 32 threads of a warp take the same execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads reconverge to the original execution path. The SM uses a branch synchronization stack to manage independent threads that diverge and converge. Branch divergence only occurs within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths. As a result, Tesla architecture GPUs are dramatically more efficient and flexible on branching code than previous generation GPUs, as their 32-thread warps are much narrower than the SIMD width of prior GPUs.¹

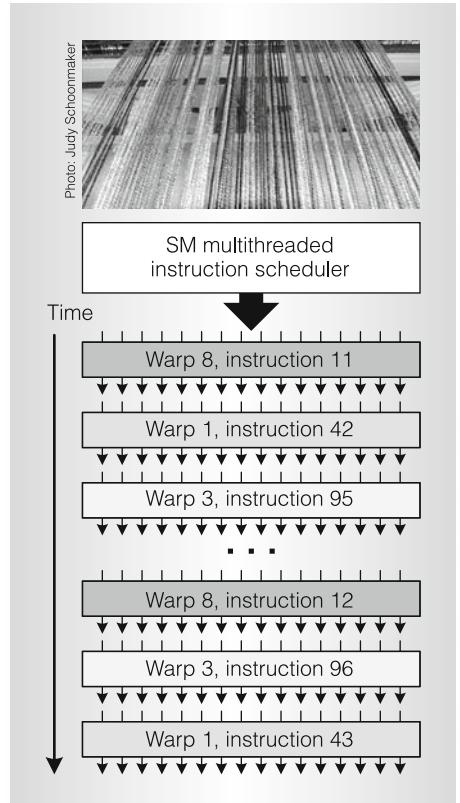


Figure 4. Single-instruction, multiple-thread (SIMT) warp scheduling.

SIMT architecture is similar to single-instruction, multiple-data (SIMD) design, which applies one instruction to multiple data lanes. The difference is that SIMT applies one instruction to multiple independent threads in parallel, not just multiple data lanes. A SIMD instruction controls a vector of multiple data lanes together and exposes the vector width to the software, whereas a SIMT instruction controls the execution and branching behavior of one thread.

In contrast to SIMD vector architectures, SIMT enables programmers to write thread-level parallel code for independent threads as well as data-parallel code for coordinated threads. For program correctness, programmers can essentially ignore SIMT execution attributes such as warps; however, they can achieve substantial performance improvements by writing code that seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines in

traditional codes: Programmers can safely ignore cache line size when designing for correctness but must consider it in the code structure when designing for peak performance. SIMD vector architectures, on the other hand, require the software to manually coalesce loads into vectors and to manually manage divergence.

SIMT warp scheduling. The SIMT approach of scheduling independent warps is simpler than previous GPU architectures' complex scheduling. A warp consists of up to 32 threads of the same type—vertex, geometry, pixel, or compute. The basic unit of pixel-fragment shader processing is the 2×2 pixel quad. The SM controller groups eight pixel quads into a warp of 32 threads. It similarly groups vertices and primitives into warps and packs 32 computing threads into a warp. The SIMT design shares the SM instruction fetch and issue unit efficiently across 32 threads but requires a full warp of active threads for full performance efficiency.

As a unified graphics processor, the SM schedules and executes multiple warp types concurrently—for example, concurrently executing vertex and pixel warps. The SM warp scheduler operates at half the 1.5-GHz processor clock rate. At each cycle, it selects one of the 24 warps to execute a SIMT warp instruction, as Figure 4 shows. An issued warp instruction executes as two sets of 16 threads over four processor cycles. The SP cores and SFU units execute instructions independently, and by issuing instructions between them on alternate cycles, the scheduler can keep both fully occupied.

Implementing zero-overhead warp scheduling for a dynamic mix of different warp programs and program types was a challenging design problem. A scoreboard qualifies each warp for issue each cycle. The instruction scheduler prioritizes all ready warps and selects the one with highest priority for issue. Prioritization considers warp type, instruction type, and “fairness” to all warps executing in the SM.

SM instructions. The Tesla SM executes scalar instructions, unlike previous GPU vector instruction architectures. Shader

programs are becoming longer and more scalar, and it is increasingly difficult to fully occupy even two components of the prior four-component vector architecture. Previous architectures employed vector packing—combining sub-vectors of work to gain efficiency—but that complicated the scheduling hardware as well as the compiler. Scalar instructions are simpler and compiler friendly. Texture instructions remain vector based, taking a source coordinate vector and returning a filtered color vector.

High-level graphics and computing-language compilers generate intermediate instructions, such as DX10 vector or PTX scalar instructions,^{10,2} which are then optimized and translated to binary GPU instructions. The optimizer readily expands DX10 vector instructions to multiple Tesla SM scalar instructions. PTX scalar instructions optimize to Tesla SM scalar instructions about one to one. PTX provides a stable target ISA for compilers and provides compatibility over several generations of GPUs with evolving binary instruction set architectures. Because the intermediate languages use virtual registers, the optimizer analyzes data dependencies and allocates real registers. It eliminates dead code, folds instructions together when feasible, and optimizes SIMT branch divergence and convergence points.

Instruction set architecture. The Tesla SM has a register-based instruction set including floating-point, integer, bit, conversion, transcendental, flow control, memory load/store, and texture operations.

Floating-point and integer operations include add, multiply, multiply-add, minimum, maximum, compare, set predicate, and conversions between integer and floating-point numbers. Floating-point instructions provide source operand modifiers for negation and absolute value. Transcendental function instructions include cosine, sine, binary exponential, binary logarithm, reciprocal, and reciprocal square root. Attribute interpolation instructions provide efficient generation of pixel attributes. Bitwise operators include shift left, shift right, logic operators, and move. Control

flow includes branch, call, return, trap, and barrier synchronization.

The floating-point and integer instructions can also set per-thread status flags for zero, negative, carry, and overflow, which the thread program can use for conditional branching.

Memory access instructions. The texture instruction fetches and filters texture samples from memory via the texture unit. The ROP unit writes pixel-fragment output to memory.

To support computing and C/C++ language needs, the Tesla SM implements memory load/store instructions in addition to graphics texture fetch and pixel output. Memory load/store instructions use integer byte addressing with register-plus-offset address arithmetic to facilitate conventional compiler code optimizations.

For computing, the load/store instructions access three read/write memory spaces:

- *local* memory for per-thread, private, temporary data (implemented in external DRAM);
- *shared* memory for low-latency access to data shared by cooperating threads in the same SM; and
- *global* memory for data shared by all threads of a computing application (implemented in external DRAM).

The memory instructions load-global, store-global, load-shared, store-shared, load-local, and store-local access global, shared, and local memory. Computing programs use the fast barrier synchronization instruction to synchronize threads within the SM that communicate with each other via shared and global memory.

To improve memory bandwidth and reduce overhead, the local and global load/store instructions coalesce individual parallel thread accesses from the same warp into fewer memory block accesses. The addresses must fall in the same block and meet alignment criteria. Coalescing memory requests boosts performance significantly over separate requests. The large thread count, together with support for many outstanding load requests, helps cover

load-to-use latency for local and global memory implemented in external DRAM.

The latest Tesla architecture GPUs provide efficient atomic memory operations, including integer add, minimum, maximum, logic operators, swap, and compare-and-swap operations. Atomic operations facilitate parallel reductions and parallel data structure management.

Streaming processor. The SP core is the primary thread processor in the SM. It performs the fundamental floating-point operations, including add, multiply, and multiply-add. It also implements a wide variety of integer, comparison, and conversion operations. The floating-point add and multiply operations are compatible with the IEEE 754 standard for single-precision FP numbers, including not-a-number (NaN) and infinity values. The unit is fully pipelined, and latency is optimized to balance delay and area.

The add and multiply operations use IEEE round-to-nearest-even as the default rounding mode. The multiply-add operation performs a multiplication with truncation, followed by an add with round-to-nearest-even. The SP flushes denormal source operands to sign-preserved zero and flushes results that underflow the target output exponent range to sign-preserved zero after rounding.

Special-function unit. The SFU supports computation of both transcendental functions and planar attribute interpolation.¹¹ A traditional vertex or pixel shader design contains a functional unit to compute transcendental functions. Pixels also need an attribute-interpolating unit to compute the per-pixel attribute values at the pixel's *x*, *y* location, given the attribute values at the primitive's vertices.

For functional evaluation, we use quadratic interpolation based on enhanced minimax approximations to approximate the reciprocal, reciprocal square root, $\log_2 x$, 2^x , and \sin/\cos functions. Table 1 shows the accuracy of the function estimates. The SFU unit generates one 32-bit floating point result per cycle.

Table 1. Function approximation statistics.

Function	Input interval	Accuracy (good bits)	ULP* error	% exactly rounded	Monotonic
$1/x$	[1, 2)	24.02	0.98	87	Yes
$1/\sqrt{x}$	[1, 4)	23.40	1.52	78	Yes
2^x	[0, 1)	22.51	1.41	74	Yes
$\log_2 x$	[1, 2)	22.57	N/A**	N/A	Yes
\sin/\cos	[0, $\pi/2)$	22.47	N/A	N/A	No

* ULP: unit-in-the-last-place.

** N/A: not applicable.

The SFU also supports attribute interpolation, to enable accurate interpolation of attributes such as color, depth, and texture coordinates. The SFU must interpolate these attributes in the (x, y) screen space to determine the values of the attributes at each pixel location. We express the value of a given attribute U in an (x, y) plane in plane equations of the following form:

$$U(x, y) = \frac{(A_U \times x + B_U \times y + C_U)}{(A_W \times x + B_W \times y + C_W)}$$

where A , B , and C are interpolation parameters associated with each attribute U , and W is related to the distance of the pixel from the viewer for perspective projection. The attribute interpolation hardware in the SFU is fully pipelined, and it can interpolate four samples per cycle.

In a shader program, the SFU can generate perspective-corrected attributes as follows:

- Interpolate $1/W$, and invert to form W .
- Interpolate U/W .
- Multiply U/W by W to form perspective-correct U .

SM controller. The SMC controls multiple SMs, arbitrating the shared texture unit, load/store path, and I/O path. The SMC serves three graphics workloads simulta-

neously: vertex, geometry, and pixel. It packs each of these input types into the warp width, initiating shader processing, and unpacks the results.

Each input type has independent I/O paths, but the SMC is responsible for load balancing among them. The SMC supports static and dynamic load balancing based on driver-recommended allocations, current allocations, and relative difficulty of additional resource allocation. Load balancing of the workloads was one of the more challenging design problems due to its impact on overall SPA efficiency.

Texture unit

The texture unit processes one group of four threads (vertex, geometry, pixel, or compute) per cycle. Texture instruction sources are texture coordinates, and the outputs are filtered samples, typically a four-component (RGBA) color. Texture is a separate unit external to the SM connected via the SMC. The issuing SM thread can continue execution until a data dependency stall.

Each texture unit has four texture address generators and eight filter units, for a peak GeForce 8800 Ultra rate of 38.4 gigabilerps/s (a bilerp is a bilinear interpolation of four samples). Each unit supports full-speed 2:1 anisotropic filtering, as well as high-dynamic-range (HDR) 16-bit and 32-bit floating-point data format filtering.

The texture unit is deeply pipelined. Although it contains a cache to capture filtering locality, it streams hits mixed with misses without stalling.

Rasterization

Geometry primitives output from the SMs go in their original round-robin input order to the viewport/clip/setup/raster/zcull block. The viewport and clip units clip the primitives to the standard view frustum and to any enabled user clip planes. They transform postclipping vertices into screen (pixel) space and reject whole primitives outside the view volume as well as back-facing primitives.

Surviving primitives then go to the setup unit, which generates edge equations for the rasterizer. Attribute plane equations are also generated for linear interpolation of pixel attributes in the pixel shader. A coarse-rasterization stage generates all pixel tiles that are at least partially inside the primitive.

The zcull unit maintains a hierarchical z surface, rejecting pixel tiles if they are conservatively known to be occluded by previously drawn pixels. The rejection rate is up to 256 pixels per clock. The screen is subdivided into tiles; each TPC processes a predetermined subset. The pixel tile address therefore selects the destination TPC. Pixel tiles that survive zcull then go to a fine-rasterization stage that generates detailed coverage information and depth values for the pixels.

OpenGL and Direct3D require that a depth test be performed after the pixel shader has generated final color and depth values. When possible, for certain combinations of API state, the Tesla GPU performs the depth test and update ahead of the fragment shader, possibly saving thousands of cycles of processing time, without violating the API-mandated semantics.

The SMC assembles surviving pixels into warps to be processed by a SM running the current pixel shader. When the pixel shader has finished, the pixels are optionally depth tested if this was not done ahead of the shader. The SMC then sends surviving pixels and associated data to the ROP.

Raster operations processor

Each ROP is paired with a specific memory partition. The TPCs feed data to the ROPs via an interconnection network.

ROPs handle depth and stencil testing and updates and color blending and updates. The memory controller uses lossless color (up to 8:1) and depth compression (up to 8:1) to reduce bandwidth. Each ROP has a peak rate of four pixels per clock and supports 16-bit floating-point and 32-bit floating-point HDR formats. ROPs support double-rate-depth processing when color writes are disabled.

Each memory partition is 64 bits wide and supports double-data-rate DDR2 and graphics-oriented GDDR3 protocols at up to 1 GHz, yielding a bandwidth of about 16 Gbytes/s.

Antialiasing support includes up to $16\times$ multisampling and supersampling. HDR formats are fully supported. Both algorithms support 1, 2, 4, 8, or 16 samples per pixel and generate a weighted average of the samples to produce the final pixel color. Multisampling executes the pixel shader once to generate a color shared by all pixel samples, whereas supersampling runs the pixel shader once per sample. In both cases, depth values are correctly evaluated for each sample, as required for correct interpenetration of primitives.

Because multisampling runs the pixel shader once per pixel (rather than once per sample), multisampling has become the most popular antialiasing method. Beyond four samples, however, storage cost increases faster than image quality improves, especially with HDR formats. For example, a single $1,600 \times 1,200$ pixel surface, storing 16 four-component, 16-bit floating-point samples, requires $1,600 \times 1,200 \times 16 \times (64 \text{ bits color} + 32 \text{ bits depth}) = 368 \text{ Mbytes}$.

For the vast majority of edge pixels, two colors are enough; what matters is more-detailed coverage information. The coverage-sampling antialiasing (CSAA) algorithm provides low-cost-per-coverage samples, allowing upward scaling. By computing and storing Boolean coverage at up to 16 samples and compressing redundant color and depth and stencil information into the memory footprint and bandwidth of four or eight samples, $16\times$ antialiasing quality can be achieved at $4\times$ antialiasing performance. CSAA is compatible with existing rendering

Table 2. Comparison of antialiasing modes.

Feature	Antialiasing mode								
	Brute-force supersampling			Multisampling			Coverage sampling		
Quality level	1×	4×	16×	1×	4×	16×	1×	4×	16×
Texture and shader samples	1	4	16	1	1	1	1	1	1
Stored color and z samples	1	4	16	1	4	16	1	4	4
Coverage samples	1	4	16	1	4	16	1	4	16

techniques including HDR and stencil algorithms. Edges defined by the intersection of interpenetrating polygons are rendered at the stored sample count quality ($4\times$ or $8\times$). Table 2 summarizes the storage requirements of the three algorithms.

Memory and interconnect

The DRAM memory data bus width is 384 pins, arranged in six independent partitions of 64 pins each. Each partition owns 1/6 of the physical address space. The memory partition units directly enqueue requests. They arbitrate among hundreds of in-flight requests from the parallel stages of the graphics and computation pipelines. The arbitration seeks to maximize total DRAM transfer efficiency, which favors grouping related requests by DRAM bank and read/write direction, while minimizing latency as far as possible. The memory controllers support a wide range of DRAM clock rates, protocols, device densities, and data bus widths.

Interconnection network. A single hub unit routes requests to the appropriate partition from the nonparallel requesters (PCI-Express, host and command front end, input assembler, and display). Each memory partition has its own depth and color ROP units, so ROP memory traffic originates locally. Texture and load/store requests, however, can occur between any TPC and any memory partition, so an interconnection network routes requests and responses.

Memory management unit. All processing engines generate addresses in a virtual address space. A memory management unit

performs virtual to physical translation. Hardware reads the page tables from local memory to respond to misses on behalf of a hierarchy of translation look-aside buffers spread out among the rendering engines.

Parallel computing architecture

The Tesla scalable parallel computing architecture enables the GPU processor array to excel in throughput computing, executing high-performance computing applications as well as graphics applications. Throughput applications have several properties that distinguish them from CPU serial applications:

- extensive data parallelism—thousands of computations on independent data elements;
- modest task parallelism—groups of threads execute the same program, and different groups can run different programs;
- intensive floating-point arithmetic;
- latency tolerance—performance is the amount of work completed in a given time;
- streaming data flow—requires high memory bandwidth with relatively little data reuse;
- modest inter-thread synchronization and communication—graphics threads do not communicate, and parallel computing applications require limited synchronization and communication.

GPU parallel performance on throughput problems has doubled every 12 to 18 months, pulled by the insatiable demands of the 3D game market. Now, Tesla GPUs in laptops, desktops, workstations,

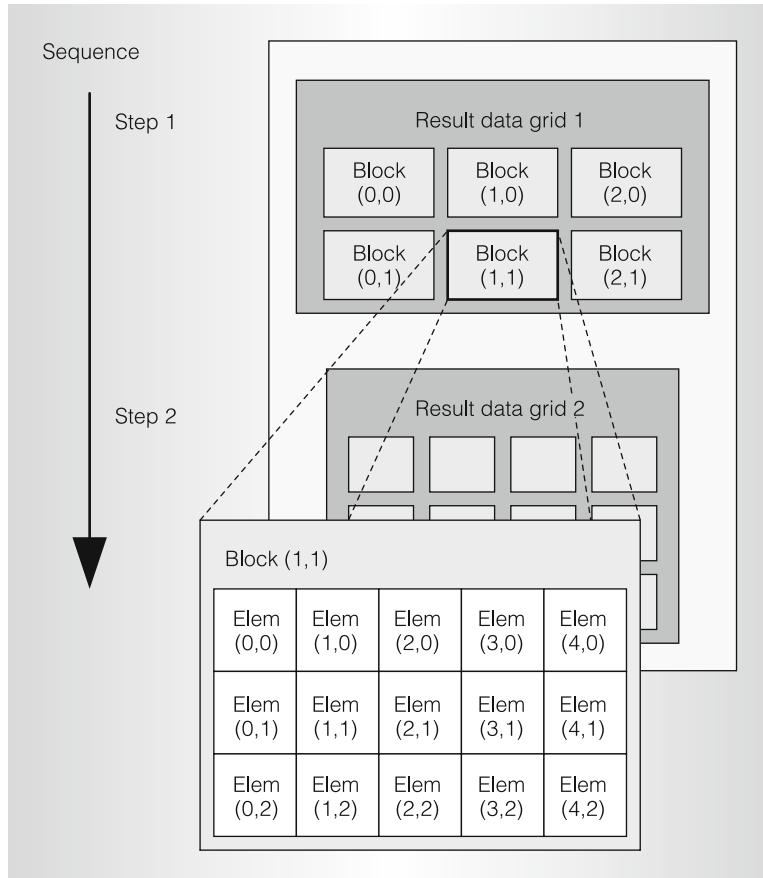


Figure 5. Decomposing result data into a grid of blocks partitioned into elements to be computed in parallel.

and systems are programmable in C with CUDA tools, using a simple parallel programming model.

Data-parallel problem decomposition

To map a large computing problem effectively to a highly parallel processing architecture, the programmer or compiler decomposes the problem into many small problems that can be solved in parallel. For example, the programmer partitions a large result data array into blocks and further partitions each block into elements, so that the result blocks can be computed independently in parallel, and the elements within each block can be computed cooperatively in parallel. Figure 5 shows the decomposition of a result data array into a 3×2 grid of blocks, in which each block is further decomposed into a 5×3 array of elements.

The two-level parallel decomposition maps naturally to the Tesla architecture: Parallel SMs compute result blocks, and parallel threads compute result elements.

The programmer or compiler writes a program that computes a sequence of result grids, partitioning each result grid into coarse-grained result blocks that are computed independently in parallel. The program computes each result block with an array of fine-grained parallel threads, partitioning the work among threads that compute result elements.

Cooperative thread array or thread block

Unlike the graphics programming model, which executes parallel shader threads independently, parallel-computing programming models require that parallel threads synchronize, communicate, share data, and cooperate to efficiently compute a result. To manage large numbers of concurrent threads that can cooperate, the Tesla computing architecture introduces the *cooperative thread array* (CTA), called a *thread block* in CUDA terminology.

A CTA is an array of concurrent threads that execute the same thread program and can cooperate to compute a result. A CTA consists of 1 to 512 concurrent threads, and each thread has a unique thread ID (TID), numbered 0 through m . The programmer declares the 1D, 2D, or 3D CTA shape and dimensions in threads. The TID has one, two, or three dimension indices. Threads of a CTA can share data in global or shared memory and can synchronize with the barrier instruction. CTA thread programs use their TIDs to select work and index shared data arrays. Multidimensional TIDs can eliminate integer divide and remainder operations when indexing arrays.

Each SM executes up to eight CTAs concurrently, depending on CTA resource demands. The programmer or compiler declares the number of threads, registers, shared memory, and barriers required by the CTA program. When an SM has sufficient available resources, the SMC creates the CTA and assigns TID numbers to each thread. The SM executes the CTA threads concurrently as SIMT warps of 32 parallel threads.

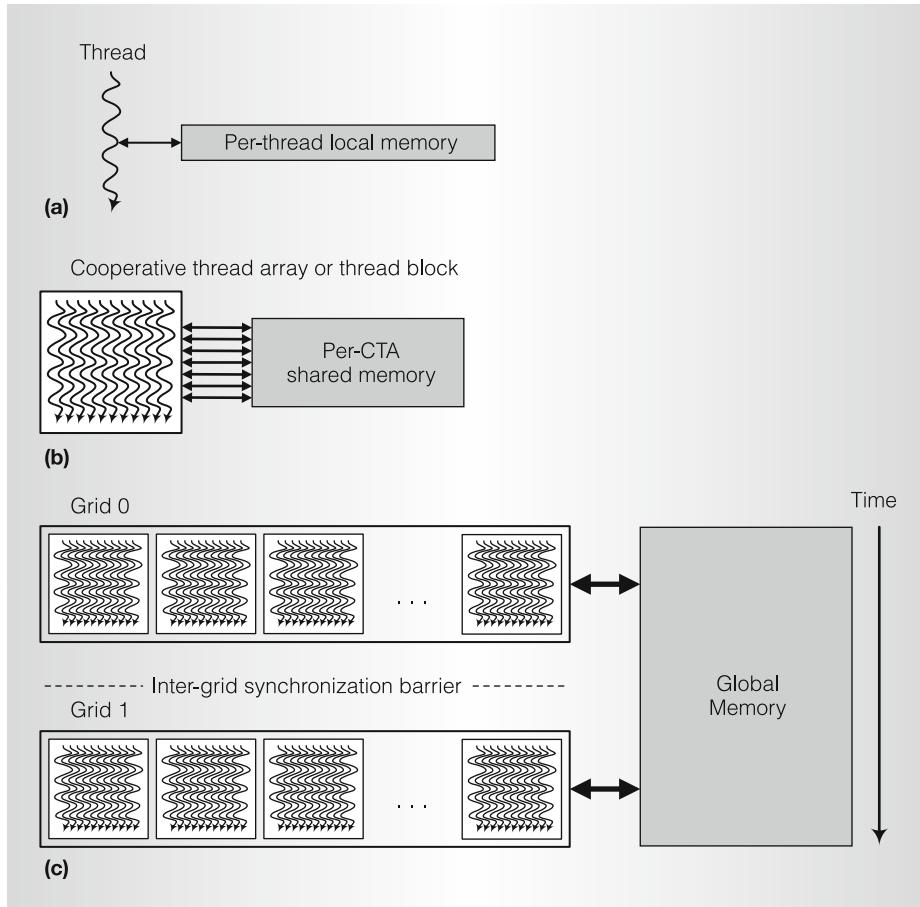


Figure 6. Nested granularity levels: thread (a), cooperative thread array (b), and grid (c). These have corresponding memory-sharing levels: local per-thread, shared per-CTA, and global per-application.

CTA grids

To implement the coarse-grained block and grid decomposition of Figure 5, the GPU creates CTAs with unique CTA ID and grid ID numbers. The compute work distributor dynamically balances the GPU workload by distributing a stream of CTA work to SMs with sufficient available resources.

To enable a compiled binary program to run unchanged on large or small GPUs with any number of parallel SM processors, CTAs execute independently and compute result blocks independently of other CTAs in the same grid. Sequentially dependent application steps map to two sequentially dependent grids. The dependent grid waits for the first grid to complete; then the CTAs of the dependent grid read the result blocks written by the first grid.

Parallel granularity

Figure 6 shows levels of parallel granularity in the GPU computing model. The three levels are

- *thread*—computes result elements selected by its TID;
- *CTA*—computes result blocks selected by its CTA ID;
- *grid*—computes many result blocks, and sequential grids compute sequentially dependent application steps.

Higher levels of parallelism use multiple GPUs per CPU and clusters of multi-GPU nodes.

Parallel memory sharing

Figure 6 also shows levels of parallel read/write memory sharing:

- *local*—each executing thread has a private per-thread local memory for register spill, stack frame, and addressable temporary variables;
- *shared*—each executing CTA has a per-CTA shared memory for access to data shared by threads in the same CTA;
- *global*—sequential grids communicate and share large data sets in global memory.

Threads communicating in a CTA use the fast barrier synchronization instruction to wait for writes to shared or global memory to complete before reading data written by other threads in the CTA. The load/store memory system uses a relaxed memory order that preserves the order of reads and writes to the same address from the same issuing thread and from the viewpoint of CTA threads coordinating with the barrier synchronization instruction. Sequentially dependent grids use a global intergrid synchronization barrier between grids to ensure global read/write ordering.

Transparent scaling of GPU computing

Parallelism varies widely over the range of GPU products developed for various market segments. A small GPU might have one SM with eight SP cores, while a large GPU might have many SMs totaling hundreds of SP cores.

The GPU computing architecture transparently scales parallel application performance with the number of SMs and SP cores. A GPU computing program executes on any size of GPU without recompiling, and is insensitive to the number of SM multiprocessors and SP cores. The program does not know or care how many processors it uses.

The key is decomposing the problem into independently computed blocks as described earlier. The GPU compute work distribution unit generates a stream of CTAs and distributes them to available SMs to compute each independent block. Scalable programs do not communicate among CTA blocks of the same grid; the same grid result is obtained if the CTAs execute in parallel on many cores, sequen-

tially on one core, or partially in parallel on a few cores.

CUDA programming model

CUDA is a minimal extension of the C and C++ programming languages. A programmer writes a serial program that calls parallel kernels, which can be simple functions or full programs. The CUDA program executes serial code on the CPU and executes parallel kernels across a set of parallel threads on the GPU. The programmer organizes these threads into a hierarchy of thread blocks and grids as described earlier. (A CUDA thread block is a GPU CTA.)

Figure 7 shows a CUDA program executing a series of parallel kernels on a heterogeneous CPU–GPU system. **KernelA** and **KernelB** execute on the GPU as grids of **nBlkA** and **nBlkB** thread blocks (CTAs), which instantiate **nTidA** and **nTidB** threads per CTA.

The CUDA compiler **nvcc** compiles an integrated application C/C++ program containing serial CPU code and parallel GPU kernel code. The CUDA runtime API manages the GPU as a computing device that acts as a coprocessor to the host CPU with its own memory system.

The CUDA programming model is similar in style to a single-program multiple-data (SPMD) software model—it expresses parallelism explicitly, and each kernel executes on a fixed number of threads. However, CUDA is more flexible than most SPMD implementations because each kernel call dynamically creates a new grid with the right number of thread blocks and threads for that application step.

CUDA extends C/C++ with the declaration specifier keywords **global** for kernel entry functions, **device** for global variables, and **shared** for shared-memory variables. A CUDA kernel's text is simply a C function for one sequential thread. The built-in variables **threadIdx.{x, y, z}** and **blockIdx.{x, y, z}** provide the thread ID within a thread block (CTA), while **blockIdx** provides the CTA ID within a grid. The extended function call syntax **kernel<<<nBlocks,nThreads>>>(args)**;

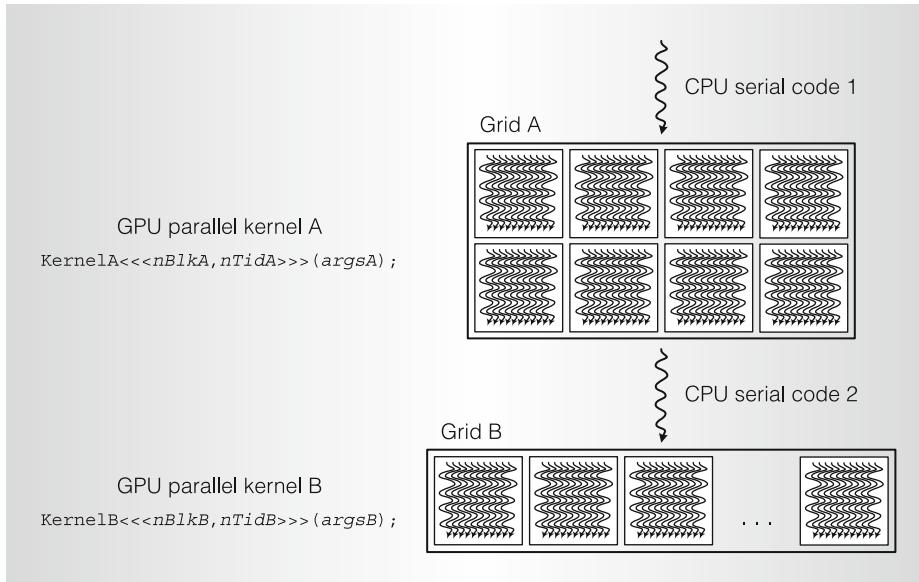


Figure 7. CUDA program sequence of kernel A followed by kernel B on a heterogeneous CPU–GPU system.

invokes a parallel kernel function on a grid of **nBlocks**, where each block instantiates **nThreads** concurrent threads, and **args** are ordinary arguments to function **kernel()**.

Figure 8 shows an example serial C program and a corresponding CUDA C program. The serial C program uses two nested loops to iterate over each array index and compute **c[idx] = a[idx] + b[idx]** each trip. The parallel CUDA C program has no loops.

It uses parallel threads to compute the same array indices in parallel, and each thread computes only one sum.

Scalability and performance

The Tesla unified architecture is designed for scalability. Varying the number of SMs, TPCs, ROPs, caches, and memory partitions provides the right mix for different performance and cost targets in the value, mainstream, enthusiast, and professional

```
void addMatrix
    (float *a, float *b, float *c, int N)
{
    int i, j, idx;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            idx = i + j*N;
            c[idx] = a[idx] + b[idx];
        }
    }
}
void main()
{
    ...
    addMatrix(a, b, c, N);
}
```

(a)

```
__global__ void addMatrixG
    (float *a, float *b, float *c, int N)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int j = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = i + j*N;
    if (i < N && j < N)
        c[idx] = a[idx] + b[idx];
}

void main()
{
    dim3 dimBlock (blocksize, blocksize);
    dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
    addMatrixG<<<dimGrid, dimBlock>>>(a, b, c, N);
}
```

(b)

Figure 8. Serial C (a) and CUDA C (b) examples of programs that add arrays.

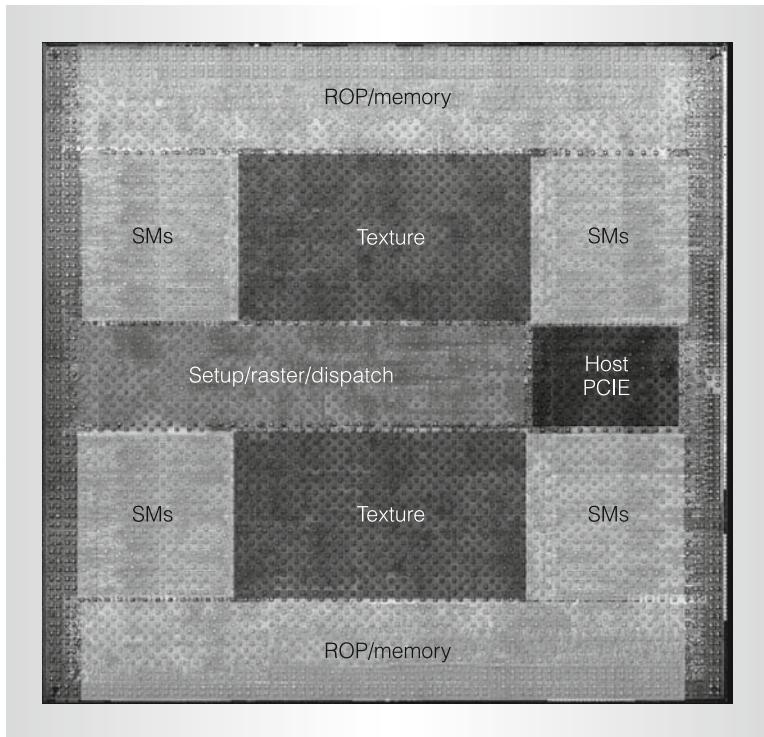


Figure 9. GeForce 8800 Ultra die layout.

market segments. NVIDIA's Scalable Link Interconnect (SLI) enables multiple GPUs to act together as one, providing further scalability.

CUDA C/C++ applications executing on Tesla computing platforms, Quadro workstations, and GeForce GPUs deliver compelling computing performance on a range of large problems, including more than 100 \times speedups on molecular modeling, more than 200 Gflops on n -body problems, and real-time 3D magnetic-resonance imaging.^{12–14} For graphics, the GeForce 8800 GPU delivers high performance and image quality for the most demanding games.¹⁵

Figure 9 shows the GeForce 8800 Ultra physical die layout implementing the Tesla architecture shown in Figure 1. Implementation specifics include

- 681 million transistors, 470 mm²;
- TSMC 90-nm CMOS;
- 128 SP cores in 16 SMs;
- 12,288 processor threads;
- 1.5-GHz processor clock rate;
- peak 576 Gflops in processors;
- 768-Mbyte GDDR3 DRAM;

- 384-pin DRAM interface;
- 1.08-GHz DRAM clock;
- 104-Gbyte/s peak bandwidth; and
- typical power of 150 W at 1.3 V.

The Tesla architecture is the first ubiquitous supercomputing platform. NVIDIA has shipped more than 50 million Tesla-based systems. This wide availability, coupled with C programmability and the CUDA software development environment, enables broad deployment of demanding parallel-computing and graphics applications.

With future increases in transistor density, the architecture will readily scale processor parallelism, memory partitions, and overall performance. Increased number of multiprocessors and memory partitions will support larger data sets and richer graphics and computing, without a change to the programming model.

We continue to investigate improved scheduling and load-balancing algorithms for the unified processor. Other areas of improvement are enhanced scalability for derivative products, reduced synchronization and communication overhead for compute programs, new graphics features, increased realized memory bandwidth, and improved power efficiency. MICRO

Acknowledgments

We thank the entire NVIDIA GPU development team for their extraordinary effort in bringing Tesla-based GPUs to market.

References

1. J. Montrym and H. Moreton, "The GeForce 6800," *IEEE Micro*, vol. 25, no. 2, Mar./Apr. 2005, pp. 41–51.
2. *CUDA Technology*, NVIDIA, 2007, <http://www.nvidia.com/CUDA>.
3. *CUDA Programming Guide 1.1*, NVIDIA, 2007; http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf.
4. J. Nickolls, I. Buck, K. Skadron, and M. Garland, "Scalable Parallel Programming with CUDA," *ACM Queue*, vol. 6, no. 2, Mar./Apr. 2008, pp. 40–53.
5. *DX Specification*, Microsoft; <http://msdn.microsoft.com/directx>.

6. E. Lindholm, M.J. Kilgard, and H. Moreton, "A User-Programmable Vertex Engine," *Proc. 28th Ann. Conf. Computer Graphics and Interactive Techniques* (Siggraph 01), ACM Press, 2001, pp. 149-158.
7. G. Elder, "Radeon 9700," Eurographics/Siggraph Workshop Graphics Hardware, Hot 3D Session, 2002, http://www.graphicshardware.org/previous/www_2002/presentations/Hot3D-RADEON9700.ppt.
8. *Microsoft DirectX 9 Programmable Graphics Pipeline*, Microsoft Press, 2003.
9. J. Andrews and N. Baker, "Xbox 360 System Architecture," *IEEE Micro*, vol. 26, no. 2, Mar./Apr. 2006, pp. 25-37.
10. D. Blythe, "The Direct3D 10 System," *ACM Trans. Graphics*, vol. 25, no. 3, July 2006, pp. 724-734.
11. S.F. Oberman and M.Y. Siu, "A High-Performance Area-Efficient Multifunction Interpolator," *Proc. 17th IEEE Symp. Computer Arithmetic* (Arith-17), IEEE Press, 2005, pp. 272-279.
12. J.E. Stone et al., "Accelerating Molecular Modeling Applications with Graphics Processors," *J. Computational Chemistry*, vol. 28, no. 16, 2007, pp. 2618-2640.
13. L. Nyland, M. Harris, and J. Prins, "Fast N-Body Simulation with CUDA," *GPU Gems* 3, H. Nguyen, ed., Addison-Wesley, 2007, pp. 677-695.
14. S.S. Stone et al., "How GPUs Can Improve the Quality of Magnetic Resonance Imaging," *Proc. 1st Workshop on General Purpose Processing on Graphics Processing Units*, 2007; <http://www.gigascale.org/pubs/1175.html>.
15. A.L. Shimpi and D. Wilson, "NVIDIA's GeForce 8800 (G80): GPUs Re-architected for DirectX 10," *AnandTech*, Nov. 2006; <http://www.anandtech.com/video/showdoc.aspx?i=2870>.

Erik Lindholm is a distinguished engineer at NVIDIA, working in the architecture

group. His research interests include graphics processor design and parallel graphics architectures. Lindholm has an MS in electrical engineering from the University of British Columbia.

John Nickolls is director of GPU computing architecture at NVIDIA. His interests include parallel processing systems, languages, and architectures. Nickolls has a BS in electrical engineering and computer science from the University of Illinois and MS and PhD degrees in electrical engineering from Stanford University.

Stuart Oberman is a design manager in the GPU hardware group at NVIDIA. His research interests include computer arithmetic, processor design, and parallel architectures. Oberman has a BS in electrical engineering from the University of Iowa and MS and PhD degrees in electrical engineering from Stanford University. He is a senior member of the IEEE.

John Montrym is a chief architect at NVIDIA, where he has worked in the development of several GPU product families. His research interests include graphics processor design, parallel graphics architectures, and hardware-software interfaces. Montrym has a BS in electrical engineering from the Massachusetts Institute of Technology.

Direct questions and comments about this article to Erik Lindholm or John Nickolls, NVIDIA, 2701 San Tomas Expressway, Santa Clara, CA 95050; elindholm@nvidia.com or jnickolls@nvidia.com.

For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/csdl>.

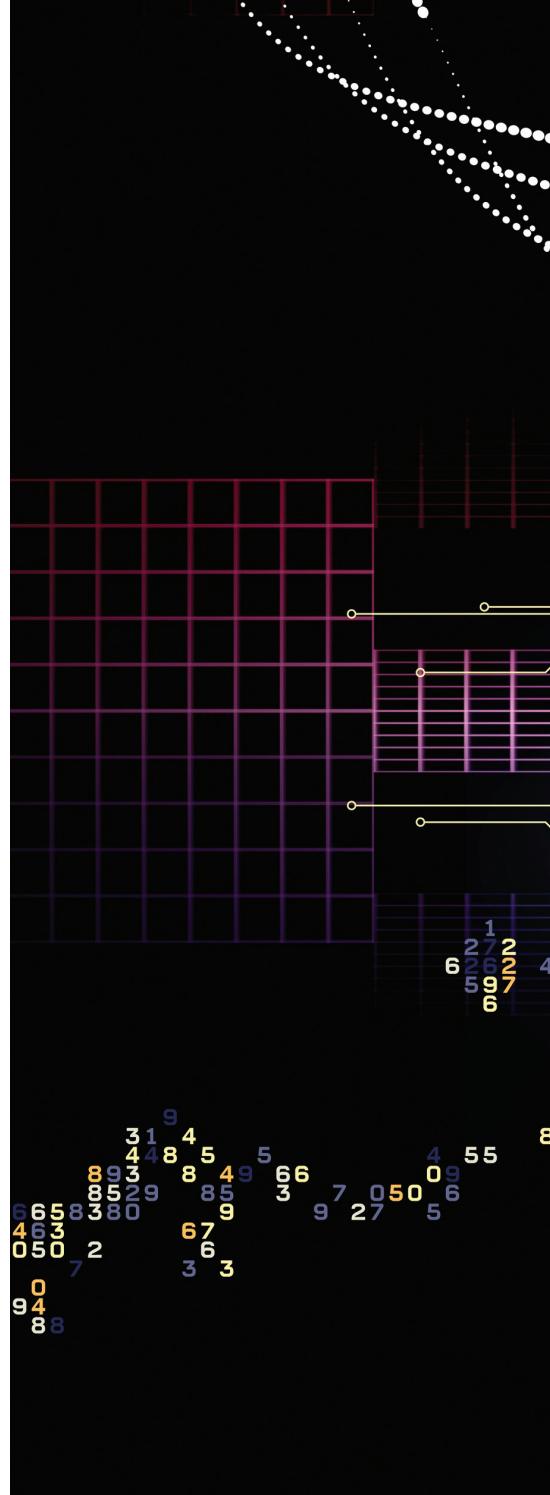
Data races are evil.

BY HANS-J. BOEHM AND SARITA V. ADVE

You Don't Know Jack About Shared Variables or Memory Models

A GOOGLE SEARCH for “threads are evil” generates 18,000 hits, but threads—evil or not—are ubiquitous. Almost all of the processes running on a modern Windows PC use them. Software threads are typically how programmers get machines with multiple cores to work together to solve problems faster. And often they are what allow user interfaces to remain responsive while the application performs a background calculation.

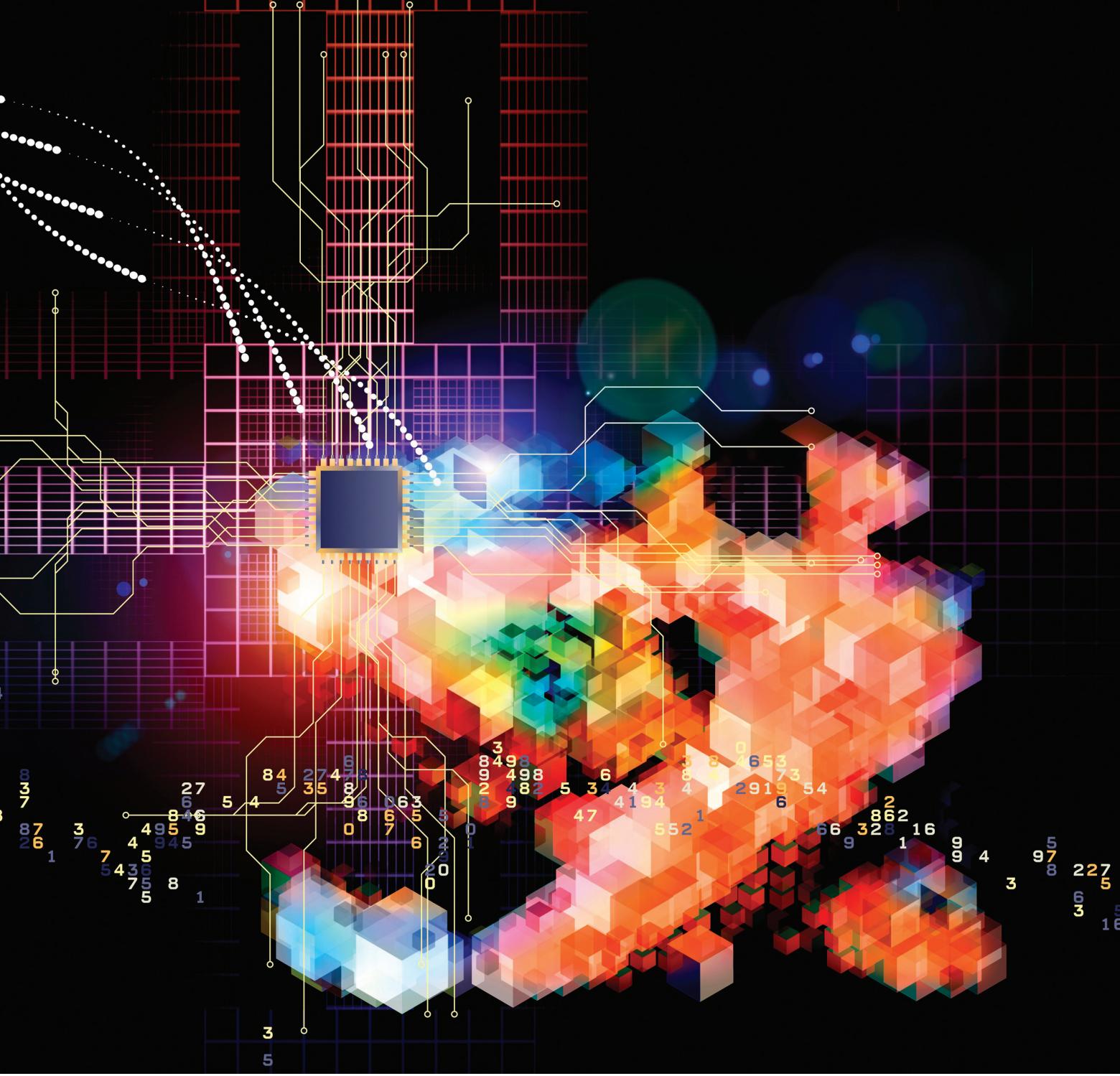
Threads are multiple programs running at the same time but sharing variables. Typically, every thread



can access all of the application’s memory. Shared variables are either the core strength of threads or the root of their evil, depending on your perspective. They allow threads to communicate easily and quickly, but they also make it possible for threads to get in each other’s way.

Although shared variables are at the core of most programs, even experts are often confused about the rules for using them. Consider the following simple example.

To implement a function `incr` that increments a counter `x`, your first at-



tempt might be

```
void incr()  
{  
    x++;  
}
```

Many would immediately object that this isn't guaranteed to produce the correct answer when called by multiple threads. The statement `x++` is equivalent to `x=x+1`, which amounts to three steps: Getting the value of `x`; adding one; and writing the result back to `x`. In the unlikely

case that two threads coincidentally perform these in lockstep, they will both read the same value, both add one to it, and then both write the same value, incrementing `x` by only one instead of two. A call to `incr()` does not behave *atomically*; it is visible to the user that it is composed of different steps. (*Atomicity* means different things to different communities; our use is called *isolation* by database folks.)

We might address the problem by using a *mutex*, which can be locked by only one thread at a time:

```
void incr()  
{  
    mtx.lock();  
    x++;  
    mtx.unlock();  
}
```

In Java, this might look like

```
void incr()  
{  
    synchronized(mtx) {  
        x++;  
    }  
}
```

or perhaps just

```
synchronized void incr()
{
    x++;
}
```

Those would all work correctly, but mutex calls can be slow, so the result may run slower than desired.

What if we are concerned only about getting an approximate count? What if we just leave off the mutex, and settle for some inaccuracy? What could go wrong?

To begin with, we observed that some actual code incrementing such a counter in two threads without a mutex routinely missed about half the counts, probably a result of unfortunate timing caused by communication between the processors' caches. It could be worse. A thread could do nothing but call `incr()` once, loading the value zero from `x` at the beginning, get suspended for a long time, and then write back one just before the program terminates. This would result in a final count of one, no matter what the other threads did.

Those are the cases that are less surprising and easier to explain. The final count can also be too high. Consider a case in which the count is bigger than a machine word. To avoid dealing with binary numbers, assume we have a decimal machine in which each word holds three digits, and the counter `x` can hold six digits. The compiler translates `x++` to something like

```
tmp_hi = x_hi;
tmp_lo = x_lo;
(tmp_hi, tmp_lo)++;
x_hi = tmp_hi;
x_lo = tmp_lo;
```

where `tmp_lo` and `tmp_hi` are machine registers, and the increment operation in the middle would really involve several machine instructions.

Now assume that `x` is 999 (`x_hi = 0`, and `x_lo = 999`), and two threads, a **blue** and a **red** one, each increment `x` as shown in Figure 1 (remember that each thread has its own copy of the machine registers `tmp_hi` and `tmp_lo`). The **blue** thread runs almost to completion; then the **red** thread runs all at once to completion; finally

Although shared variables are at the core of most programs, even experts are often confused about the rules for using them.

the **blue** thread runs its last step. The result is that we incremented 999 twice to get 2000. This is difficult to explain to a programmer who doesn't understand precisely how the code is being compiled.

The fundamental problem is that multiple threads were accessing `x` at the same time, without proper locking or other synchronization to make sure that one occurred after the other. This situation is called a *data race*—which really is evil! We will get back to avoiding data races without locks later.

Another Racy Example

We have only begun to see the problems caused by data races. Here is an example commonly tried in real code. One thread initializes a piece of data (say, `x`) and sets a flag (call it `done`) when it finishes. Any thread that later reads `x` first waits for the `done` flag, as in Figure 2. What could possibly go wrong?

This code may work reliably with a “dumb” compiler, but any “clever” optimizing compiler is likely to break it. When the compiler sees the loop, it is likely to observe that `done` is not modified in the loop (that is, it is “loop-invariant”). Thus, it gets to assume that `done` does not change in the loop.

Of course, this assumption isn't actually correct for our example, but the compiler gets to make it anyway, for two reasons: compilers were traditionally designed to compile sequential, not multithreaded code; and because, as we will see, even modern multi-threaded languages continue to allow this, for good reason.

Thus, the loop is likely to be transformed to

```
tmp = done; while (!tmp) {}
```

or maybe even

```
tmp = done; if (!tmp) while (true) {}
```

In either case, if `done` is not already set when a **red** thread starts, the **red** thread is guaranteed to enter an infinite loop.

Assume we have a “dumb” compiler that does not perform such transformations and compiles the code exactly

as written. Depending on the hardware, this code can still fail.

The problem this time is that the hardware may optimize the blue thread. Nearly all processor architectures allow stores to memory to be saved in a buffer visible only to that processor core before writing them to memory visible to other processor cores.² Some, such as the ARM chip that is probably in your smartphone, allow the stores to become visible to other processor cores in a different order. On such a processor the **blue** thread's write to `done` may become visible to the **red** thread, running on another core, *before* the **blue** thread's write to `x`. Thus, the **red** thread may see `done` set to `true`, and the loop may terminate before it can retrieve the proper value of `x`. Thus, when the **red** thread accesses `x`, it may still get the uninitialized value.

Unlike the original problem of reading `done` once outside the loop, this problem will occur infrequently, and may well be missed during testing.

Again the core problem here is that although the `done` flag is intended to prevent simultaneous accesses to `x`, it can itself be simultaneously accessed by both threads. *And data races are evil!*

Bits and Bytes

So far, we have talked only about data races in which two threads access exactly the same variable, or object field, at the same time. That has not always been the only concern. According to some older standards, when you declare two small fields `b1` and `b2` next to each other, for example, then updating `b1` could be implemented with the following steps:

1. Load the machine word containing both `b1` and `b2` into a machine register.
2. Update the `b1` piece in the machine register.
3. Store the register back to the location from which it was loaded.

Unfortunately, if another thread updates `b2` just before the last step, then that update is overwritten by the last step and effectively lost. If both fields were initially zero, and one thread executed `b1 = 1`, while the other executed `b2 = 1`, `b2` could still

be zero when they both finished. Although the original program was well behaved and had no data races, the compiler added an implicit update to `b2` that created a data race.

This kind of data-race insertion has been clearly disallowed in Java for a long time. The recently published C++11 and C11 standards also disallow it. We know of no Java implementations with such problems, nor do modern C and C++ compilers generally exhibit precisely this problem. Unfortunately, many do introduce data races under certain obscure, unlikely, and unpredictable conditions. This problem will disappear as C++11 and C11 become widely supported.

For C and C++, the story for bit-fields is slightly more complicated. We'll discuss that more, later.

And the Real Rules Are...

The simplest view of threads, and the one we started with, is that a multi-threaded program is executed by interleaving steps from each thread. Logically the computer executes a step from one thread, then picks another thread, or possibly the same one, executes its next step, and so on. This is a *sequentially consistent* execution.

As already shown, real machines and compilers sometimes result in non-sequentially consistent executions: for example, when the assignment to a variable and a `done` flag are made visible to other threads out of order. Sequential consistency, how-

ever, is critical in understanding the behavior of real shared variables, for two reasons:

- Essentially all modern languages (Java, C++11, C11) do in fact promise sequential consistency *for programs without data races*. This guarantee is normally violated by a few low-level language features—notably, Java's `lazySet()` and C++11 and C11's explicit `memory_order...` specifications, which are easy to avoid (with the possible exception of OpenMP's atomic directive) and which we'll mostly ignore here. Most programmers will also want to ignore these features.

- So far we have been a bit imprecise about what constitutes a data race. Since this has now become a critical part of our programming rules, we can make it more precise as follows: two memory operations *conflict* if they access the same memory location and at least one of the accesses is a write. For our purposes, a *memory location* is a unit of memory that is separately updatable. Normally every scalar (unstructured) variable or field occupies its own memory location; each can be independently updated. Contiguous sequences of C or C++ bit fields, however, normally share a single location; updating one potentially interferes with the others.

Two conflicting data operations form a *data race* if they are from different threads and can be executed "at the same time." But when is this possible? Clearly that depends on how

Figure 1. Two interleaved multi-word increments.

```
tmp_hi = x_hi;
tmp_lo = x_lo;
(tmp_hi, tmp_lo)++; // tmp_hi = 1, tmp_lo = 0
x_hi = tmp_hi; // x_hi = 1, x_lo = 999, x = 1999
    x++;
// red runs all steps
    // x_hi = 2, x_lo = 0, x = 2000
x_lo = tmp_lo; // x_hi = 2, x_lo = 0
```

Figure 2. Waiting on a flag.

Blue Thread

```
x = ...;
done = true;
```

Other Threads

```
while (!done) {}
... = x;
```

shared variables behave, which we're trying to define.

We break this circularity by considering only *sequentially consistent* executions: two conflicting operations in a sequentially consistent execution execute at the same time, if one appears immediately after the other in that execution's interleaving. Now we can say that a program is *data-race-free* if none of its sequentially consistent executions has a data race.

Here we have defined a *data race* in terms of data operations explicitly to exclude *synchronization* operations such as locking and unlocking a mutex. Two operations on the same mutex do not introduce a data race if they appear next to each other in the interleaving. Indeed, they could not usefully control simultaneous data accesses if concurrent accesses to the mutexes were disallowed.

Thus, the basic programming model is:

- Write code such that data races are impossible, assuming that the implementation follows sequential consistency rules.
- The implementation then guarantees sequential consistency for such code (assuming that the low-level fea-

tures previously mentioned are avoided).

This is very different from promising full sequential consistency for all programs; our earlier examples are not guaranteed to work as expected, since they all have data races. Nonetheless, when writing a program, there is no need to think explicitly about compiler or hardware memory reordering; we can still reason entirely in terms of sequential consistency, as long as we follow the rules and avoid data races.

This has some consequences that often surprise programmers. Consider the program in Figure 3, where *x* and *y* are initially false. When reasoning about whether this has a data race, we observe that there is no sequentially consistent execution (that is, no interleaving of thread steps) in which either assignment is executed. Thus, there are no pairs of conflicting operations, and hence certainly no data races.

Work at a Higher Level

So far, our programming model still has us thinking of interleaving thread execution at the memory-access or instruction level. Data races are defined

in terms of accesses to memory locations, and sequential consistency is defined in terms of interleaving indivisible steps, which are effectively machine instructions. This is an entirely new complication. A programmer writing sequential code does not need to know about the granularity of machine instructions and whether memory is accessed a byte or a word at a time.

Fortunately, once we insist on data-race-free programs, this issue disappears. A very useful side effect of our model is that a thread's synchronization-free regions appear indivisible or atomic. Thus, although our model is defined in terms of memory locations and individual steps, there is really no way to tell what those steps and memory locations are without introducing data races.

More generally, data-race-free programs always behave as though they were interleaved only at synchronization operations, such as mutex lock/unlock operations. If this were not the case, synchronization-free code sections from different threads would appear to interleave as in figures 4 and 5.

In the first case (Figure 4), no such interleaved code sections contain conflicting operations, and each section effectively operates on its own separate set of memory locations. The instruction interleaving is entirely equivalent to one in which these code sections execute one after the other as shown in the figure, with the only visible interleaving at synchronization operations (not shown).

In the second case (Figure 5), two code sections contain conflicting operations on the same memory location. In this case there is an alternate interleaving in which the conflicting operations appear next to each other, and a data race is effectively exhibited, as shown. Thus, this cannot happen for data-race-free programs.

This means that, for a data-race-free program, any section of code containing no synchronization operations behaves as though it executes atomically (that is, all at once) without being affected by other threads and without another thread being able to see any variable values occurring in the middle of that code section. Thus, insisting on data-race-free programs

Figure 3. Is there a data race if initially *x* = *y* = false?

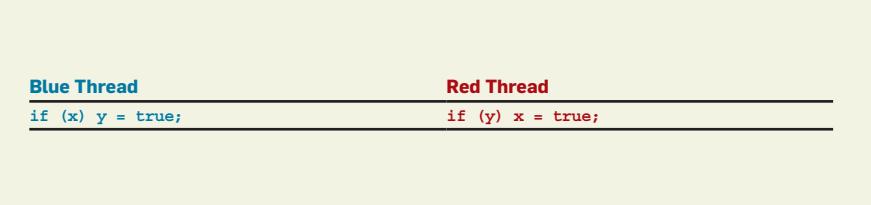
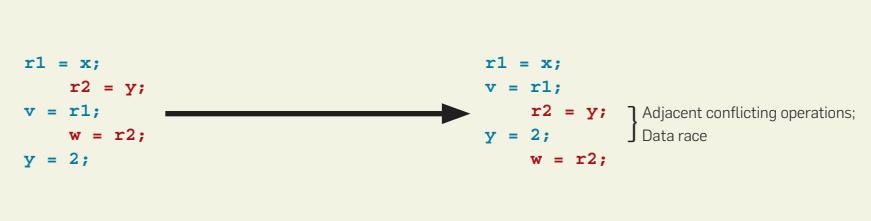


Figure 4. Conflict-free interleaving is not observable.



Figure 5. Interleaving with conflict implies a data race.



has some pleasant consequences:

- We no longer care whether memory is updated a byte or a word at a time. Properly written code can't tell any more than it could for sequential code.

- Library calls that do not use internal synchronization behave as if they execute in a single step. The intermediate states cannot be seen by another thread. Thus, such libraries can continue to specify only the overall effect of making a call, not which intermediate values might be taken by variables. Of course, that is what we have been doing all along, but it really makes sense only with data-race freedom.

- Reasoning about multithreaded programs is still hard, but without data races, it's not as hard as people often claim. In particular, we don't have to care about all possible ways of interleaving threads' instructions. At most, we care about the interleavings of synchronization-free regions.

Of course, all of these properties require that the program be data-race-free. Today, detecting and avoiding data-race bugs can be far from easy. Later we discuss recent progress toward making it easier.

In particular, to ensure data-race-freedom, it suffices to ensure that synchronization-free code sections that run at the same time neither write, nor read and write, the same variables. Thus, we can prune a significant number of instruction-level interleavings that need to be explored for this purpose.

Libraries can be (and generally are) designed to cleanly partition the responsibility for avoiding data races between library and client code. In the client code, we reason about data races at the level of logical objects, not memory locations. When deciding whether it is safe to call two library routines simultaneously, we need to make sure only that they don't both access the same *object*, or if they do, that neither access modifies the *object*. It is the library's responsibility to make sure that accesses to logically distinct objects do not introduce a data race as a result of unprotected accesses to some internal hidden memory locations. Similarly, it is the library's responsibility to make sure that reading an object doesn't introduce an inter-

When writing a program, there is no need to think explicitly about compiler or hardware memory reordering; we can still reason entirely in terms of sequential consistency, as long as we follow the rules and avoid data races.

nal write to the object that can create a data race.

With the data-race-free approach, library-implemented container data types can behave as built-in integers or pointers; the programmer does not need to be concerned with what goes on inside. As long as two different threads don't access the same *container* at the same time, or they are both read accesses, the implementation remains hidden.

Again, while all of these properties simplify reasoning about parallel code, they assume that the library writer and the client are responsible for obeying the prescribed disciplines.

But What if Locks are Too Slow?

The most common way to avoid data races is to use mutexes to ensure mutual exclusion between code sections accessing the same variable. In certain contexts, other synchronization mechanisms such as OpenMP's barriers are more appropriate. Experience has shown, however, that such mechanisms are insufficient in a few cases. Mutexes don't work well with signal or interrupt handlers, and they often involve significant overhead, even if they have started to get faster on recent processors.

Unfortunately, many environments, such as Posix threads, have not provided any real alternatives—so people cheat. Pthreads code commonly contains data races, which are typically claimed to be "benign." Some of these are outright bugs, in that the code, as currently compiled, will fail with small probability. The rest often risk getting "miscompiled" by compilers that either outright assume there are no data races⁴ and are hence misled by bad assumptions or that just produce some of the surprising effects previously discussed.

To escape this dilemma, most modern programming languages provide a way to declare *synchronization* variables. These behave as ordinary variables, but since accesses to them are considered to be *synchronization* operations, not *data* operations, synchronization variables can be safely accessed from multiple threads without creating a data race. In Java, a volatile int is an integer that can be accessed concurrently from multi-

ple threads. In C++11, you would write `atomic<int>` instead (`volatile` means something subtly different in C or C++).

Compilers treat synchronization variables specially, so our basic programming model is preserved. If there are no data races, threads still behave as though they execute in an interleaved fashion. Accessing a synchronization variable is a synchronization operation, however; code sequences extending across such accesses no longer appear indivisible.

Synchronization variables are sometimes the right tool for very simple shared data, such as the done flag in Figure 2. The only data race here is on the done flag, so simply declaring that as a synchronization variable fixes the problem.

Remember, however, that synchronization variables are difficult to use for complex data structures, since there is no easy way to make multiple updates to a data structure in one atomic operation. Synchronization variables are not replacements for mutexes.

In cases such as that shown in Figure 2, synchronization variables often avoid most of the locking overhead. Since they are still too expensive, both C++11 and Java provide some explicit experts-only mechanisms that allow you to relax the interleaving-based model, as mentioned before. Unlike programming with data races, it is possible to write correct code that uses these mechanisms, but our experience is that few people actually get this right. Our hope is that future hardware will reduce the need for it—and hardware is already getting better at this.

Real Languages

Most real languages fit our basic model. C++11 and C11 provide exactly this model. Data races have “undefined behavior;” they are errors in the same sense as an out-of-bounds array access. This is often referred to as *catch-fire* semantics for data races (though we do not know of any cases in which machines have actually caught fire as the result of a data race).

Although catch-fire semantics are sometimes still controversial, they are hardly new. The Ada 83 and 1995 Posix

thread specifications are less precise, but took basically the same position.

C++11 and C11 provide synchronization variables as `atomic<t>` and `_Atomic(t)`, respectively. In addition to reading and writing these variables, they support some simple indivisible compound operations; for example, incrementing a synchronization (`atomic`) variable with the “`++`” operator is an indivisible operation.

The situation for managed languages is more complex, mostly because of the security requirements they add to support untrusted code. Java fully supports our programming model, but it also, with only limited success, attempts to provide some guarantees for programs with data races. Although data races are not officially errors, it is now clear that we cannot precisely define what programs with data races actually mean.⁸ Data races remain evil.

Toward a Future Without Evil?

We have discussed how the absence of data races leads to a simple programming model supported by common languages. There simply does not appear to be any other reasonable alternative.¹ Unfortunately, one sticky problem remains: *guaranteeing* data-race-freedom is still difficult. Large programs almost always contain bugs, and often those bugs are data races. Today’s popular languages do not provide any usable semantics to such programs, making debugging difficult.

Looking forward, it is imperative that we develop automated techniques that detect or eliminate data races. Indeed, there is significant recent progress on several fronts: dynamic precise detection of data races;^{5,6} hardware support to raise an exception on a data race;⁷ and language-based annotations to eliminate data races from programs by design.³ These techniques guarantee that the considered execution or program has *no* data race (allowing the use of the simple model), but they still require more research to be commercially viable. Commercial products that detect data races have begun to appear (for example, Intel Inspector), and although they do not guarantee data-race-freedom, they are a big step in the right direction. We are optimistic

that one way or another, we will (we must!) conquer evil (data races) in the near future. C

Related articles on queue.acm.org

Trials and Tribulations of Debugging Concurrency

Kang Su Gatlin

<http://queue.acm.org/detail.cfm?id=1035623>

Scalable Parallel Programming with CUDA

John Nickolls, Ian Buck,

Michael Garland and Kevin Skadron

<http://queue.acm.org/detail.cfm?id=1365500>

Building Systems to Be Shared, Securely

Poul-Henning Kamp and Robert Watson

<http://queue.acm.org/detail.cfm?id=1017001>

References

For a more complete set of background references, please see reference 1.

1. Adve, S.V. and Boehm, H.-J. Memory models: A case for rethinking parallel languages and hardware. *Commun. ACM* 53, 8 (Aug. 2010), 90–101.
2. Adve, S.V. and Gharachorloo, K. Shared memory consistency models: A tutorial. *IEEE Computer* 29, 12 (1996), 66–76.
3. Bochino, R., et al. A type and effect system for deterministic parallel Java. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2009.
4. Boehm, H.-J. How to miscompile programs with “benign” data races. *Hot Topics in Parallelism (HotPar)*, 2011.
5. Elmas, T., Qadeer, S. and Tasiran, S. Goldilocks: A race-aware Java runtime. *Commun. ACM* 53, 11 (Nov. 2010), 85–92.
6. Flanagan, C. and Freund, S. FastTrack: Efficient and precise dynamic race detection. *Commun. ACM* 53, 11 (Nov. 2010), 93–101.
7. Lucia, B., Ceze, L., Strauss, K., Qadeer, S. and Boehm, H.-J. Conflict exceptions: Providing simple concurrent language semantics with precise hardware exceptions. In *Proceedings of the 2010 International Symposium on Computer Architecture*.
8. Sevcik, J. and Aspinall, D. On validity of program transformations in the Java memory model. In *European Conference on Object-oriented Programming*, 2008, 27–51.

Hans-J. Boehm is a research manager at Hewlett Packard Labs. He is probably best known as the primary author of a commonly used garbage collection library. Experiences with threads in that project eventually led him to initiate the effort to properly define threads and shared variables in C++11.

Sarita V. Adve is a professor in the department of computer science at the University of Illinois at Urbana-Champaign. Her research interests are in computer architecture and systems, parallel computing, and power and reliability-aware systems. She co-developed the memory models for the C++ and Java programming languages, based on her early work on data-race-free models.

The SGI Origin: A ccNUMA Highly Scalable Server

James Laudon and Daniel Lenoski

Silicon Graphics, Inc.

2011 North Shoreline Boulevard
Mountain View, California 94043
laudon@sgi.com lenoski@sgi.com

Abstract

The SGI Origin 2000 is a cache-coherent non-uniform memory access (ccNUMA) multiprocessor designed and manufactured by Silicon Graphics, Inc. The Origin system was designed from the ground up as a multiprocessor capable of scaling to both small and large processor counts without any bandwidth, latency, or cost cliffs. The Origin system consists of up to 512 nodes interconnected by a scalable Craylink network. Each node consists of one or two R10000 processors, up to 4 GB of coherent memory, and a connection to a portion of the XIO IO subsystem. This paper discusses the motivation for building the Origin 2000 and then describes its architecture and implementation. In addition, performance results are presented for the NAS Parallel Benchmarks V2.2 and the SPLASH2 applications. Finally, the Origin system is compared to other contemporary commercial ccNUMA systems.

1 Background

Silicon Graphics has offered multiple generations of symmetric multiprocessor (SMP) systems based on the MIPS microprocessors. From the 8 processor R3000-based Power Series to the 36 processor R4000-based Challenge and R10000-based Power Challenge systems, the cache-coherent, globally addressable memory architecture of these SMP systems has provided a convenient programming environment for large parallel applications while at the same time providing for efficient execution of both parallel and throughput based workloads.

The follow-on system to the Power Challenge needed to meet three important goals. First, it needed to scale beyond the 36 processor limit of the Power Challenge and provide an infrastructure that supports higher performance per processor. Given the factor of four processor count increase between the Power Series and Power Challenge lines, it was desired to have the next system support at least another factor of four in maximum processor count. Second, the new system had to retain the cache-coherent globally addressable memory model of the Power Challenge. This model is critical for achieving high performance on loop-level parallelized code and for supporting the existing Power Challenge users. Finally, the entry level and incremental cost of the system was desired to be lower than that of a high-performance SMP, with the cost ideally approaching that of a cluster of workstations.

Simply building a larger and faster snoopy bus-based SMP system could not meet all three of these goals. The second goal might be achievable, but it would surely compromise performance for larger processor counts and costs for smaller configurations.

Therefore a very different architecture was chosen for use in the next generation Origin system. The Origin employs distributed

Permission to make digital/hard copy of part or all this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
ISCA '97 Denver, CO, USA

© 1997 ACM 0-89791-901-7/97/0006...\$3.50

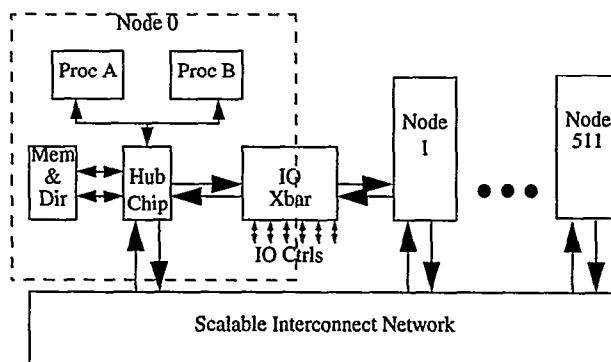


Figure 1 Origin block diagram

shared memory (DSM), with cache coherence maintained via a directory-based protocol. A DSM system has the potential for meeting all three goals: scalability, ease of programming, and cost. The directory-based coherence removes the broadcast bottleneck that prevents scalability of the snoopy bus-based coherence. The globally addressable memory model is retained, although memory access times are no longer uniform. However, as will be shown in this paper, Origin was designed to minimize the latency difference between remote and local memory and to include hardware and software support to insure that most memory references are local. Finally, a low initial and incremental cost can be provided if the natural modularity of a DSM system is exploited at a relatively fine granularity by the product design.

In the following section of this paper, the scalable shared-memory multiprocessing (S^2MP) architecture of the Origin is presented. Section 3 details the implementation of the Origin 2000. Performance of the Origin 2000 is presented in Section 4. Section 5 compares the Origin system with other contemporary ccNUMA systems. Finally, Section 6 concludes the paper.

2 The Origin S^2MP Architecture

A block diagram of the Origin architecture is shown in Figure 1. The basic building block of the Origin system is the dual-processor node. In addition to the processors, a node contains up to 4 GB of main memory and its corresponding directory memory, and has a connection to a portion of the IO subsystem.

The architecture supports up to 512 nodes, for a maximum configuration of 1024 processors and 1 TB of main memory. The nodes can be connected together via any scalable interconnection network. The cache coherence protocol employed by the Origin system does not require in-order delivery of point-to-point messages to allow the maximum flexibility in implementing the interconnect network.

The DSM architecture provides global addressability of all memory, and in addition, the IO subsystem is also globally addressable. Physical IO operations (PIOs) can be directed from any processor

to any IO device. IO devices can DMA to and from all memory in the system, not just their local memory.

While the two processors share the same bus connected to the Hub, they do not function as a snoopy cluster. Instead they operate as two separate processors multiplexed over the single physical bus (done to save Hub pins). This is different from many other ccNUMA systems, where the node is a SMP cluster. Origin does not employ a SMP cluster in order to reduce both the local and remote memory latency, and to increase remote memory bandwidth. Local memory latency is reduced because the bus can be run at a much higher frequency when it needs to support only one or two processor than when it must support large numbers of processors. Remote memory latency is also reduced by a higher frequency bus, and in addition because a request made in a snoopy bus cluster must generally wait for the result of the snoop before being forwarded to the remote node[7]. Remote bandwidth can be lower in a system with a SMP cluster node if memory data is sent across the remote data bus before being sent to the network, as is commonly done in DSM systems with SMP-based nodes[7][8]. For remote requests, the data will traverse the data bus at both the remote node and at the local node of the requestor, leading to the remote bandwidth being one-half the local bandwidth. One of the major goals for the Origin system was to keep both absolute memory latency and the ratio of remote to local latency as low as possible and to provide remote memory bandwidth equal to local memory bandwidth in order to provide an easy migration path for existing SMP software. As we will show in the paper, the Origin system does accomplish both goals, whereas in Section 6 we see that all the snoopy-bus clustered ccNUMA systems do not achieve all of these goals.

In addition to keeping the ratio of remote memory to local memory latency low, Origin also includes architectural features to address the NUMA aspects of the machine. First, a combination of hardware and software features are provided for effective page migration and replication. Page migration and replication is important as it reduces effective memory latency by satisfying a greater percentage of accesses locally. To support page migration Origin provides per-page hardware memory reference counters, contains a block copy engine that is able to copy data at near peak memory speeds, and has mechanisms for reducing the cost of TLB updates.

Other performance features of the architecture include a high-performance local and global interconnect design, coherence protocol features to minimize latency and bandwidth per access, and a rich set of synchronization primitives. The intra-node interconnect consists of single Hub chip that implements a full four-way crossbar between processors, local memory, and the I/O and network interfaces. The global interconnect is based on a six-ported router chip configured in a multi-level fat-hypercube topology.

The coherence protocol supports a clean-exclusive state to minimize latency on read-modify-write operations. Further, it allows cache dropping of clean-exclusive or shared data without notifying the directory in order to minimize the impact on memory/directory bandwidth caused by directory coherence. The architecture also supports request forwarding to reduce the latency of interprocessor communication.

For effective synchronization in large systems, the Origin system provides fetch-and-op primitives on memory in addition to the standard MIPS load-linked/store-conditional (LL/SC) instructions. These operations greatly reduce the serialization for highly contended locks and barrier operations.

Origin includes many features to enhance reliability and availability. All external cache SRAM and main memory and directory DRAM are protected by a SECDED ECC code. Furthermore, all high-speed router and I/O links are protected by a full CRC code and a hardware link-level protocol that detects and automatically retries faulty packets. Origin's modular design provides the overall

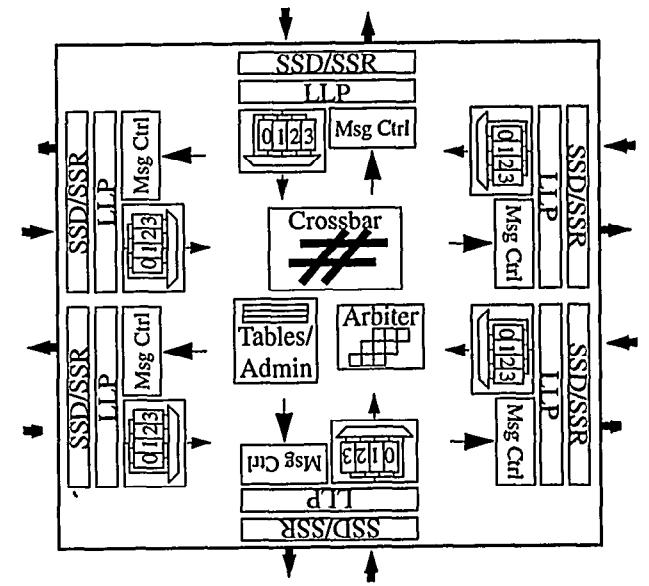


Figure 2 SPIDER ASIC block diagram

basis for a highly available hardware architecture. The flexible routing network supports multiple paths between nodes, partial population of the interconnect, and the hot plugging of cabled-links that permits the bypass, service, and reintegration of faulty hardware.

To address software availability in large systems, Origin provides access protection rights on both memory and IO devices. These access protection rights prevent unauthorized nodes from being able to modify memory or IO and allows an operating system to be structured into cells or partitions with containment of most failures to within a given partition[10][12].

3 The Origin Implementation

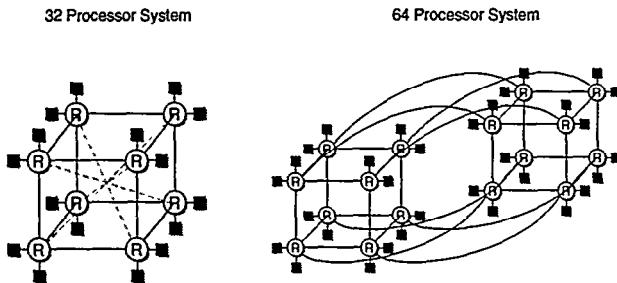
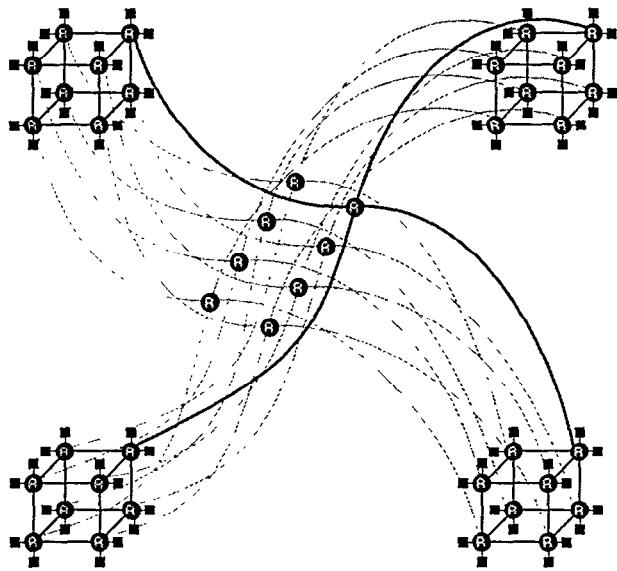
While existence proofs for the DSM architecture have been available in the academic community for some time[1][6], the key to commercial success of this architecture will be an aggressive implementation that provides for a truly scalable system with low memory latency and no unexpected bandwidth bottlenecks. In this section we explore how the Origin 2000 implementation meets this goal. We start by exploring the global interconnect of the system. We then present an overview of the cache coherence protocol, followed with a discussion of the node design. The IO subsystem is explored next, and then the various subsystems are tied together with the presentation of the product design. Finally, this section ends with a discussion of interesting performance features of the Origin system.

3.1 Network Topology

The interconnect employed in the Origin 2000 system is based on the SGI SPIDER router chip[4]. A block diagram of this chip is shown in Figure 2. The main features of the SPIDER chip are:

- six pairs of unidirectional links per router
- low latency (41 ns pin-to-pin) wormhole routing
- DAMQ buffer structures[4] with global arbitration to maximize utilization under load.
- four virtual channels per physical channel
- congestion control allowing messages to adaptively switch between two virtual channels

32 Processor System

**Figure 3 32P and 64P Bristled Hypercubes****Figure 4 128P Hierarchical Fat Bristled Hypercube**

- support for 256 levels of message priority with increased priority via packet aging
- CRC checking on each packet with retransmission on error via a go-back-n sliding window protocol
- software programmable routing tables

The Origin 2000 employs SPIDER routers to create a bristled fat hypercube interconnect topology. The network topology is bristled in that two nodes are connected to a single router instead of one. The fat hypercube comes into play for systems beyond 32 nodes (64 processors). For up to 32 nodes, the routers connect in a bristled hypercube as shown in Figure 3. The SPIDER routers are labeled using R, the nodes are the block boxes connecting to the routers. In the 32 processor configuration, the otherwise unused SPIDER ports are shown as dotted lines being used for Express Links which connect the corners of the cube, thereby reducing latency and increasing bisection bandwidth.

Beyond 64 processors, a hierarchical fat hypercube is employed. Figure 4 shows the topology of a 128 processor Origin system. The vertices of four 32-processor hypercubes are connected to eight meta-routers. To scale up to 1024 processors, each of the single meta-routers in the 128 processor system is replaced with a 5-D hypercubes.

3.2 Cache Coherence Protocol

The cache coherence protocol employed in Origin is similar to the Stanford DASH protocol[6], but has several significant perfor-

mance improvements. Like the DASH protocol, the Origin cache coherence protocol is non-blocking. Memory can satisfy any incoming request immediately; it never buffers requests while waiting for another message to arrive. The Origin protocol also employs the request forwarding of the DASH protocol for three party transactions. Request forwarding reduces the latency of requests which target a cache line that is owned by another processor.

The Origin coherence protocol has several enhancements over the DASH protocol. First, the Clean-exclusive (CEX) processor cache state (also known as the exclusive state in MESI) is fully supported by the Origin protocol. This state allows for efficient execution of read-modify-write accesses since there is only a single fetch of the cache line from memory. The protocol also permits the processor to replace a CEX cache line without notifying the directory. The Origin protocol is able to detect a rerequest by a processor that had replaced a CEX cache line and immediately satisfy that request from memory. Support of CEX state in this manner is very important for single process performance as much of the gains from the CEX state would be lost if directory bandwidth was needed each time a processor replaced a CEX line. By adding protocol complexity to allow for the "silent" CEX replacement, all of the advantages of the CEX state are realized.

The second enhancement of the Origin protocol over DASH is full support of upgrade requests which move a line from a shared to exclusive state without the bandwidth and latency overhead of transferring the memory data.

For handling incoming I/O DMA data, Origin employs a write-invalidate transaction that uses only a single memory write as opposed to the processor's normal write-allocate plus writeback. This transaction is fully cache coherent (i.e., any cache invalidations/interventions required by the directory are sent), and increases I/O DMA bandwidth by as much as a factor of two.

Origin's protocol is fully insensitive to network ordering. Messages are allowed to bypass each other in the network and the protocol detects and resolves all of these out-of-order message deliveries. This allows Origin to employ adaptive routing in its network to deal with network congestion.

The Origin protocol uses a more sophisticated network deadlock avoidance scheme than DASH. As in DASH, two separate networks are provided for requests and replies (implemented in Origin via different virtual channels). The Origin protocol does have requests which generate additional requests (these additional requests are referred to as *interventions* or *invalidations*). This request-to-request dependency could lead to deadlock in the request network. In DASH, this deadlock was broken by detecting a potential deadlock situation and sending negative-acknowledgments (NAKs) to all requests which needed to generate additional requests to be serviced until the potential deadlock situation was resolved. In Origin, rather than sending NAKs in such a situation, a *backoff* intervention or invalidate is sent to the requestor on the reply network. The backoff message contains either the target of the intervention or the list of sharers to invalidate, and is used to signal the requestor that the memory was unable to generate the intervention or invalidate directly and therefore the requestor must generate that message instead. The requestor can always sink the backoff reply, which causes the requestor to then queue up the intervention or invalidate for injection into the request network as soon as the request network allows. The backoff intervention or invalidate changes the request-intervention-reply chain to two request-reply chains (one chain being the request-backoff message, one being the intervention-reply chain), with the two networks preventing deadlock on these two request-reply chains. The ability to generate backoff interventions and invalidations allows for better forward progress in the face of very heavily loaded systems since the deadlock detection in both DASH and Origin is conservatively

done based on local information, and a processor that receives a backoff is guaranteed that it will eventually receive the data, while a processor that receives a NAK must retry its request.

Since the Origin system is able to maintain coherence over 1024 processors, it obviously employs a more scalable directory scheme than in DASH. For tracking sharers, Origin supports a bit-vector directory format with either 16 or 64 bits. Each bit represents a node, so with a single bit to node correspondence the directory can track up to a maximum of 128 processors. For systems with greater than 64 nodes, Origin dynamically selects between a full bit vector and coarse bit vector[12] depending on where the sharers are located. This dynamic selection is based on the machine being divided into up to eight 64 node *octants*. If all the processors sharing the cache line are from the same octant, the full bit vector is used (in conjunction with a 3-bit octant identifier). If the processors sharing the cache line are from different octants, a coarse bit vector where each bit represents eight nodes is employed.

Finally, the coherence protocol includes an important feature for effective page migration known as *directory poisoning*. The use of directory poisoning will be discussed in more detail in Section 3.6. A slightly simplified flow of the cache coherence protocol is now presented for both read, read-exclusive, and writeback requests. We start with the basic flow for a read request.

1. Processor issues read request.
2. Read request goes across network to home memory (requests to local memory only traverse Hub).
3. Home memory does memory read and directory lookup.
4. If directory state is Unowned or Exclusive with requestor as owner, transitions to Exclusive and returns an exclusive reply to the requestor. *Go to 5a*.
If directory state is Shared, the requesting node is marked in the bit vector and a shared reply is returned to the requestor. *Go to 5a*.
- If directory state is Exclusive with another owner, transitions to Busy-shared with requestor as owner and send out an intervention shared request to the previous owner and a speculative reply to the requestor. *Go to 5b*.
- If directory state is Busy, a negative acknowledgment is sent to the requestor, who must retry the request. QED
- 5a. Processor receives exclusive or shared reply and fills cache in CEX or shared (SHD) state respectively. QED
- 5b. Intervention shared received by owner. If owner has a dirty copy it sends a shared response to the requestor and a sharing writeback to the directory. If owner has a clean-exclusive or invalid copy it sends a shared ack (no data) to the requestor and a sharing transfer (no data) to the directory.
- 6a. Directory receives shared writeback or shared transfer, updates memory (only if shared writeback) and transitions to the shared state.
- 6b. Processor receives both speculative reply and shared response or ack. Cache filled in SHD state with data from response (if shared response) or data from speculative reply (if shared ack). QED

The following list details the basic flow for a read-exclusive request.

1. Processor issues read-exclusive request.
2. Read-exclusive request goes across network to home memory (only traverses Hub if local).
3. Home memory does memory read and directory lookup.
4. If directory state is Unowned or Exclusive with requestor as owner, transitions to Exclusive and returns an exclusive reply to the requestor. *Go to 5a*.

If directory state is Shared, transitions to Exclusive and a exclusive reply with invalidates pending is returned to the re-

questor. Invalidations are sent to the sharers. *Go to 5b*.

If directory state is Exclusive with another owner, transitions to Busy-Exclusive with requestor as owner and sends out an intervention exclusive request to the previous owner and a speculative reply to the requestor. *Go to 5c*.

If directory state is Busy, a negative acknowledgment is sent to the requestor, who must retry the request. QED

- 5a. Processor receives exclusive reply and fills cache in dirty exclusive (DEX) state. QED
- 5b. Invalidates received by sharers. Caches invalidated and invalidate acknowledgments sent to requestor. *Go to 6a*.
- 5c. Intervention shared received by owner. If owner has a dirty copy it sends an exclusive response to the requestor and a dirty transfer (no data) to the directory. If owner has a clean-exclusive or invalid copy it sends an exclusive ack to the requestor and a dirty transfer to the directory. *Go to 6b*.
- 6a. Processor receives exclusive reply with invalidates pending and all invalidate acks. (Exclusive reply with invalidates pending has count of invalidate acks to expect.) Processor fills cache in DEX state. QED
- 6b. Directory receives dirty transfer and transitions to the exclusive state with new owner.
- 6c. Processor receives both speculative reply and exclusive response or ack. Cache filled in DEX state with data from response (if exclusive response) or data from speculative reply (if exclusive ack). QED

The flow for an upgrade (write hit to SHD state) is similar to the read-exclusive, except it only succeeds for the case where the directory is in the shared state (and the equivalent reply to the exclusive reply with invalidates pending does not need to send the memory data). In all other cases a negative acknowledgment is sent to the requestor in response to the upgrade request.

Finally, the flow for a writeback request is presented. Note that if a writeback encounters the directory in one of the busy states, this means that the writeback was issued before an intervention targeting the cache line being written back made it to the writeback issuer. This race is resolved in the Origin protocol by “bouncing” the writeback data off the memory as a response to the processor that caused the intervention, and sending a special type of writeback acknowledgment that informs the writeback issuer to wait for (and then ignore) the intervention in addition to the writeback acknowledgment.

1. Processor issues writeback request.
2. Writeback request goes across network to home memory (only traverses Hub if local).
3. Home memory does memory write and directory lookup.
4. If directory state is Exclusive with requestor as owner, transitions to Unowned and returns a writeback exclusive acknowledge to the requestor. *Go to 5a*.
If directory state is Busy-shared, transitions to Shared, a shared response is returned to the owner marked in the directory. A writeback busy acknowledgment is also sent to the requestor. *Go to 5b*.
- If directory state is Busy-exclusive, transitions to Exclusive, an exclusive response is returned to the owner marked in the directory. A writeback busy acknowledgment is also sent to the requestor. *Go to 5b*.
- 5a. Processor receives writeback exclusive acknowledgment. QED
- 5b. Processor receives both a writeback busy acknowledgment and an intervention. QED

3.3 Node Design

The design of an Origin node fits on a single 16" x 11" printed circuit board. A drawing of the Origin node board is shown in Figure

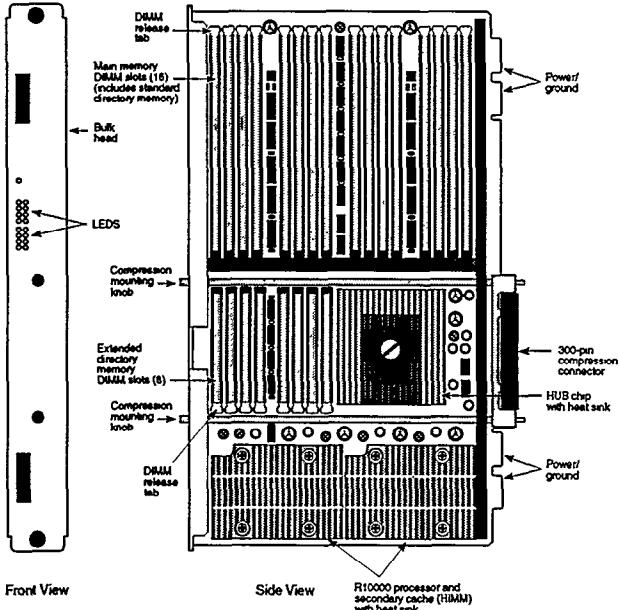


Figure 5 An Origin node board

5. At the bottom of the board are two R10000 processors with their secondary caches. The R10000 is a four-way out-of-order superscalar processor[14]. Current Origin systems run the processor at 195 MHz and contain 4 MB secondary caches. Each processor and its secondary cache is mounted on a horizontal in-line memory module (HIMM) daughter card. The HIMM is parallel to the main node card and connects via low-inductance fuzz-button processor and HIMM interposers. The system interface buses of the R10000s are connected to the Hub chip. The Hub chip also has connections to the memory and directory on the node board, and has two ports that exit the node board via the 300-pin CPOP (compression pad-on-pad) connector. These two ports are the Craylink connection to router network and the XIO connection to the IO subsystem.

As was mentioned in Section 3.2, a 16 bit-vector directory format and a 64 bit-vector format are supported by the Origin system. The directory that implements the 16-bit vector format is located on the same DIMMs as main memory. For systems larger than 32 processors, additional expansion directory is needed. These expansion directory slots, shown to the left of the Hub chip in Figure 5, operate by expanding the width of the standard directory included on the main memory boards. The Hub chip operates on standard 16-bit directory entries by converting them to expanded entries upon their entry into the Hub chip. All directory operations within the Hub chip are done on the expanded directory entries, and the results are then converted back to standard entries before being written back to the directory memory. Expanded directory entries obviously bypass the conversion stages.

Figure 6 shows a block diagram of the Hub chip. The hub chip is divided into five major sections: the crossbar (XB), the IO interface (II), the network interface (NI), the processor interface (PI), and the memory and directory interface (MD). All the interfaces communicate with each other via FIFOs that connect to the crossbar.

The IO interface contains the translation logic for interfacing to the XIO IO subsystem. The XIO subsystem is based on the same low-level signalling protocol as the Craylink network (and uses the same interface block to the XIO pins as in the SPIDER router of Figure 2), but utilizes a different higher level message protocol. The IO section also contains the logic for two block transfer engines (BTEs) which are able to do memory to memory copies at

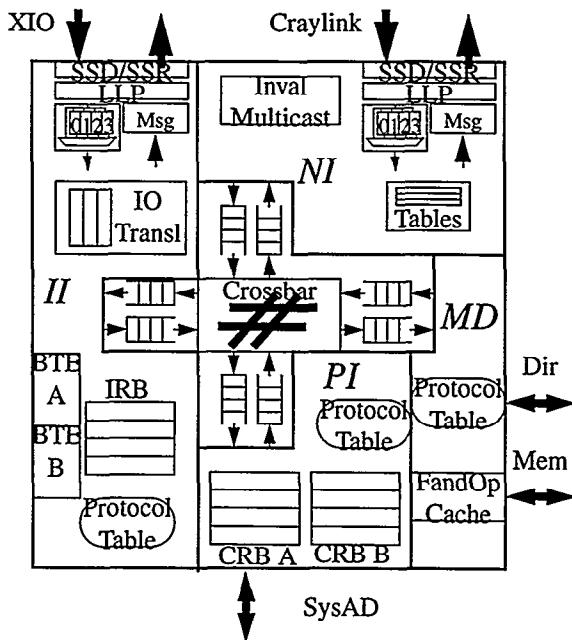


Figure 6 Hub ASIC block diagram

near the peak of a node's memory bandwidth. It also implements the IO request tracking portion of the cache coherence protocol via the IO request buffers (IRB) and the IO protocol table. The IRB tracks both full and partial cache line DMA requests by IO devices as well as full cache line requests by the BTEs.

The network interface takes messages from the II, PI, and MD and sends them out on the Craylink network. It also receives incoming messages for the MD, PI, II, and local Hub registers from the Craylink network. Routing tables for outgoing messages are provided in the NI as the software programmable routing of the SPIDER chip is pipelined by one network hop[4]. The NI also is responsible for taking a compact intra-Hub version of the invalidation message resulting from a coherence operation (a bit-vector representation) and generating the multiple unicast invalidate messages required by that message.

The processor interface contains the logic for implementing the request tracking for both processors. Read and write requests are tracked via a coherent request buffer (CRB), with one CRB per processor. The PI also includes the protocol table for its portion of the cache coherence protocol. The PI also has logic for controlling the flow of requests to and from the R10000 processors and contains the logic for generating interrupts to the processors.

Finally, the memory/directory section contains logic for sequencing the external memory and directory synchronous DRAMs (SDRAMs). Memory on a node is banked 4-32 way depending on how many memory DIMMs are populated. Requests to different banks and requests to the same page within a bank as the previous request can be serviced at minimum latency and full bandwidth. Directory operations are performed in parallel with the memory data access. A complete directory entry (and page reference counter, as will be discussed in Section 3.6) read-modify-write can be performed in the same amount of time it takes to fetch the 128B cache line from memory. The MD performs the directory portion of the cache coherence protocol via its protocol table and generates the appropriate requests and/or replies for all incoming messages. The MD also contains a small fetch-and-op cache which sits in front of the memory. This fetch-and-op cache allows fetch-and-op variables that hit in the cache to be updated at the minimum net-

Port	SysAD	Mem	XIO	Craylink
GB/s	0.78	0.78	1.56	1.56

Table 1 Hub ASIC port bandwidths

Section	XB	IO	NI	PI	MD
K gates	246	296	56	133	77

Table 2 Hub ASIC gate count

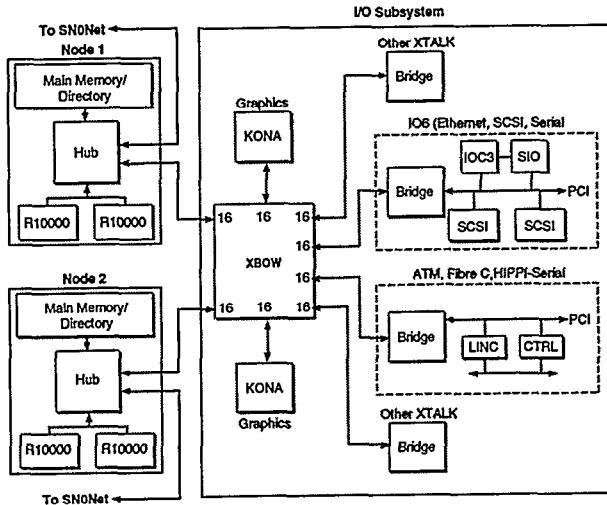


Figure 7 Example IO subsystem block diagram

work reply serialization rate of 41 ns instead of at the much slower SDRAM read-modify-write timing.

Note that all the protocol tables in the Hub are hard-wired. While programmable protocol engines can come close to achieving the performance of a hard-wired protocol state machine[5][9], we opted for hard-wiring the protocol to minimize latency and maximize bandwidth. We were also concerned about the variability in latency and bandwidth given the caching of directory information used by most programmable approaches. To ensure that the cache coherence protocol implemented in the tables was correct, we employed formal verification[3]. Formal verification worked extremely well; no bugs have been found in the Origin cache coherence protocol since the formal verification was completed.

The raw data bandwidth of the Hub chip ports is listed in Table 1. A summary of the sizes of the units is shown in Table 2. Note that most of the chip is allocated either to interfacing to the IO subsystem or in the crossbar itself, rather than in implementing global cache coherence.

3.4 IO Subsystem

Not too surprisingly, the Origin system also utilizes crossbars in its IO subsystem. Figure 7 shows one possible configuration of IO cards connected to two nodes. Using the same link technology as in the Craylink interconnect, each Hub link provides a peak of 1.56 GB/sec of bandwidth to the six XIO cards connected to it (actually limited to half this amount if only local memory bandwidth is considered). At the heart of the IO subsystem is the Crossbow (Xbow) ASIC, which has many similarities with the SPIDER router. The primary differences between the Xbow and the router is a simplifi-

Board	Number of Ports
Base IO	2 Ultra SCSI, 1 Fast Enet, 2 serial
Ultra SCSI	4
10/100 Enet	4
HiPPI	1 serial
Fibre Channel	2 Cu loops
ATM OC3	4
Infinite Reality Gfx	1
Standard PCI Cage	3
VME Adapter	1

Table 3 Origin IO boards

cation of the Xbow buffering and arbitration protocols given the chips more limited configuration. These simplifications reduce costs and permit eight ports to be integrated on a single chip. Some of the main features of the Xbow are:

- eight XIO ports, connected in Origin to 2 nodes and 6 XIO cards.
- two virtual channels per physical channel
- low latency wormhole routing
- support for allocated bandwidth of messages from particular devices
- CRC checking on each packet with retransmission on error via a go-back-n sliding window protocol

The Crossbow has support in its arbiter for allocating a portion of the bandwidth to a given IO device. This feature is important for certain system applications such as video on demand.

A large number of XIO cards are available to connect to the Crossbow. Table 3 contains a listing of the common XIO cards. The highest performance XIO cards connect directly to the XIO, but most of the cards bridge XIO to an embedded PCI bus with multiple external interfaces. The IO bandwidth together with integration provide IO performance which is effectively added as a PCI-bus at a time versus individual PCI cards.

3.5 Product Design

The Origin 2000 is a highly modular design. The basic building block is the deskside module, which has slots for 4 node boards, 2 router boards, and 12 XIO boards. The module also includes a CDROM and up to 5 Ultra SCSI devices. Figure 8 shows a block diagram of the deskside module, while Figure 9 shows a rear-view perspective of a deskside system. The system has a central midplane, which has two Crossbow chips mounted on it. The 4 node and 12 XIO boards plug into the midplane from the rear of the system, while the 2 router boards, the power supply and the UltraSCSI devices plug into the midplane from the front of the system. A module can be used as a stand-alone deskside system or two modules (without the deskside plastic skins) can be mounted in a rack to form a 16 processor system. In addition to the two modules, the rack also includes a disk bay for up to 8 additional disks. One of the modules can be replaced with an Infinite reality graph-

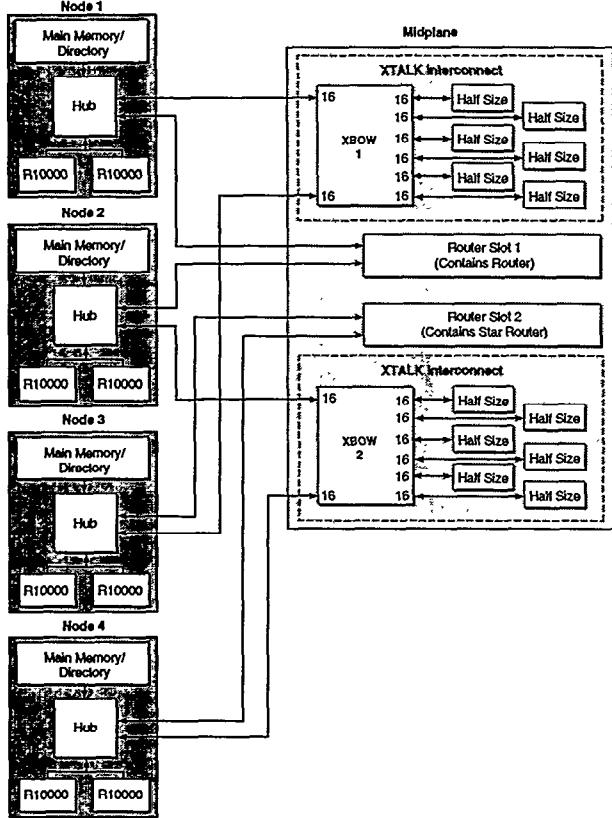


Figure 8 Deskside module block diagram

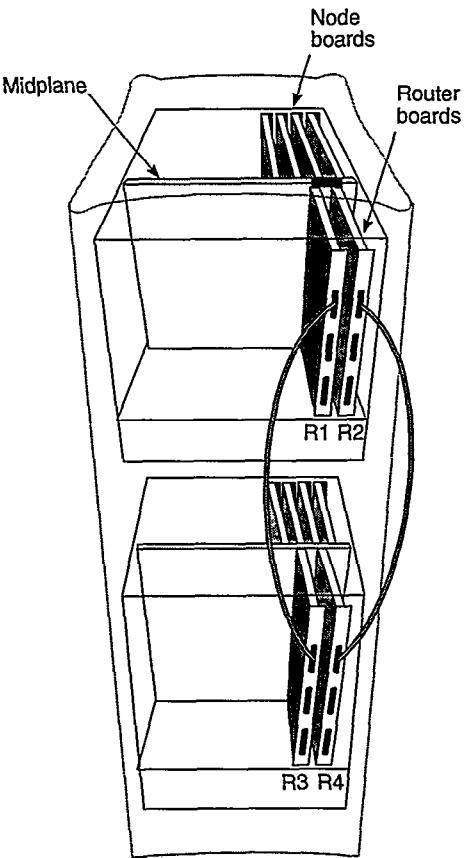


Figure 10 16 processor Origin system.

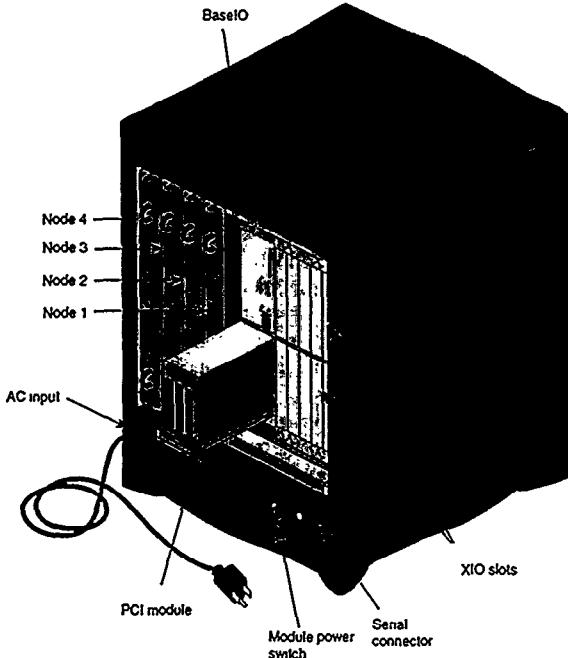


Figure 9 Deskside module, rear view

ics module or with 4 additional 8-disk bays. An Origin Vault which contains 9 8-disk bays in a single rack is also available. Figure 10 depicts a configured rack supporting 16 processors, 24 XIO boards, and 18 UltraSCSI devices.

3.6 Performance Features

The Origin system has two features very important for achieving good performance in a highly scalable system. First, fetch-and-op primitives are provided as uncached operations that occur at the memory. Fetch-and-op variables are used for highly contended locks, barriers, and other synchronization mechanisms. The typical serialization rate (the rate at which a stream of requests can be serviced) for fetch-and-op variables is 41 ns. In Section 4.1 we will show how fetch-and-op variables can improve the performance of highly contended objects.

Second, Origin provides hardware and software support for page migration. Page migration is important for NUMA systems as it changes many of the cache misses which would have gone to remote memory to local misses. To help the OS in determining when and which page to migrate the Origin system provides an array of per-page memory reference counters, which are stored in the directory memory. This array is indexed by the nodes in a system (up to 64 nodes, beyond this 8 nodes share a single counter). When a request comes in, its reference counter is read out during the directory lookup and incremented. In addition, the reference counter of the home node is read out during the same directory lookup. The requestor's count and home count are compared and if the difference exceeds a software programmable threshold register (and the migration control bits stored with the requestor's reference counter says that this page is a candidate for migration), an interrupt is generated to the home node. This interrupt signals a potential migration candidate to the operating system.

When the operating system determines it does indeed want to migrate the page[13], two operations need to be performed. First, the OS needs to copy the page from its current location to a free memory page on the requestor's node. Second, the OS needs to invali-

Memory level	Latency (ns)
L1 cache	5.1
L2 cache	56.4
local memory	310
4P remote memory	540
8P avg. remote memory	707
16P avg. remote memory	726
32P avg. remote memory	773
64P avg. remote memory	867
128P avg. remote memory	945

Table 4 Origin 2000 latencies

date all the translations to the old page cached in processor's TLBs and then update the translation for the migrated page to point to the new page.

The block transfer engine allows a 16 KB page to be copied from one node's memory to another in under 30 microseconds. Unfortunately, in a very large Origin system, the cost to invalidate all the TLBs and update the translation using a conventional TLB shootdown algorithm can be 100 microseconds or more, removing much of the benefit of providing a fast memory to memory copy. Recent page migration research has also identified TLB shootdown as a significant cost of page migration[11]. To solve the TLB update problem, the directory supports a block transfer copy mode known as directory poisoning, which works as follows.

During the read phase of the poisoning block copy, in addition to reading the data, the directory protocol makes sure the latest copy of the data is written back to memory, and the directory is placed in the POISON state. Any access by a processor to a poisoned directory entry during the copy will result in a synchronous bus error. The bus error handler is able to quickly determine that the bus error was due to page migration, and the handler invalidates the processor's TLB entry for the page, and then has the process wait for the new translation to be produced.

Once the poisoning block copy has completed, the new translation is updated and all processors that took the poison bus error will load their TLB with the new translation. The poisoned page is now placed on a poisoned list to "age". The operating system invalidates one sequential TLB entry per scheduler tick, so after a time equal to the number of per-processor TLB entries times the period between scheduler ticks, the page can be moved off the poisoned list and onto the free list. This directory poisoning allows the TLB shootdown and page copy to proceed in parallel, and as a result the cost to migrate a page is much lower than if a standard TLB shootdown were invoked. This low cost of migration enables the operating system to be fairly aggressive in determining when to migrate a page.

4 Origin Performance

This section examines the performance of the Origin system using both microbenchmarks to measure latencies and bandwidths, and by using the NAS Parallel Benchmarks V2.2 and the SPLASH2 suite to measure performance of a set of parallel applications.

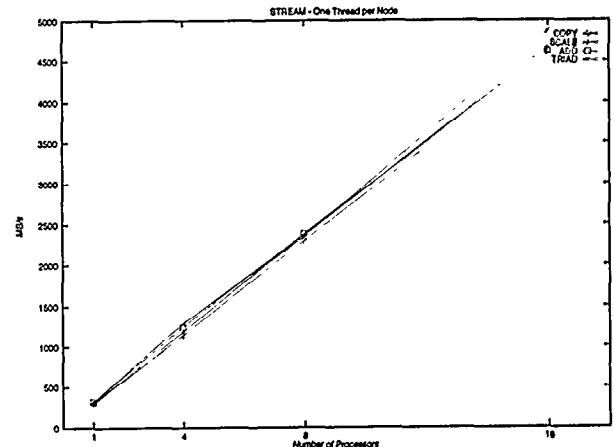


Figure 11 STREAM results - one thread per node

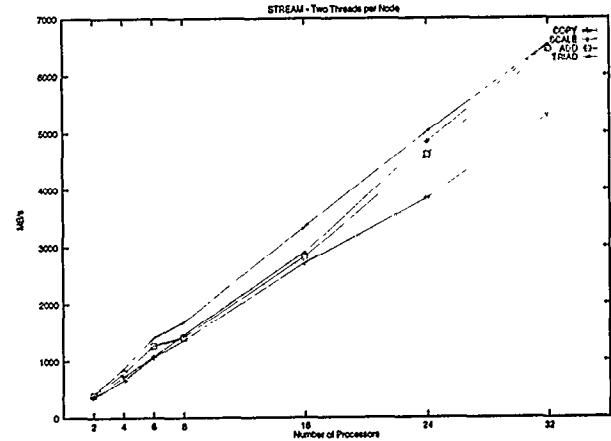


Figure 12 STREAM results - two threads per node

4.1 Microbenchmarks

The first microbenchmarks examine the latency and bandwidth of the Origin memory system. Table 4 shows the latency measured for a memory reference in isolation. This is the time from when the L1 cache is accessed until the instruction associated with the cache miss can graduate. The remote latency numbers for 16 and 32 processors assume that express links are employed.

The STREAM benchmark is the standard memory bandwidth benchmark for the high performance computing industry. STREAM measures the performance of four simple long vector kernels, which are sized to eliminate cache re-use, and reports the results in terms of sustained memory bandwidth. On parallel systems, STREAM is essentially completely parallel; the only communication required is for synchronization at the end of execution of each kernel.

On the Origin 2000, each processor can effectively utilize more than half of the memory bandwidth available on a node. Thus, we've included STREAM results in MB/s with only one thread running per node in Figure 11, and with two threads running per node in Figure 12.

Table 5 shows the effectiveness of the fetch-and-increment operations in implementing a globally shared counter. Note that LL/SC does much better for a single processor, at 6.9 million increments/second, since the counter variable and the lock surrounding it (which are allocated from the same cache line) stays loaded in the processor's cache, whereas the fetch-and-increment variable is always accessed via an uncached reference to local memory, and de-

M op/s	1 P	2 P	4 P	8 P	16 P	32 P
fch-inc	4.0	7.4	6.1	10.0	19.3	23.0
LL/SC	6.9	2.3	0.84	0.23	0.12	0.09

Table 5 Comparison of LL/SC and fetch-and-op for atomic increments

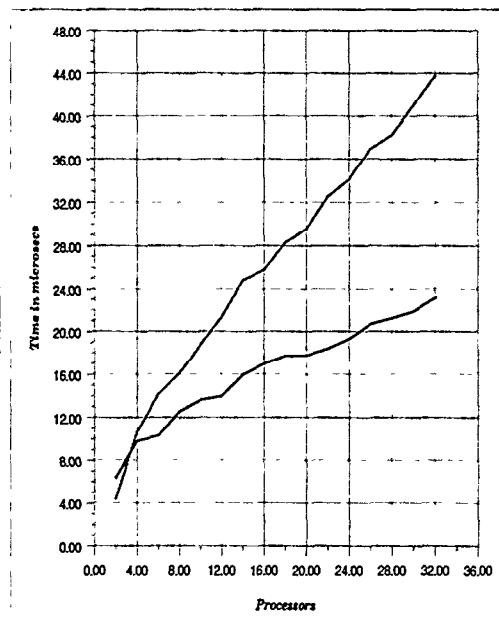


Figure 13 Comparison of LL/SC and fetch-and-op for a null doacross loop

livers only 4.0 million increments/sec. As more processors are added, however, the number of increments on the fetch-and-increment variable is able to increase to near the fetch-and-op cache throughput limit of 24.4 million/sec (one every 41 ns), while the number of increments on the LL/SC variable falls off dramatically, to under 100 thousand increments/second with 32 processors. Also note the drop off in fetch-and-op increments/second between two and four processors. With a small number of processors, the number of fetch-and-increments achievable per second is limited by the fetch-and-increment latency, since each R10000 processor can only have a single uncached read outstanding. Therefore with two processors, the fetch-and-increment can be allocated locally, whereas with four processors, only two of the processors can access the fetch-and-increment variable from local memory, and the other two processors must pay the longer remote latency.

Figure 13 shows the advantages of using fetch-and-op for barrier implementation. The graph shows the time to execute a null FORTRAN doacross statement. In addition to the barrier time, the null doacross includes the time to perform the work dispatch to the slaves and the execution of one iteration of an empty do-loop. Therefore the graph actually understates the performance benefits of fetch-and-ops in implementing the barrier itself.

4.2 Applications

In this section we examine the performance of the Origin system using the NAS Parallel Benchmarks V2.2 Class A and the

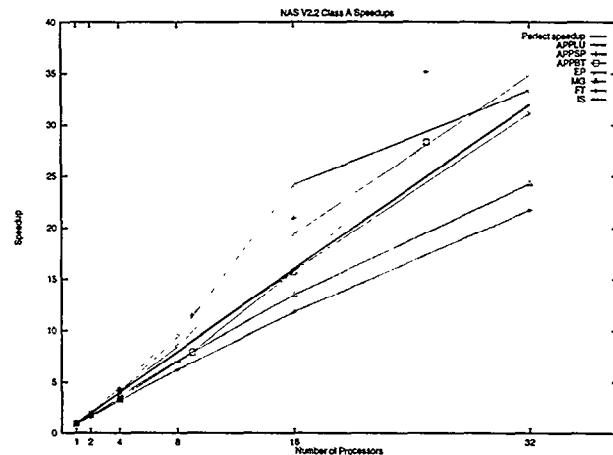


Figure 14 NAS Parallel V2.2 Speedups

Application	Command Line
radiosity	-batch -room
raytrace	balls4.env
lu	-n 2048 -b 16
ocean	-n 1026
barnes	< input.512

Table 6 SPLASH2 Applications

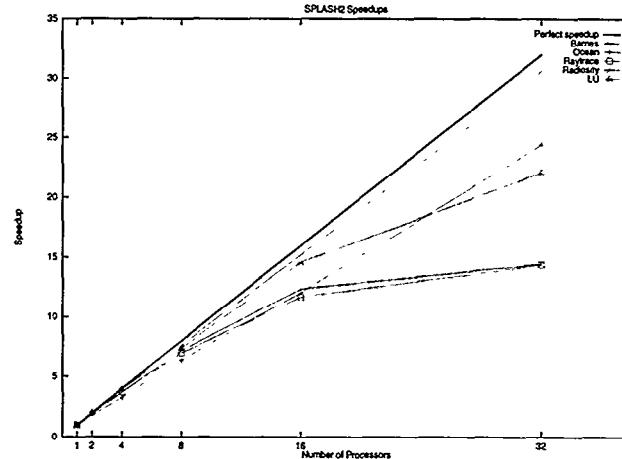


Figure 15 SPLASH2 Speedups

SPLASH2 suite. Application and operating system tuning is ongoing, so these results are a snapshot as of late February, 1997.

Figure 14 shows the performance of the NAS parallel benchmarks using the Class A datasets for up to 32 processors. Overall, speedup on the NAS benchmarks is very good. For several of the benchmarks, superlinear speedups are achieved due to the larger total cache size and memory bandwidth available as the number of processors (and therefore the number of nodes) increases.

Table 6 lists the applications run from the SPLASH2 suite along with their command line arguments. Figure 15 shows speedups for

the four SPLASH2 applications and one SPLASH2 kernel on the Origin system. Barnes and Ocean get good speedups to 32 processors, while Lu starts to roll off beyond 16 processors. Radiosity and Raytrace both begin to have speedup fall-off after 8 processors, and show very small increases between 16 and 32 processors. We have just started our investigation into the performance of the SPLASH suite, so we are not certain of the exact cause of the speedup roll-off for Lu, Radiosity, and Raytrace. Our benchmark machine had a limited amount of total memory, so the small data sets for these applications may be a contributor to the limited speedup at 32 processors.

5 Related Systems

The Origin system benefitted from the many lessons the authors learned in designing the Stanford DASH[7], so we start by discussing the major differences between Origin and DASH. We then contrast the Origin with three contemporary ccNUMA systems: the Sequent NUMAQ, the Data General NUMALiiNE, and the Convex Exemplar X.

5.1 Stanford DASH

The differences between the Origin and DASH coherence protocols was already explored in Section 3.2. The main architectural difference between the Origin and DASH systems is that DASH employed a four processor SMP cluster as its node[7], while Origin uses a two processor node where coherence between the processors in the node is handled by the directory-based protocol.

The major advantage of a SMP-based node is the potential for cache-to-cache sharing within the node. In [7] the 4-way intra-node sharing of DASH is shown to produce small performance gains for 3 out of 4 SPLASH applications with only the Barnes application showing significant performance gains.

On the other hand, the disadvantages of the SMP-based nodes are three-fold. First, to get to the number of processors where intra-node sharing will be significant the bus will most likely not be able to be on a single board. This causes local memory latency to be longer as the bus must run slower to support the large number of devices connected to it and more ASIC crossings will generally be between the processor and local memory. In addition, this makes the initial cost of the SMP node much higher since even a single processor node requires several boards. Second, remote latency also increases as requests to remote memory will generally have to wait for the results of the local processor snoops before being able to issue to the remote memory. Finally, as discussed earlier, the remote memory bandwidth for a SMP-based node is half the local memory bandwidth. In fact, in DASH, the remove memory bandwidth was reduced by a factor of three since each remote memory reference needed to traverse the local bus for the initial request, the home memory bus to get the data, and the local bus again to return the data to the processor.

While the DASH prototype did suffer increased latency on its local memory access time due to the snoopy bus, it did manage to have a 3:1 best case (nearest-neighbor) remote to local latency, which as we will see when we examine some commercial contemporary systems is quite good for a SMP-based ccNUMA machine. Despite being much better than other SMP-based node solutions, this ratio is still less than the 2:1 best case (nearest-neighbor) remote to local latency achieved by Origin.

5.2 Sequent NUMAQ and DG NUMALiiNE

The Sequent NUMAQ consists of up to 63 nodes, where each node is a 4-processor Pentium Pro-based SMP[8] referred to as a *quad*. The nodes are connected together via an SCI-based ring. Both the low-level transport layers and the higher-level coherence protocol

of the SCI are utilized in the NUMAQ. A programmable protocol engine is used to implement the SCI coherence protocol. A full board (the Lynx Board) implements the complete interface between the Pentium Pro based quad.

The Sequent NUMAQ is architecturally similar to the Stanford DASH, albeit with a different processor, a simpler network topology, and the SCI coherence protocol instead of the DASH coherence protocol. It has the same advantages and disadvantages of DASH: the local memory latency is good, around 250 ns, but the best case remote memory latency is around 8 times the local latency[8]. The choice of a ring with its low bisection bandwidth as the interconnect network causes large degradation in remote latency as the system interconnect becomes loaded.

The Data General NUMALiiNE is architecturally very similar to the Sequent NUMAQ. The node is a Pentium-Pro quad, and the nodes are connected via a SCI-based ring. As such it suffers from the same limitations in remote latency and network performance as the NUMAQ.

5.3 Convex Exemplar X

The Convex Exemplar X is similar to earlier Exemplar systems implementing a crossbar connected set of hyper-nodes that are then connected by parallel ring interconnects that implement a modified version of the SCI protocol[2]. In the X-class machines the hyper-node size has increased from 8 to 16 processors and the four 1-D rings have been replaced by eight sets of 2-D rings. Initial configurations support 64 processors (4 hypernodes), but the machine can architecturally scale to 512 processors in an 8x4 torus configuration.

The use of a crossbar intra-connect for the hypernode does reduce the bandwidth penalty of using an SMP node compared with the NUMAQ or NUMALiiNE machines, but still adds to latency in comparison with the smaller, more integrated nodes in Origin. While no latency data has been published to date, the ratio of local to remote in the Exemplar is likely to be similar to previous machines (5:1 without loading on small configurations), whereas the base numbers in the Origin start at 2:1 in small configurations and grow very slowly.

Another major difference is the use of a third-level cluster cache in the Exemplar. This mechanism is in contrast to Origin's page migration mechanism. The cluster cache can adapt to capacity misses more quickly than Origin's migration mechanism, but does this at the cost of increased latency for communication misses and misses that result from conflicts in the cluster cache. The cluster cache also hurts remote bandwidth because it implies that at least three DRAM accesses be made per remote miss (one in the local cluster cache, one at the home, an additional one or two if the line is held dirty in another cluster cache, and a final access to allocate into the local cluster cache), leading to remote bandwidth being one-third of the local bandwidth.

5.4 Overall Comparison of DSM Systems

The major difference between the Convex, Sequent and DG machines and the Origin is that the Origin has a much more tightly integrated DSM structure with the assumption of treating local accesses as an optimization of a general DSM memory reference. The other commercial DSM machines add a DSM mechanism on top of a relatively large SMP node in which local communication and memory accesses are optimized over performance of the general DSM structure. Which one of these models is more appropriate depends upon a number of factors:

1. Is scaling the system down in size and cost important? The overhead of the SMP nodes sets a minimum on how effective such machines can be in small configurations (one to two processors).

2. Is the workload primarily throughput oriented, with only a small degree of parallelism? If so, the SMP solutions might achieve higher performance due to lower communication costs between processors within the same SMP node. Oppositely, if parallelism beyond the size of the SMP node is important, then the latency and bandwidth overheads are likely to be very high in comparison to Origin.
3. Is I/O structured as a global resource or must nodes treat I/O as a local resource similar to a distributed memory system? If the goal is global accessibility then the tight DSM integration of Origin would also be preferred.

6 Conclusions

The Origin 2000 is a highly scalable server designed to meet the needs of both the technical and commercial marketplaces. This is accomplished by providing a highly modular system with a low-entry point and incremental costs. A bristled fat hypercube network is used to provide a high bisection bandwidth, low-latency interconnect. Low latency to local memory and a low remote to local memory latency ratio allow the existing application base to easily migrate their applications from the uniform access of the existing SMP Challenge and Power Challenge systems to the NUMA Origin systems. Origin also includes several features to help the performance of these applications including hardware and software support for page migration and fast synchronization.

7 Acknowledgments

The Origin system design resulted from the very hard work of a top-notch team of chip, board, and system engineers. Major contributors on the core logic design/verification team besides the authors included: Michael Anderson, John Andrews, Rick Bahr, John Burger, John Carlson, Hansel Collins, Pat Conway, Ken Choy, Asgeir Eriksson, Paul Everhardt, Mike Galles, Sameer Gupta, Gary Hagensen, Dick Hessel, Roger Hu, Lee Jones, George Kaldani, John Keen, Ron Kolb, Yuval Koren, Waidy Lee, Viranji Madan, John Manton, Greg Marlan, Dawn Maxon, David McCracken, Ali Moyedian, Bob Newhall, Kianoosh Naghshineh, Chuck Narad, Ron Nikel, Steve Padnos, Dave Parry, Ed Priest, Azmeer Salleh, Ken Sarocky, Chris Satterlee, Alex Silbey, Doug Solomon, Jim Smith, Tuan Tran, Swami Venkataraman, Rich Weber, Eric Williams, Mike Woodacre, and Steve Yurash.

The authors would also like to thank Jaswinder Pal Singh and Dongming Jiang for their help in getting the SPLASH performance numbers on Origin, John McCalpin for providing the NAS Parallel Benchmark and STREAM results, and the anonymous referees for their comments which helped to improve the final version of this paper.

References

- [1] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2-13, June 1995.
- [2] Tony Brewer and Greg Astfalk. The evolution of the HP/Convex Exemplar. In *Proceedings of COMPCON Spring '97: Forty-Second IEEE Computer Society International Conference*, pages 81-86, February 1997.
- [3] Asgeir Th. Eriksson and Ken L. McMillan. Using formal verification/analysis methods on the critical path in system design: A case study. In *Proceedings of Computer Aided Verification Conference*, Liege Belgium, LNCS 939, Springer Verlag, 1995.
- [4] Mike Galles. Scalable Pipelined Interconnect for Distributed Endpoint Routing: The SGI SPIDER chip. In *Hot Interconnects '96*.
- [5] Mark Heinrich, Jeffrey Kuskin, David Ofelt, John Heinlein, Joel Baxter, Jaswinder Pal Singh, Richard Simoni, Kourosh Gharachorloo, David Nakahira, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The performance impact of flexibility in the Stanford FLASH multiprocessor. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274-285, October 1994.
- [6] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148-159, May 1990.
- [7] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH prototype: Logic overhead and performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41-61, January 1993.
- [8] Tom Lovett and Russell Clapp. STiNG: A CC-NUMA computer system for the commercial marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308-317, May 1996.
- [9] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325-336, April 1994.
- [10] Mendel Rosenblum, John Chapin, Dan Teodosiu, Scott Devine, Tirthankar Lahiri, and Anoop Gupta. Implementing efficient fault containment for multiprocessors. *Communications of the ACM*, 39(3):52-61, September, 1996.
- [11] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279-289, October 1996.
- [12] Wolf-Dietrich Weber. *Scalable Directories for Cache-Coherent Shared-Memory Multiprocessors*. Ph.D.thesis, Stanford University, Stanford, California, January 1993.
- [13] Steve Whitney, John McCalpin, Nawaf Bitar, John L. Richardson, and Luis Stevens. The SGI Origin software environment and application performance. In *Proceedings of COMPCON Spring '97: Forty-Second IEEE Computer Society International Conference*, pages 165-170, February 1997.
- [14] Kenneth Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28-40, April, 1996.

Instruction-Level Parallel Processing: History, Overview, and Perspective

B. RAMAKRISHNA RAU AND JOSEPH A. FISHER
Hewlett-Packard Laboratories, 1501 Page Mill Road, Bldg. 3U, Palo Alto, CA 94304

(October 20, 1992)

Abstract. Instruction-level parallelism (ILP) is a family of processor and compiler design techniques that speed up execution by causing individual machine operations to execute in parallel. Although ILP has appeared in the highest performance uniprocessors for the past 30 years, the 1980s saw it become a much more significant force in computer design. Several systems were built and sold commercially, which pushed ILP far beyond where it had been before, both in terms of the amount of ILP offered and in the central role ILP played in the design of the system. By the end of the decade, advanced microprocessor design at all major CPU manufacturers had incorporated ILP, and new techniques for ILP had become a popular topic at academic conferences. This article provides an overview and historical perspective of the field of ILP and its development over the past three decades.

Keywords. Instruction-level parallelism, VLIW processors, superscalar processors, pipelining, multiple operation issue, speculative execution, scheduling, register allocation.

1. Introduction

Instruction-level parallelism (ILP) is a family of processor and compiler design techniques that speed up execution by causing individual machine operations, such as memory loads and stores, integer additions, and floating point multiplications, to execute in parallel. The operations involved are normal RISC-style operations, and the system is handed a single program written with a sequential processor in mind. Thus an important feature of these techniques is that like circuit speed improvements, but unlike traditional multiprocessor parallelism and massive parallel processing, they are largely transparent to users. VLIWs and superscalars are examples of processors that derive their benefit from instruction-level parallelism, and software pipelining and trace scheduling are example software techniques that expose the parallelism that these processors can use.

Although small amounts of ILP have been present in the highest performance uniprocessors of the past 30 years, the 1980s saw it become a much more significant force in computer design. Several systems were built and sold commercially, which pushed ILP far beyond where it had been before, both in terms of the amount of ILP offered and in the central role ILP played in the design of the system. By the early 1990s, advanced microprocessor design at all major CPU manufacturers incorporated ILP, and new techniques for ILP became a popular topic at academic conferences. With all of this activity we felt that, in contrast to a report on suggested future techniques, there would be great value in gathering, in an archival reference, reports on experience with real ILP systems and reports on the measured potential of ILP. Thus this special issue of *The Journal of Supercomputing*.

1.1. ILP Execution

A typical ILP processor has the same type of execution hardware as a normal RISC machine. The differences between a machine with ILP and one without is that there may be more of that hardware, for example, several integer adders instead of just one, and that the control will allow, and possibly arrange, simultaneous access to whatever execution hardware is present.

Consider the execution hardware of a simplified ILP processor consisting of four functional units and a branch unit connected to a common register file (Table 1). Typically ILP execution hardware allows multiple-cycle operations to be pipelined, so we may assume that a total of four operations can be initiated each cycle. If in each cycle the longest latency operation is issued, this hardware could have ten operations "in flight" at once, which would give it a maximum possible speedup of a factor of ten over a sequential processor with similar execution hardware. As the papers in this issue show, this execution hardware resembles that of several VLIW processors that have been built and used commercially, though it is more limited in its amount of ILP. Several superscalar processors now being built also offer a similar amount of ILP.

There is a large amount of parallelism available even in this simple processor. The challenge is to make good use of it—we will see that with the technology available today, an ILP processor is unlikely to achieve nearly as much as a factor of ten on many classes of programs, though scientific programs and others can yield far more than that on a processor that has more functional units. The first question that comes to mind is whether enough ILP exists in programs to make this possible. Then, if this is so, what must the compiler and hardware do to successfully exploit it? In reality, as we shall see in Section 4, the two questions have to be reversed; in the absence of techniques to find and exploit ILP, it remains hidden, and we are left with a pessimistic answer.

Figure 1a shows a very large expression taken from the inner loop of a compute-intensive program. It is presented cycle by cycle as it might execute on a processor with functional units similar to those shown in Table 1, but capable of having only one operation in flight

Table 1. Execution hardware for a simplified ILP processor.

Functional Unit	Operations Performed	Latency
Integer unit 1	Integer ALU operations	1
	Integer multiplication	2
	Loads	2
	Stores	1
Integer unit 2/branch unit	Integer ALU operations	1
	Integer multiplication	2
	Loads	2
	Stores	1
	Test-and-branch	1
Floating point unit 1	Floating point operations	3
Floating point unit 2		

```

CYCLE 1 xseed1 = xseed * 1309
CYCLE 2 nop
CYCLE 3 nop
CYCLE 4 yseed1 = yseed * 1308
CYCLE 5 nop
CYCLE 6 nop
CYCLE 7 xseed2 = xseed1 + 13849
CYCLE 8 yseed2 = yseed1 + 13849
CYCLE 9 xseed = xseed2 && 65535
CYCLE 10 yseed = yseed2 && 65535
CYCLE 11 tseed1 = tseed * 1307
CYCLE 12 nop
CYCLE 13 nop
CYCLE 14 vseed1 = vseed * 1306
CYCLE 15 nop
CYCLE 16 nop
CYCLE 17 tseed2 = tseed1 + 13849
CYCLE 18 vseed2 = vseed1 + 13849
CYCLE 19 tseed = tseed2 && 65535
CYCLE 20 vseed = vseed2 && 65535
CYCLE 21 xsq = xseed * xseed
CYCLE 22 nop
CYCLE 23 nop
CYCLE 24 ysq = yseed * yseed
CYCLE 25 nop
CYCLE 26 nop
CYCLE 27 xysumsq = xsq + ysq
CYCLE 28 tsq = tseed * tseed
CYCLE 29 nop
CYCLE 30 nop
CYCLE 31 vsq = vseed * vseed
CYCLE 32 nop
CYCLE 33 nop
CYCLE 34 tvsumsq = tsq + vsq
CYCLE 35 plc = plc + 1
CYCLE 36 tp = tp + 2
CYCLE 37 if xysumsq > radius goto @xy-no-hit

```

(a)

INT ALU	INT ALU	FLOAT ALU	FLOAT ALU
CYCLE 1 tp=tp+2	plc=plc+1	vseed1=vseed*1306	tseed1=tseed*1307
CYCLE 2		yseed1=yseed*1308	xseed1=xseed*1309
CYCLE 3 nop			
CYCLE 4 vseed2=vseed1+13849	tseed2=tseed1+13849		
CYCLE 5 yseed2=yseed1+13849	xseed2=xseed1+13849		
CYCLE 6 yseed=yseed2&&65535	xseed=xseed2&&65535		
CYCLE 7 vseed=vseed2&&65535	tseed=tseed2&&65535	ysq=yseed*yseed	xsq=xseed*xseed
CYCLE 8		vsq=vseed*vseed	tsq=tseed*tseed
CYCLE 9 nop			
CYCLE 0 xysumsq=xsq+ysq			
CYCLE 11 tvsumsq=tsq+vsq			
		if xysumsq>radius goto @xy-no-hit	

(b)

Figure 1. (a) An example of the sequential record of execution for a loop. (b) The instruction-level parallel record of execution for the same loop.

at a time. Figure 1b shows the same program fragment as it might be executed on the hardware indicated in Table 1.

Note that several of the cycles in Figure 1a contain no-ops. This is because the sequential processor must await the completion of the three-cycle latency multiply issued in cycle 1 before issuing the next operation. (These no-ops would not appear in the text of a program, but are shown here as the actual record of what is executed each cycle.) Most instruction-level parallel processors can issue operations during these no-op cycles, when previous operations are still in flight, and many can issue more than one operation in a given cycle.

In our ILP record of execution (Figure 1b), both effects are evident: In cycle 1, four operations are issued; in cycle 2, two more operations are issued even though neither multiply in cycle 1 has yet completed execution.

This special issue of *The Journal of Supercomputing* concerns itself with the technology of systems that try to attain the kind of record of execution in Figure 1b, given a program written with the record of execution in Figure 1a in mind.

1.2. Early History of Instruction-Level Parallelism

In small ways, instruction-level parallelism factored into the thinking of machine designers in the 1940s and 1950s. Parallelism that would today be called horizontal microcode appeared in Turing's 1946 design of the Pilot ACE [Carpenter and Doran 1986] and was carefully described by Wilkes [1951]. Indeed, in 1953 Wilkes and Stringer wrote, "In some cases it may be possible for two or more micro-operations to take place at the same time" [Wilkes and Stringer 1953].

The 1960s saw the appearance of transistorized computers. One effect of this revolution was that it became practical to build reliable machines with far more gates than was necessary to build a general-purpose CPU. This led to commercially successful machines that used this available hardware to provide instruction-level parallelism at the machine-language level. In 1963 Control Data Corporation started delivering its CDC 6600 [Thornton 1964, 1970], which had ten functional units—integer add, shift, increment (2), multiply (2), logical branch, floating point add and divide. Any one of these could start executing in a given cycle whether or not others were still processing data-independent earlier operations. In this machine the hardware decided, as the program executed, which operation to issue in a given cycle; its model of execution was well along the way toward what we would today call superscalar. Indeed, in many ways it strongly resembled its direct descendant, the scalar portion of the CRAY-1. The CDC 6600 was the scientific supercomputer of its day.

Also during the 1960s, IBM introduced, and in 1967–68 delivered, the 360/91 [IBM 1967]. This machine, based partly on IBM's instruction-level parallel experimental Stretch processor, offered less instruction-level parallelism than the CDC 6600, having only a single integer adder, a floating point adder, and a floating point multiply/divide. But it was far more ambitious than the CDC 6600 in its attempt to rearrange the instruction stream to keep these functional units busy—a key technology in today's superscalar designs. For various nontechnical reasons the 360/91 was not as commercially successful as it might have been, with only about 20 machines delivered [Bell and Newell 1971]. But its CPU architecture was the start of a long line of successful high-performance processors. As with the CDC 6600, this ILP pioneer started a chain of superscalar architectures that has lasted into the 1990s.

In the 1960s, research into "parallel processing" often was concerned with the ILP found in these processors. By the mid-1970s the term was used more often for multiple processor parallelism and for regular array and vector parallelism. In part, this was due to some very pessimistic results about the availability of ILP in ordinary programs, which we discuss below.

1.3. Modern Instruction-Level Parallelism

In the late 1970s the beginnings of a new style of ILP, called very long instruction word (VLIW), emerged on several different fronts. In many ways VLIWs were a natural outgrowth of horizontal microcode, the first ILP technology, and they were triggered, in the 1980s, by the same changes in semiconductor technology that had such a profound impact upon the entire computer industry.

For sequential processors, as the speed gap between writeable and read-only memory narrowed, the advantages of a small, dedicated, read-only control store began to disappear. One natural effect of this was to diminish the advantage of microcode; it no longer made as much sense to define a complex language as a compiler target and then interpret this in very fast read-only microcode. Instead, the vertical microcode interface was presented as a clean, simple compiler target. This concept was called RISC [Hennessy, Jouppi, Baskett et al. 1982; Patterson and Sequin 1981; Radin 1982]. In the 1980s the general movement of microprocessor products was towards the RISC concept, and instruction-level parallel techniques fell out of favor. In the minisupercomputer price-bracket though, one innovative superscalar product, the ZS-1, which could issue up to two instructions each cycle, was built and marketed by Astronautics [Smith et al. 1987].

The same changes in memory technology were having a somewhat different effect upon horizontally microcoded processors. During the 1970s a large market had grown in specialized signal processing computers. Not aimed at general-purpose use, these CPUs hard-wired FFTs and other important algorithms directly into the horizontal control store, gaining tremendous advantages from the instruction-level parallelism available there. When fast, writeable memory became available, some of these manufacturers, most notably Floating Point Systems [Charlesworth 1981], replaced the read-only control store with writeable memory, giving users access to instruction-level parallelism in far greater amounts than the early superscalar processors had. These machines were extremely fast, the fastest processors by far in their price ranges, for important classes of scientific applications. However, despite attempts on the part of several manufacturers to market their products for more general, everyday use, they were almost always restricted to a narrow class of applications. This was caused by the lack of good system software, which in turn was caused by the idiosyncratic architecture of processors built for a single application, and by the lack at that time of good code generation algorithms for ILP machines with that much parallelism.

As with RISC, the crucial step was to present a simple, clean interface to the compiler. However, in this case the clean interface was horizontal, not vertical, so as to afford greater ILP [Fisher 1983; Rau, Glaeser, and Greenawalt 1982]. This style of architecture was dubbed VLIW [Fisher 1983]. Code generation techniques, some of which had been developed for generating horizontal microcode, were extended to these general-purpose VLIW machines so that the compiler could specify the parallelism directly [Fisher 1981; Rau and Glaeser 1981].

In the 1980s VLIW CPUs were offered commercially in the form of capable, general-purpose machines. Three computer start-ups—Culler, Multiflow, and Cydrome—built VLIWs with varying degrees of parallelism [Colwell et al. 1988; Rau et al. 1989]. As a group these companies were able to demonstrate that it was possible to build practical machines that achieved large amounts of ILP on scientific and engineering codes. Although,

for various reasons, none was a lasting business success, several major computer manufacturers acquired access to the technologies developed at these start-ups and there are several active VLIW design efforts underway. Furthermore, many of the compiler techniques developed with VLIWs in mind, and reported upon in this issue, have been used to compile for superscalar machines as well.

1.3.1. ILP in the 1990s. Just as had happened 30 years ago when the transistor became available, CPU designers in the 1990s now have offered to them more silicon space on a single chip than a RISC processor requires. Virtually all designers have begun to add some degree of superscalar capability, and some are investigating VLIWs as well. It is a safe bet that by 1995 virtually all new CPUs will embody some degree of ILP.

Partly as a result of this commercial resurgence of interest in ILP, research into that area has become a dominant feature of architecture and systems conferences of the 1990s. Unfortunately, those researchers who found themselves designing state-of-the-art products at computer start-ups did not have the time to document the progress that was made and the large amount that was learned. Virtually everything that was done by these groups was relevant to what designers wrestle with today.

2. ILP Architectures

The end result of instruction-level parallel execution is that multiple operations are simultaneously in execution, either as a result of having been issued simultaneously or because the time to execute an operation is greater than the interval between the issuance of successive operations. How exactly are the necessary decisions made as to when an operation should be executed and whether an operation should be speculatively executed? The alternatives can be broken down depending on the extent to which these decisions are made by the compiler rather than by the hardware and on the manner in which information regarding parallelism is communicated by the compiler to the hardware via the program.

A computer architecture is a contract between the class of programs that are written for the architecture and the set of processor implementations of that architecture. Usually this contract is concerned with the instruction format and the interpretation of the bits that constitute an instruction, but in the case of ILP architectures it extends to information embedded in the program pertaining to the available parallelism between the instructions or operations in the program. With this in mind, ILP architectures can be classified as follows.

- **Sequential architectures:** architectures for which the program is not expected to convey any explicit information regarding parallelism. Superscalar processors are representative of ILP processor implementations for sequential architectures [Anderson et al. 1967; Apollo Computer 1988; Bahr et al. 1991; Blanck and Krueger 1992; DeLano et al. 1992; Diefendorff and Allen 1992; IBM 1990; Intel 1989b; Keller et al. 1975; Popescu et al. 1991; Smith et al. 1987; Thompson 1964].
- **Dependence architectures:** architectures for which the program explicitly indicates the dependences that exist between operations. Dataflow processors [Arvind and Gostelow 1982; Arvind and Kathail 1981; Gurd et al. 1985] are representative of this class.

- *Independence architectures*: architectures for which the program provides information as to which operations are independent of one another. Very long instruction word (VLIW) processors [Charlesworth 1981; Colwell et al. 1988; Rau et al. 1989] are examples of the class of independence architectures.

In the context of this taxonomy, vector processors [Hintz and Tate 1972; Russell 1978; Watson 1972] are best thought of as processors for a sequential, CISC (complex instruction set computer) architecture. The complex instructions are the vector instructions that do possess a stylized form of instruction-level parallelism internal to each vector instruction. Attempting to execute multiple instructions in parallel, whether scalar or vector, incurs all of the same problems that are faced by a superscalar processor. Because of their stylized approach to parallelism, vector processors are less general in their ability to exploit all forms of instruction-level parallelism. Nevertheless, vector processors have enjoyed great commercial success over the past decade. Not being true ILP processors, vector processors are outside the scope of this special issue. (Vector processors have received a great deal of attention elsewhere over the past decade and have been treated extensively in many books and articles, for instance, the survey by Dongarra [1986] and the book by Schneek [1987].) Also, certain hybrid architectures [Danelutto and Vanneschi 1990; Franklin and Sohi 1992; Wolfe and Shen 1991], which also combine some degree of multithreading with ILP, fall outside of this taxonomy for uniprocessors.

If ILP is to be achieved, between the compiler and the run-time hardware, the following functions must be performed:

1. The dependences between operations must be determined.
2. The operations that are independent of any operation that has not as yet completed must be determined.
3. These independent operations must be scheduled to execute at some particular time, on some specific functional unit, and must be assigned a register into which the result may be deposited.

Figure 2 shows the breakdown of these three tasks, between the compiler and run-time hardware, for the three classes of architecture.

2.1. Sequential Architectures and Superscalar Processors

The program for a sequential architecture contains no explicit information regarding the dependences that exist between instructions. Consequently, the compiler need neither identify parallelism nor make scheduling decisions since there is no explicit way to communicate this information to the hardware. (It is true, nevertheless, that there is value in the compiler performing these functions and ordering the instructions so as to facilitate the hardware's task of extracting parallelism.) In any event, if instruction-level parallelism is to be employed, the dependences that exist between instructions must be determined by the hardware. It is only necessary to determine dependences with sequentially preceding operations that are in flight, that is, those that have been issued but have not yet completed.

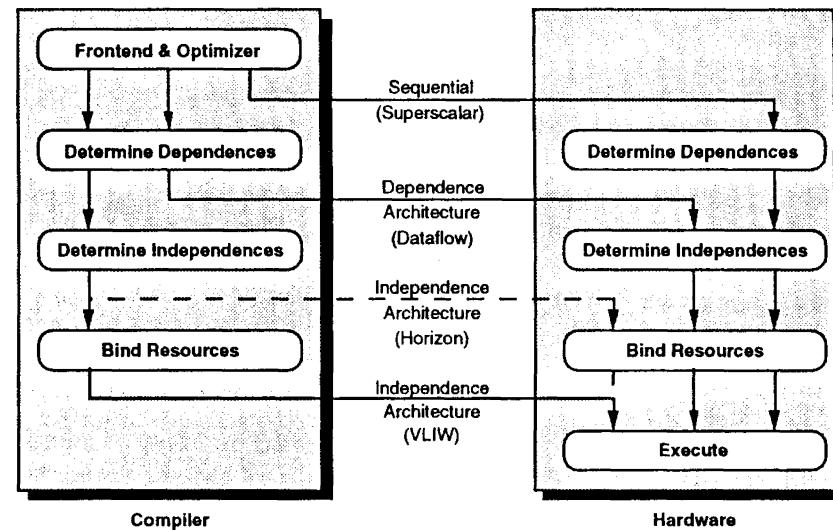


Figure 2. Division of responsibilities between the compiler and the hardware for the three classes of architecture.

When the operation is independent of all other operations it may begin execution. At this point the hardware must make the scheduling decision of when and where this operation is to execute.

A superscalar processor¹ strives to issue an instruction every cycle so as to execute many instructions in parallel, even though the hardware is handed a sequential program. The problem is that a sequential program is constructed with the assumption only that it will execute correctly when each instruction waits for the previous one to finish, and that is the only order that the architecture guarantees to be correct. The first task, then, for a superscalar processor is to understand, for each instruction, which other instructions it actually is dependent upon. With every instruction that a superscalar processor issues, it must check whether the instruction's operands (registers or memory locations that the instruction uses or modifies) interfere with the operands of any other instruction in flight, that is, one that is either

- already in execution or
- has been issued but is waiting for the completion of interfering instructions that would have been executed earlier in a sequential execution of the program.

If either of these conditions is true, the instruction in question must be delayed until the instructions on which it is dependent have completed execution. For each waiting operation, these dependences must be monitored to determine the point at which neither condition is true. When this happens, the instruction is independent of all other uncompleted instructions and can be allowed to begin executing at any time thereafter. In the meantime the processor may begin execution of subsequent instructions that prove to be independent

of all sequentially preceding instructions in flight. Once an instruction is independent of all other ones in flight, the hardware must also decide exactly when and on which available functional unit to execute the instruction. The Control Data CDC 6600 used a mechanism, called the *scoreboard*, to perform these functions [Thornton 1964]. The IBM System/360 Model 91, built in the early 1960s, used an even more sophisticated method known as Tomasulo's algorithm to carry out these functions [Tomasulo 1967].

The further goal of a superscalar processor is to issue *multiple* instructions every cycle. The most problematic aspect of doing so is determining the dependences between the operations that one wishes to issue simultaneously. Since the semantics of the program, and in particular the essential dependences, are specified by the sequential ordering of the operations, the operations must be processed in this order to determine the essential dependences. This constitutes an unacceptable performance bottleneck in a machine that is attempting parallel execution. On the other hand, eliminating this bottleneck can be very expensive, as is always the case when attempting to execute an inherently sequential task in parallel. An excellent reference on superscalar processor design and its complexity is the book by Johnson [1991].

A number of superscalar processors have been built during the past decade including the Astronautics' ZS-1 decoupled access minisupercomputer [Smith 1989; Smith et al. 1987], Apollo's DN10000 personal supercomputer [Apollo 1988; Bahr et al. 1991], and, most recently, a number of microprocessors [Blanck and Krueger 1992; DeLano et al. 1992; Diefendorff and Allen 1992; IBM 1990; Intel 1989b; Popescu et al. 1991].

Note that an ILP processor need not issue multiple operations per cycle in order to achieve a certain level of performance. For instance, instead of a processor capable of issuing five instructions per cycle, the same performance could be achieved by pipelining the functional units and instruction issue hardware five times as deeply, speeding up the clock rate by a factor of five but issuing only one instruction per cycle. This strategy, which has been termed *superpipelining* [Jouppi 1989], goes full circle back to the single-issue, superscalar processing of the 1960s. Superpipelining may result in some parts of the processor (such as the instruction unit and communications buses) being less expensive and better utilized and other parts (such as the execution hardware) being more costly and less well used.

2.2. Dependence Architectures and Dataflow Processors

In the case of dependence architectures the compiler or the programmer identifies the parallelism in the program and communicates it to the hardware by specifying, in the executable program, the dependences between operations. The hardware must still determine, at run time, when each operation is independent of all other operations and then perform the scheduling. However, the inherently sequential task, of scanning the sequential program in its original order to determine the dependences, has been eliminated.

The objective of a dataflow processor is to execute an instruction at the earliest possible time subject only to the availability of the input operands and a functional unit upon which to execute the instruction [Arvind and Gostelow 1982; Arvind and Kathail 1981]. To do so, it counts on the program to provide information about the dependences between instructions. Typically, this is accomplished by including in each instruction a list of successor

instructions. (An instruction is a successor of another instruction if it uses as one of its input operands the result of that other instruction.) Each time an instruction completes, it creates a copy of its result for each of its successor instructions. As soon as all of the input operands of an instruction are available, the hardware fetches the instruction, which specifies the operation to be performed and the list of successor instructions. The instruction is then executed as soon as a functional unit of the requisite type is available. This property, whereby the availability of the data triggers the fetching and execution of an instruction, is what gives rise to the name of this type of processor. Because of this property, it is redundant for the instruction to specify its input operands. Rather, the input operands specify the instruction! If there is always at least one instruction ready to execute on every functional unit, the dataflow processor achieves peak performance.

Computation within a basic block typically does not provide adequate levels of parallelism. Superscalar and VLIW processors use control parallelism and speculative execution to keep the hardware fully utilized. (This is discussed in greater detail in Sections 3 and 4.) Dataflow processors have traditionally counted on using control parallelism alone to fully utilize the functional units. A dataflow processor is more successful than the others at looking far down the execution path to find abundant control parallelism. When successful, this is a better strategy than speculative execution since every instruction executed is a useful one and the processor does not have to deal with error conditions raised by speculative operations.

As far as the authors are aware, there have been no commercial products built based on the dataflow architecture, except in a limited sense [Schmidt and Caesar 1991]. There have, however, been a number of research prototypes built, for instance, the ones built at the University of Manchester [Gurd et al. 1985] and at MIT [Papadopoulos and Culler 1990].

2.3. Independence Architectures and VLIW Processors

In order to execute operations in parallel, the system must determine that the operations are independent of one another. Superscalar processors and dataflow processors represent two ways of deriving this information at run time. In the case of the dataflow processor the explicitly provided dependence information is used to determine when an instruction may be executed so that it is independent of all other concurrently executing instructions. The superscalar processor must do the same, but since programs for it lack any explicit information, it must also first determine the dependences between instructions. In contrast, for an independence architecture the compiler identifies the parallelism in the program and communicates it to the hardware by specifying which operations are independent of one another. This information is of direct value to the hardware, since it knows with no further checking which operations it can execute in the same cycle. Unfortunately, for any given operation, the number of operations of which it is independent is far greater than the number of operations on which it is dependent, so it is impractical to specify all independencies. Instead, for each operation, independencies with only a subset of all independent operations (those operations that the compiler thinks are the best candidates to execute concurrently) are specified.

By listing operations that could be executed simultaneously, code for an independence architecture may be very close to the record of execution produced by an implementation

of that architecture. If the architecture additionally requires that programs specify where (on which functional unit) and when (in which cycle) the operations are executed, then the hardware makes no run time decisions at all and the code is virtually identical to the desired record of execution. The VLIW processors that have been built to date are of this type and represent the predominant examples of machines with independence architectures. The program for a VLIW processor specifies exactly which functional unit each operation should be executed on and exactly when each operation should be issued so as to be independent of all operations that are being issued at the same time as well as of those that are in execution. A particular processor implementation of a VLIW architecture could choose to disregard the scheduling decisions embedded in the program, making them at run time instead. In doing so, the processor would still benefit from the independence information but would have to perform all of the scheduling tasks of a superscalar processor. Furthermore, when attempting to execute concurrently two operations that the program did not specify as being independent of each other, it must determine independence, just as a superscalar processor must.

With a VLIW processor it is important to distinguish between an instruction and an operation. An operation is a unit of computation, such as an addition, memory load, or branch, which would be referred to as an instruction in the context of a sequential architecture. A VLIW instruction is the set of operations that are intended to be issued simultaneously. It is the task of the compiler to decide which operations should go into each instruction. This process is termed *scheduling*. Conceptually, the compiler schedules a program by emulating at compile time what a dataflow processor, with the same execution hardware, would do at run time. All operations that are supposed to begin at the same time are packaged into a single VLIW instruction. The order of the operations within the instruction specifies the functional unit on which each operation is to execute. A VLIW program is a transliteration of a desired record of execution that is feasible in the context of the given execution hardware.

The compiler for a VLIW machine specifies that an operation be executed speculatively merely by performing speculative code motion, that is, scheduling an operation before the branch that determines that it should, in fact, be executed. At run time, the VLIW processor blindly executes this operation exactly as specified by the program, just as it would for a nonspeculative operation. Speculative execution is virtually transparent to the VLIW processor and requires little additional hardware. When the compiler decides to schedule an operation for speculative execution, it can arrange to leave behind enough of the state of the computation to assure correct results when the flow of the program requires that the operation be ignored. The hardware required for the support of speculative code motion consists of having some extra registers, of fetching some extra instructions, and of suppressing the generation of spurious error conditions. The VLIW compiler must perform many of the same functions that a superscalar processor performs at run time to support speculative execution, but it does so at compile time.

The earliest VLIW processors built were the so-called attached array processors [Charlesworth 1981; Floating Point Systems 1979; IBM 1976; Intel 1989a; Ruggiero and Coryell 1969] of which the best known were the Floating Point Systems products, the AP-I20B, the FPS-164, and the FPS-264. The next generation of products were the minisupercomputers: Multiflow's Trace series of machines [Colwell et al. 1988; Colwell et al. 1990]

and Cydrome's Cydra 5 [Beck et al. 1993; Rau 1988; Rau et al. 1989] and the Culler machine for which, as far as we are aware, there is no published description in the literature. Over the last few years the VLIW architecture has begun to show up in microprocessors [Kohn and Margulis 1989; Labrousse and Slavenburg 1988, 1990a, 1990b; Peterson et al. 1981].

Other types of processors with independence architectures have been built or proposed. A superpipelined machine may issue only one operation per cycle, but if there is no superscalar hardware devoted to preserving the correct execution order of operations, the compiler will have to schedule them with full knowledge of dependences and latencies. From the compiler's point of view these machines are virtually the same as VLIWs, though the hardware design of such a processor offers some tradeoffs with respect to VLIWs. Another proposed independence architecture, dubbed *Horizon* [Thistle and Smith 1988], encodes an integer H into each operation. The architecture guarantees that all of the next H operations in the instruction stream are data-independent of the current operation. All the hardware has to do to release an operation, then, is to assure itself that no more than H subsequent operations are allowed to issue before this operation has completed. The hardware does all of its own scheduling, unlike the VLIWs and deeply pipelined machines that rely on the compiler, but the hardware is relieved of the task of determining data dependence. The key distinguishing features of these three ILP architectures are summarized in Table 2.

Table 2. A comparison of the instruction-level parallel architecture discussed in this paper.

	Sequential Architecture	Dependence Architecture	Independence Architecture
Additional information required in the program	None	Complete specification of dependences between operations	Minimally, a partial list of independences. Typically, a complete specification of when and where each operation is to be executed
Typical kind of ILP processor	Superscalar	Dataflow	VLIW
Analysis of dependences between operations	Performed by hardware	Performed by the compiler	Performed by the compiler
Analysis of independent operations	Performed by hardware	Performed by hardware	Performed by the compiler
Final operation scheduling	Performed by hardware	Performed by hardware	Typically, performed by the compiler
Role of compiler	Rearranges the code to make the analysis and scheduling hardware more successful	Replaces some analysis hardware	Replaces virtually all the analysis and scheduling hardware

3. Hardware and Software Techniques for ILP Execution

Regardless of which ILP architecture is considered, certain functions must be performed if a sequential program is to be executed in an ILP fashion. The program must be analyzed to determine the dependences; the point in time at which an operation is independent of all operations that are as yet not complete, must be determined; scheduling and register allocation must be performed; often, operations must be executed speculatively, which in turn requires that branch prediction be performed. All these functions must be performed. The choice is, first, whether they are to be performed by the compiler or by run-time hardware and, second, which specific technique is to be used. These alternatives are reviewed in the rest of this section.

3.1. Hardware Features to Support ILP Execution

Instruction-level parallelism involves the existence of multiple operations in flight at any one time, that is, operations that have begun, but not completed, executing. This implies the presence of execution hardware that can simultaneously process multiple operations. This has, historically, been achieved by two mechanisms: first, providing multiple, parallel functional units and, second, pipelining the functional units. Although both are fairly similar from a compiler's viewpoint—the compiler must find enough independent operations to keep the functional units busy—they have their relative strengths and weaknesses from a hardware viewpoint.

In principle, pipelining is the more cost-effective way of building ILP execution hardware. For the relatively low cost of adding pipeline latches within each functional unit, the amount of ILP can be doubled, tripled, or more. The limiting factors in increasing the performance by this means are the data and clock skews and the latch setup and hold times. These issues were studied during the 1960s and 1970s, and the upper limits on the extent of pipelining were determined [Chen 1971; Cotten 1965, 1969; Fawcett 1975; Hallin and Flynn 1972]. However, the upper limit on pipelining is not necessarily the best from the viewpoint of achieved performance. Pipelining adds delays to the execution time of individual operations (even though multiples of them can be in flight on the same functional unit). Beyond a certain point, especially on computations that have small amounts of parallelism, the increase in the latency counterbalances the benefits of the increase in ILP, yielding lower performance [Kunkel and Smith 1986]. Parallelism achieved by adding more functional units does not suffer from this drawback, but has its own set of disadvantages. First, the amount of functional unit hardware goes up in linear proportion to the parallelism. Worse, the cost of the interconnection network and the register files goes up proportional to the square of the number of functional units since, ideally, each functional unit's output bus must communicate with every functional unit's input buses through the register file. Also, as the number of loads on each bus increases, so must the cycle time or the extent of pipelining, both of which degrade performance on computation with little parallelism.

The related techniques of pipelining and overlapped execution were employed as early as in the late 1950s in computers such as IBM's STRETCH computer [Bloch 1959; Buchholz 1962] and UNIVAC's LARC [Eckert et al. 1959]. Traditionally, overlapped execution refers

to the parallelism that results from multiple active instructions, each in a different one of the phases of instruction fetch, decode, operand fetch, and execute, whereas pipelining is used in the context of functional units such as multipliers and floating point adders [Chen 1975; Kogge 1981]. (A potential source of confusion is that, in the context of RISC processors, overlapped execution and pipelining, especially when the integer ALU is pipelined, have been referred to as *pipelining* and *superpipelining*, respectively [Jouppi 1989].)

The organization of the register files becomes a major issue when there are multiple functional units operating concurrently. For ease of scheduling, it is desirable that every operation (except loads and stores) be register-register and that the register file be the hub for communication between all the functional units. However, with each functional unit performing two reads and one write per cycle from or to the register file, the implementation of the register file becomes problematic. The chip real estate of a multiported register file is proportional to the product of the number of read ports and the number of write ports. The loading of multiple read ports on each register cell slows down the access time. For these reasons, highly parallel ILP hardware is structured as multiple clusters of functional units, with all the functional units within a single cluster sharing the same multiported register files [Colwell et al. 1988; Colwell et al. 1990; Fisher 1983; Fisher et al. 1984]. Communication between clusters is slower and occurs with lower bandwidth. This places a burden upon the compiler to partition the computation intelligently across the clusters; an inept partitioning can result in worse performance than if just a single cluster were used, leaving the rest of them idle.

The presence of multiple, pipelined function units places increased demands upon the instruction issue unit. In a fully sequential processor, each instruction is issued after the previous one has completed. Of course, this totally defeats the benefits of parallel execution hardware. However, if the instruction unit attempts to issue an instruction every cycle, care must be taken not to do so if an instruction, upon which this one is dependent, is still not complete. The scoreboard in the CDC 6600 [Thornton 1964] was capable of issuing an instruction every cycle until an output dependence was discovered. In the process, instructions following one that was waiting on a flow dependence could begin execution. This was the first implementation of an out-of-order execution scheme. Stalling instruction issue is unnecessary on encountering an output dependence if register renaming is performed. The Tomasulo algorithm [Tomasulo 1967], which was implemented in the IBM System/360 Model 91 [Anderson et al. 1967], is the classical scheme for register renaming and has served as the model for subsequent variations [Hwu and Patt 1986, 1987; Oehler and Blasgen 1991; Popescu et al. 1991; Weiss and Smith 1984]. A different, programmatically controlled register renaming scheme is obtained by providing rotating register files, that is, base-displacement indexing into the register file using an instruction-provided displacement off a dedicated base register [Advanced Micro Devices 1989; Charlesworth 1981; Rau 1988; Rau et al. 1989]. Although applicable only for renaming registers across multiple iterations of a loop, rotating registers have the advantage of being considerably less expensive in their implementation than are other renaming schemes.

The first consideration given to the possibility of issuing multiple instructions per cycle from a sequential program was by Tjaden and Flynn [1970]. This line of investigation into the logic needed to perform multiple-issue was continued by various researchers [Acosta et al. 1986; Dwyer and Torng 1992; Hwu and Patt 1986, 1987; Tjaden and Flynn 1973;

Uht 1986; Wedig 1982]. This idea, of multiple instruction issue of sequential programs, was probably first referred to as superscalar execution by Agerwala and Cocke [1987]. A careful assessment of the complexity of the control logic involved in superscalar processors is provided by Johnson [1991]. An interesting variation on multiple-issue, which made use of architecturally visible queues to simplify the out-of-order execution logic, was the decoupled access/execute architecture proposed by Smith [1982] and subsequently developed as a commercial product [Smith 1989; Smith et al. 1987].

A completely different approach to achieving multiple instruction issue, which grew out of horizontal microprogramming, was represented by attached-processor products such as the Floating Point Systems AP-I20B [Floating Point Systems 1979], the Polycyclic project at ESL [Rau and Glaeser 1981; Rau, Glaeser, and Greenwalt 1982; Rau, Glaeser, and Picard 1982], the Stanford University MIPS project [Hennessy, Jouppi, Przybelski et al. 1982] and the ELI project at Yale [Fisher 1983; Fisher et al. 1984]. The concept is to have the compiler decide which operations should be issued in parallel and to group them in a single, long instruction. This style of architecture, which was dubbed a *very long instruction word* (VLIW) architecture [Fisher 1983], has the advantage that the instruction issue logic is trivial in comparison to that for a superscalar machine, but suffers the disadvantage that the set of operations that are to be issued simultaneously is fixed once and for all at compile time. One of the implications of issuing multiple operations per instruction is that one needs the ability to issue (and process) multiple branches per second. Various types of multiway branches, each corresponding to a different detailed model of execution or compilation, have been suggested [Colwell et al. 1988; Ebcioğlu 1988; Fisher 1980; Nicolau 1985a].

The first obstacle that one encounters when attempting ILP computation is the generally small size of basic blocks. In light of the pipeline latencies and the interoperation dependences, little instruction-level parallelism is to be found. It is important that operations from multiple basic blocks be executed concurrently if a parallel machine is to be fully utilized. Since the branch condition, which determines which block is to be executed next, is often resolved only at the end of a basic block, it is necessary to resort to speculative execution, that is, continuing execution along one or more paths before it is known which way the branch will go. Dynamic schemes for speculative execution [Hwu and Patt 1986, 1987; Smith and Pleszkun 1988; Sohi and Vajapayem 1987] must provide ways to

- terminate unnecessary speculative computation once the branch has been resolved,
- undo the effects of the speculatively executed operations that should not have been executed,
- ensure that no exceptions are reported until it is known that the excepting operation should, in fact, have been executed, and
- preserve enough execution state at each speculative branch point to enable execution to resume down the correct path if the speculative execution happened to proceed down the wrong one.

All this can be expensive in hardware. The alternative is to perform speculative code motion at compile time, that is, move operations from subsequent blocks up past branch operations into preceding blocks. These operations will end up being executed before the branch that they were supposed to follow; hence, they are executed speculatively. Such code motion is fundamental to global scheduling schemes such as trace scheduling [Ellis

1985; Fisher 1979, 1981]. The hardware support needed is much less demanding: first, a mechanism to ensure that exceptions caused by speculatively scheduled operations are reported if and only if the flow of control is such that they would have been executed in the nonspeculative version of the code [Mahlke, Chen et al. 1992] and, second, additional architecturally visible registers to hold the speculative execution state. A limited form of speculative code motion is provided by the “boosting” scheme [Smith et al. 1992; Smith et al. 1990].

Since all speculative computation is wasted if the wrong path is followed, it is important that accurate branch prediction be used to guide speculative execution. Various dynamic schemes of varying levels of sophistication and practicality have been suggested that gather execution statistics of one form or another while the program is running [Lee and Smith 1984; McFarling and Hennessy 1986; Smith 1981; Yeh and Patt 1992]. The alternative is to use profiling runs to gather the appropriate statistics and to embed the prediction, at compile time, into the program. Trace scheduling and superblock scheduling [Hwu et al. 1989; Hwu et al. 1993] use this approach to reorder the control flow graph to reflect the expected branch behavior. Hwu and others claim better performance than with dynamic branch prediction [Hwu et al. 1989]. Fisher and Freudenberger [1992] have examined the extent to which branch statistics gathered using one set of data are applicable to subsequent runs with different data. Although static prediction can be useful for guiding both static and dynamic speculation, it is not apparent how dynamic prediction can assist static speculative code motion.

Predicted execution is an architectural feature that permits the execution of individual operations to be determined by an additional, Boolean input. It has been used to selectively squash operations that have been moved up from successor blocks into the delay slots of a branch operation [Ebcioğlu 1988; Hsu and Davidson 1986]. In its more general form [Beck et al. 1993; Rau 1988; Rau et al. 1989] it is used to eliminate branches in their entirety over an acyclic region of a control flow graph [Dehnert and Towle 1993; Dehnert et al. 1989; Mahlke, Lin et al. 1992] that has been IF-converted [Allen et al. 1983].

3.2. ILP Compilation

3.2.1. Scheduling. Scheduling algorithms can be classified based on two broad criteria. The first one is the nature of the control flow graph that can be scheduled by the algorithm. The control flow graph can be described by the following two properties:

- whether it consists of a single basic block or multiple basic blocks, and
- whether it is an acyclic or cyclic control flow graph.

Algorithms that can only schedule single acyclic basic blocks are known as *local scheduling* algorithms. Algorithms that jointly schedule multiple basic blocks (even if these are multiple iterations of a single static basic block) are termed *global scheduling* algorithms. Acyclic global scheduling algorithms deal either with control flow graphs that contain no cycles or, more typically, cyclic graphs for which a self-imposed scheduling barrier exists

at each back edge in the control flow graph. As a consequence of these scheduling barriers, back edges present no opportunity to the scheduler and are therefore irrelevant to it. Acyclic schedulers can yield better performance on cyclic graphs by unrolling the loop, a transformation which though easier to visualize for cyclic graphs with a single back edge, can be generalized to arbitrary cyclic graphs. The benefit of this transformation is that the acyclic scheduler now has multiple iterations' worth of computation to work with and overlap. The penalty of the scheduling barrier is amortized over more computation. Cyclic global scheduling algorithms attempt to directly optimize the schedule across back edges as well. Each class of scheduling algorithms is more general than the previous one and, as we shall see, attempts to build on the intuition and heuristics of the simpler, less general algorithm. As might be expected, the more general algorithms experience greater difficulty in achieving near-optimality or of even articulating intuitively appealing heuristics.

The second classifying criterion is the type of machine for which scheduling is being performed, which in turn is described by the following assumed properties of the machine:

- finite versus unbounded resources
- unit latency versus multiple cycle latency execution, and
- simple resource usage patterns for every operation (i.e., each operation uses just one resource for a single cycle, typically during the first cycle of the operation's execution) versus more complex resource usage patterns for some or all of the operations.

Needless to say, real machines have finite resources, generally have at least a few operations that have latencies greater than one cycle, and often have at least a few operations with complex usage patterns. We believe that the value of a scheduling algorithm is proportional to the degree of realism of the assumed machine model.

Finally, the scheduling algorithm can also be categorized by the nature of the process involved in generating the schedule. At one extreme are one-pass algorithms that schedule each operation once and for all. At the other extreme are algorithms that perform an exhaustive, branch-and-bound style of search for the schedule. In between is a spectrum of possibilities such as iterative but nonexhaustive search algorithms or incremental algorithms that make a succession of elementary perturbations to an existing legal schedule to nudge it toward the final solution. This aspect of the scheduling algorithm is immensely important in practice. The further one diverges from a one-pass algorithm, the slower the scheduler gets until, eventually, it is unacceptable in a real-world setting.

3.2.1.1. Local Scheduling. Scheduling, as a part of the code generation process, was first studied extensively in the context of microprogramming. Local scheduling is concerned with generating as short a schedule as possible for the operations within a single basic block; in effect a scheduling barrier is assumed to exist between adjacent basic blocks in the control flow graph. Although it was typically referred to as *local code compaction*,² the similarity to the job of scheduling tasks on processors was soon understood [Adam et al. 1974; Baker 1974; Coffman 1976; Coffman and Graham 1972; Fernandez and Bussel 1973; Gonzalez 1977; Hu 1961; Kasahara and Narita 1984; Kohler 1975; Ramamoorthy et al. 1972], and a number of notions and algorithms from scheduling theory were borrowed by the microprogramming community. Attempts at automating this task have been made since

at least the late 1960s [Agerwala 1976; Davidson et al. 1981; DeWitt 1975; Fisher 1979; 1981; Kleir and Ramamoorthy 1971; Landskov et al. 1989; Ramamoorthy and Gonzalez 1969; Tokoro et al. 1977; Tsuchiya and Gonzalez 1974, 1976; Wood 1978]. Since scheduling is known to be NP-complete [Coffman 1976], the initial focus was on defining adequate heuristics [Dasgupta and Tartar 1976; Fisher 1979; Gonzalez 1977; Mallett 1978; Ramamoorthy and Gonzalez 1969; Ramamoorthy and Tsuchiya 1974]. The consensus was that list scheduling using the highest-level-first priority scheme [Adam et al. 1974; Fisher 1979] is relatively inexpensive computationally (a one-pass algorithm) and near-optimal most of the time. Furthermore, this algorithm has no difficulty in dealing with nonunit execution latencies.

The other dimension in which local scheduling matured was in the degree of realism of the machine model. From an initial model in which each operation used a single resource for a single cycle (the simple resource usage model) and had unit latency, algorithms for local scheduling were gradually generalized to cope with complex resource usage and arbitrary latencies [Dasgupta and Tartar 1976; DeWitt 1975; Kleir 1974; Mallett 1978; Ramamoorthy and Tsuchiya 1974; Tsuchiya and Gonzalez 1974; Yau et al. 1974] culminating in the fully general resource usage "microtemplate" model proposed in [Tokoro et al. 1981], and which was known in the hardware pipeline design field as a reservation table [Davidson 1971]. In one form or another, this is now the commonly used machine model in serious instruction schedulers. This machine model is quite compatible with the highest-level-first list scheduling algorithm and does not compromise the near-optimality of this algorithm [Fisher 1981].

3.2.1.2. Global Acyclic Scheduling. A number of studies have established that basic blocks are quite short—typically about 5–20 instructions on the average—so whereas local scheduling can generate a near-optimal schedule, data dependences and execution latencies conspire to make the optimal schedule itself rather disappointing in terms of its speedup over the original sequential code. Further improvements require overlapping the execution of successive basic blocks, which is achieved by global scheduling.

Early strategies for global scheduling attempted to automate and emulate the ad hoc techniques that hand coders practiced of first performing local scheduling of each basic block and then attempting to move operations from one block to an empty slot in a neighboring block [Tokoro et al. 1981; Tokoro et al. 1978]. The shortcoming of such an approach is that, during local compaction, too many arbitrary decisions have already been made that failed to take into account the needs of and opportunities in the neighboring blocks. Many of these decisions might need to be undone before the global schedule can be improved.

In one very important way the mindset inherited from microprogramming was an obstacle to progress in global scheduling. Traditionally, code compaction was focused on the objective of reducing the size of the microprogram so as to allow it to fit in the microprogram memory. In the case of individual basic blocks the objectives of local compaction and local scheduling are aligned. This alignment of objectives is absent in the global case. Whereas global code compaction wishes to minimize the sum of the code sizes for the individual basic blocks, global scheduling must attempt to minimize the total execution time of all the basic blocks. In other words, global scheduling must minimize the sum of the code sizes of the individual basic blocks *weighted by the number of times each basic block is executed*. Thus, effective global scheduling might actually increase the size of the program

by greatly lengthening an infrequently visited basic block in order to slightly reduce the length of a high-frequency basic block. This difference between global compaction and global scheduling, which was captured neither by the early ad hoc techniques nor by the syntactically-driven hierarchical reduction approach proposed by Wood [1979], was noted by Fisher [1979, 1981].

Furthermore, the focus of Fisher's work was on reducing the length of those *sequences* of basic blocks that are frequently executed by the program. These concepts were captured by Fisher in the global scheduling algorithm known as *trace scheduling* [Fisher 1979, 1981]. Central to this procedure is the concept of a trace, which is an acyclic sequence of basic blocks embedded in the control flow graph, that is, a path through the program that could conceivably be taken for some set of input data. Traces are selected and scheduled in order of their frequency of execution. The next trace to be scheduled is defined by selecting the highest frequency basic block that has not yet been scheduled as the seed of the trace. The trace is extended forward along the highest frequency edge out of the last block of the trace as long as that edge is also the most frequent edge into the successor block and as long as the successor block is not already part of the trace. Likewise, the trace is extended backwards, as well, from the seed block. The selected trace is then scheduled as if it were a single block; that is, there is no special consideration given to branches, except that they are constrained to remain in their original order. Implicit in the resulting schedule is inter-block code motion along the trace in either the upward or downward direction. Matching off-trace code motions must be performed as prescribed by the rules of interblock code motion specified by Fisher. This activity is termed *bookkeeping*. Therafter, the next trace is selected and scheduled. This procedure is repeated until the entire program has been scheduled. The key property of trace scheduling is that, unlike previous approaches to global scheduling, the decisions as to whether to move an operation from one block to another, where to schedule it, and which register to allocate to hold its result (see Section 3.2.2 below) are all made jointly rather than in distinct compiler phases.

Fisher and his coworkers at Yale went on to implement trace scheduling in the Bulldog compiler as part of the ELI project [Fisher 1983; Fisher et al. 1986]. This trace scheduling implementation and other aspects of the Bulldog compiler have been extensively documented by Ellis [1986]. The motion of code downwards across branches and upwards across merges results in code replication. Although this is generally acceptable as the price to be paid for better global schedules, Fisher recognized the possibility that the greediness of highest-level-first list scheduling could sometimes cause more code motion and, hence, replication then is needed to achieve a particular schedule length [Fisher 1981]. Su and his colleagues have recommended certain heuristics for the list scheduling of traces to address this problem [Grishman and Su 1983; Su and Ding 1985; Su et al. 1984]. Experiments over a limited set of test cases indicate that these heuristics appear to have the desired effect.

The research performed in the ELI project formed the basis of the production-quality compiler that was built at Multiflow. One of the enhancements to trace scheduling implemented in the Multiflow compiler was the elimination of redundant copies of operations caused by bookkeeping. When an off-trace path, emanating from a branch on the trace, rejoins the trace lower down, an operation that is moved above the rejoin and all the way to a point above the branch can make the off-trace copy redundant under the appropriate circumstances. The original version of trace scheduling, oblivious to such situations, retains

two copies of the operation. Gross and Ward [1990] describe an algorithm to avoid such redundancies. Freudenberg and Ruttenberg [1992] discuss the integrated scheduling and register allocation in the Multiflow compiler. Lowney and others provide a comprehensive description of the Multiflow compiler [1993].

Hwu and his colleagues on the IMPACT project have developed a variant of trace scheduling that they term *superblock scheduling* [Chang, Mahlke et al. 1991; Hwu and Chang 1988]. In an attempt to facilitate the task of incorporating profile-driven global scheduling into more conventional compilers, they separate the trace selection and code replication from the actual scheduling and bookkeeping. To do this, they limit themselves to only moving operations up above branches, never down, and never up past merges. To make this possible, they outlaw control flow into the interior of a trace by means of tail duplication, that is, creating a copy of the trace below the entry point and redirecting the incoming control flow path to that copy. Once this is done for each incoming path, the resulting trace consists of a sequence of basic blocks with branches out of the trace but no incoming branches except to the top of the trace. This constitutes a superblock, also known as an *extended basic block* in the compiler literature. Chang and Hwu [1988] have studied different trace selection strategies and have measured their relative effectiveness. A comprehensive discussion of the results and insights from the IMPACT project are provided in this special issue [Hwu et al. 1993].

Although the global scheduling of linear sequences of basic blocks represents a major step forward, it has been criticized for its total focus on the current trace and neglect of the rest of the program. For instance, if there are two equally frequent paths through the program that have basic blocks in common, it is unclear as part of which trace these blocks should be scheduled. One solution is to replicate the code as is done for superblock scheduling. The other is to generalize trace scheduling to deal with more general control flow graphs. Linn [1988] and Hsu and Davidson [1986] proposed profile-driven algorithms for scheduling trees of basic blocks in which all but the root basic block have a single incoming path. Nicolau [1985a, 1985b] attempted to extend global scheduling to arbitrary, acyclic control flow graphs using percolation scheduling. However, since percolation scheduling assumes unbounded resources, it cannot realistically be viewed as a scheduling algorithm. Percolation scheduling was then extended to nonunit execution latencies (but still with unbounded resources) [Nicolau and Potasman 1990].

The development of practical algorithms for the global scheduling of arbitrary, acyclic control flow graphs is an area of active research. Preliminary algorithms, assuming finite resources have been defined by Ebcioğlu [Ebcioğlu and Nicolau 1989; Moon and Ebcioğlu 1992] and by Fisher [1992]. These are both generalizations of trace scheduling. However, there are numerous difficulties in the engineering of a robust and efficient scheduler of this sort. The challenges in this area of research revolve around finding pragmatic engineering solutions to these problems.

A rather different approach to global acyclic scheduling has been pursued in the IMPACT project [Mahlke, Lin et al. 1992]. An arbitrary, acyclic control flow graph, having a single entry can be handled by this technique. The control flow graph is IF-converted [Allen et al. 1983; Park and Schlansker 1991] so as to eliminate all branches internal to the flow graph. The resulting code, which is similar to a superblock in that it can only be entered at the top but has multiple exits, is termed a *hyperblock*. This is scheduled in much the

same manner as a superblock except that two operations with disjoint predicates (i.e., operations that cannot both be encountered on any single path through the original flow graph) may be scheduled to use the same resources at the same time. After scheduling, reverse IF-conversion is performed to regenerate the control flow graph. Portions of the schedule in which m predicates are active yield 2^m versions of the code.

3.2.1.3. Cyclic Scheduling. As with acyclic flow graphs, instruction-level parallelism in loops is obtained by overlapping the execution of multiple basic blocks. With loops, however, the multiple basic blocks are the multiple iterations of the same piece of code. The most natural extension of the previous global scheduling ideas to loops is to unroll the body of the loop some number of times and to then perform trace scheduling, or some other form of global scheduling, over the unrolled loop body. This approach was suggested by Fisher [Fisher et al. 1981]. A drawback of this approach is that no overlap is sustained across the back edge of the unrolled loop. Fisher and others went on to propose a solution to this problem, which is to continue unrolling and scheduling successive iterations until a repeating pattern is detected in the schedule. The repeating pattern can be rerolled to yield a loop whose body is the repeating schedule. As we shall see, this approach was subsequently pursued by various researchers. In the meantime, loop scheduling moved off in a different direction, which, as is true of most VLIW scheduling work, had its roots in hardware design.

Researchers concerned with the design of pipelined functional units, most notably Davidson and coworkers, had developed the theory of and algorithms for the design of hardware controllers for pipelines to maximize the rate at which functions could be evaluated [Davidson 1971, 1974; Davidson et al. 1975; Patel 1976; Patel and Davidson 1976; Thomas and Davidson 1974]. The issues considered here were quite similar to those faced by individuals programming the innermost loops of signal processing algorithms [Cohen 1978; Kogge 1973, 1974, 1977a, 1977b; Kogge and Stone 1973] on the early peripheral array processors [Floating Point Systems 1979; IBM 1976; Ruggiero and Coryell 1969]. In both cases the objective was to sustain the initiation of successive function evaluations (loop iterations) before prior ones had completed. Since this style of computation is termed pipelining in the hardware context, it was dubbed *software pipelining* in the programming domain [Charlesworth 1981].

Early work in software pipelining consisted of ad hoc hand-coding techniques [Charlesworth 1981; Cohen 1978]. Both the quality of the schedules and the attempts at automating the generation of software pipelined schedules were hampered by the architecture of the early array processors. Nevertheless, Floating Point Systems developed, for the FPS-164 array processor, a compiler that could software pipeline a loop consisting of a single basic block [Touzeau 1984]. Weiss and Smith [1987] note that a limited form of software pipelining was present both in certain hand-coded libraries for the CDC 6600 and also as a capability in the Fortran compiler for the CDC 6600.

The general formulation of the software pipelining process for single basic block loops was stated by Rau and others [Rau and Glaeser 1981; Rau, Glaeser, and Picard 1982] drawing upon and generalizing the theory developed by Davidson and his coworkers on the design of hardware pipelines. This work identified the attributes of a VLIW architecture that make it amenable to software pipelining, most importantly, the availability of conflict-free access to register storage between the output of a functional unit producing a result

and the functional unit that uses that result. This provides freedom in scheduling each operation and is in contrast to the situation in array processors where, due to limited register file bandwidth, achieving peak performance required that a majority of the operations be scheduled to start at the same instant that their predecessor operations completed so that they could pluck their operands right off the result buses.

Rau and others also presented a condition that has to be met by any legal software pipelined schedule—the *modulo constraint*—and derived lower bounds on the rate at which successive iterations of the loop can be started, that is, the *initiation interval* (II). (II is also the length of the software pipelined loop, measured in VLIW instructions, when no loop unrolling is employed.) This lower bound on II, the *minimum initiation interval* (MII), is the maximum of the lower bound due to the resource usage constraints (ResMII) and the lower bound due to the cyclic data dependence constraints caused by recurrences (RecMII). This lower bound is applicable both to vectorizable loops as well as those with arbitrary recurrences and for operation latencies of arbitrary length. A simple, deterministic software pipelining algorithm based on list scheduling, the modulo scheduling algorithm, was shown to achieve the MII, thereby yielding an asymptotically optimal schedule. This algorithm was restricted to DO loops whose body is a single basic block being scheduled on a machine in which each operation has a simple pattern of resource usage, viz., the resource usage of each operation can be abstracted to the use of a single resource for a single cycle (even though the latency of the operation is not restricted to a single cycle). The task of generating an optimal, resource-constrained schedule for loops with arbitrary recurrences is known to be NP-complete [Hsu 1986; Lam 1987] and any practical algorithm must utilize heuristics to guide a generally near-optimal process. These heuristics were only broadly outlined in this work.

Three independent sets of activity took this work and extended it in various directions. The first one was the direct continuation at Cydrome, over the period 1984–88, of the work done by Rau and others [Dehnert et al. 1989; Dehnert and Towle 1993]. In addition to enhancing the modulo scheduling algorithm to handle loops with recurrences and arbitrary acyclic control flow in the loop body, attention was paid to coping with the very complex resource usage patterns that were the result of compromises forced by pragmatic implementation considerations. Complex recurrences and resource usage patterns make it unlikely that a one-pass scheduling algorithm, such as list scheduling, will be able to succeed in finding a near-optimal modulo schedule, even when one exists, and performing an exhaustive search was deemed impractical. Instead, an iterative scheduling algorithm was used that could unschedule and reschedule operations. This iterative algorithm is guided by heuristics based on dynamic slack-based priorities. The initial attempt is to schedule the loop with the II equal to the MII. If unsuccessful, the II is incremented until a modulo schedule is achieved.

Loops with arbitrary acyclic control flow in the loop body are dealt with by performing IF-conversion [Allen et al. 1983] to replace all branching by predicated (guarded) operations. This transformation, which assumes the hardware capability of predicated execution [Rau 1988; Rau et al. 1989], yields a loop with a single basic block that is then amenable to the modulo scheduling algorithm [Dehnert et al. 1989]. A disadvantage of predicated modulo scheduling is that the ResMII must be computed as if all the operations in the body of the loop are executed each iteration, whereas, in reality, only those along one of

the control flow paths are actually executed. As a result, during execution, some fraction of the operations in an instruction are wasted. Likewise, the RecMII is determined by the worst-case dependence chain across all paths through the loop body. Both contribute to a degree of suboptimality that depends on the structure of the loop.

Assuming the existence of hardware to support both predicated execution and speculative execution [Mahlke, Chen et al. 1992], Cydrome's modulo scheduling algorithm has been further extended to handle WHILE loops and loops with conditional exits [Tirumalai et al. 1990]. The problem that such loops pose is that it is not known until late in one iteration whether the next one should be started. This eliminates much of the overlap between successive iterations. The solution is to start iterations speculatively, in effect, by moving operations from one iteration into a prior one. The hardware support makes it possible to avoid observing exceptions from operations that should not have been executed, without overlooking exceptions from nonspeculative operations.

Independently of the Cydrome work, Hsu [1986] proposed a modulo scheduling algorithm for single basic block loops with general recurrences that recognizes each strongly connected class (SCC) of nodes in the cyclic dependence graph as a distinct entity. Once the nodes in all the SCCs have been jointly scheduled at the smallest possible II using a combinatorial search, the nodes in a given SCC may only be rescheduled as a unit and at a time that is displayed by a multiple of II. This rescheduling is performed to enable the remaining nodes that are not part of any SCC to be inserted into the schedule. Hsu also described an II extension technique that can be used to take a legal modulo schedule for one iteration and trivially convert it into a legal modulo schedule for a larger II without performing any scheduling. This works with simple resource usage patterns. With complex patterns a certain amount of rescheduling would be required, but less than starting from scratch.

Lam's algorithm, too, utilizes the SCC structure but lists schedules each SCC separately, ignoring the inter-iteration dependences [Lam 1987, 1988]. Thereafter, an SCC is treated as a single pseudo-operation with a complex resource usage pattern, employing the technique of hierarchical reduction proposed by Wood [1979]. After this hierarchical reduction has been performed, the dependence graph of the computation is acyclic and can be scheduled using modulo scheduling. With an initial value equal to the MII, the II is iteratively increased until a legal modulo schedule is obtained. By determining and fixing the schedule of each SCC in isolation, Lam's algorithm can result in SCCs that cannot be scheduled together at the minimum achievable II.

On the other hand, the application of hierarchical reduction enables Lam's algorithm to cope with loop bodies containing structured control flow graphs without any special hardware support such as predicated execution. Just as with the SCCs, structured constructs such as IF-THEN-ELSE are listed and treated as atomic objects. Each leg of the IF-THEN-ELSE is listed separately and the union of the resource usages represents that of the reduced IF-THEN-ELSE construct. This permits loops with structured flow of control to be modulo scheduled. After modulo scheduling, the hierarchically reduced IF-THEN-ELSE pseudo-operations must be expanded. Each portion of the schedule in which m IF-THEN-ELSE pseudo-operations are active must be expanded into 2^m control flow paths with the appropriate branching and merging between the paths.

Since Lam takes the union of the resource usages in a conditional construct while predicated modulo scheduling takes the sum of the usages, the former approach should yield the smaller MII. However, since Lam separately lists each leg of the conditional creating pseudo-operations with complex resource usage patterns, the II that she actually achieves should deviate from the MII to a greater extent. Warter and others have implemented both techniques and have observed that, on the average, Lam's approach results in smaller MIIs but larger IIs [Warter et al. 1992]. This effect increases for processors with higher issue rates. Warter and others go on to combine the best of both approaches in their enhanced modulo scheduling algorithm. They derive the modulo schedule as if predicated execution were available, except that two operations from the same iteration are allowed to be scheduled on the same resource at the same time if their predicates are mutually exclusive, that is, they cannot both be true. This is equivalent to taking the union of the resource usages. Furthermore, it is applicable to arbitrary, possibly unstructured, acyclic flow graphs in the loop body. After modulo scheduling, the control flow graph is regenerated much as in Lam's approach. Enhanced modulo scheduling results in MIIs that are as small as for hierarchical reduction, but as with predicated modulo scheduling, the achieved II is rarely more than the MII.

Yet another independent stream of activity has been the work of Su and his colleagues [Su et al. 1984; Su et al. 1986]. When limited to loops with a single basic block, Su's URPR algorithm is an ad hoc approximation to modulo scheduling and is susceptible to significant suboptimality when confronted by nonunit latencies and complex resource usage patterns. The essence of the URPR algorithm is to unroll and schedule successive iterations until the first iteration has completed. Next the smallest contiguous set of instructions, which contain at least one instance of each operation in the original loop, is identified. After deleting multiple instances of all operations, this constitutes the software pipelined schedule. This deletion process introduced "holes" in the schedule and the attendant suboptimality. Also, for nonunit latencies, there is no guarantee that the schedule, as constructed, can loop back on itself without padding the schedule out with no-op cycles. This introduces further degradation.

Subsequently, Su extended URPR to the GURPR* algorithm for software pipelining loops containing control flow [Su et al. 1987; Su and Wang 1991a, 1991b]. GURPR* consists of first performing global scheduling on the body of the loop and then using a URPR-like procedure, as if each iteration was IF-converted, to derive the repeating pattern. Finally, as with enhanced modulo scheduling, a control flow graph is regenerated. The shortcomings of URPR are inherited by GURPR*. Warter and others, who have implemented GURPR* within the IMPACT compiler, have found that GURPR* performs significantly worse than hierarchical reduction, predicated modulo scheduling, or enhanced modulo scheduling [Warter et al. 1992].

The idea proposed by Fisher and others of incrementally unrolling and scheduling a loop until the pattern repeats [Fisher et al. 1981] was pursued by Nicolau and his coworkers, assuming unbounded resources, initially for single basic block loops [Aiken and Nicolau 1988b] and then, under the title of perfect pipelining, for multiple basic block loops [Aiken and Nicolau 1988a; Nicolau and Potasman 1990]. The latter was subsequently extended to yield a more realistic algorithm assuming finite resources [Aiken and Nicolau 1991]. For single basic block loops the incremental unrolling yields a growing linear trace, the

expansion of which is terminated once a repeating pattern is observed. In practice there are complications since the various SCCs might proceed at different rates, never yielding a repeating pattern. For multiple basic block loops, the unrolling yields a growing tree of schedules, each leaf of which spawns two further leaves when a conditional branch is scheduled. A new leaf is not spawned if the (infinite) tree, of which it would be the root, is identical to another (infinite) tree (of which it might be the leaf) whose root has already been generated.

This approach addresses a shortcoming of all the previously mentioned approaches to software pipelining multiple basic block loops. In general, both RecMII and ResMII are dependent upon the specific control flow path followed in each iteration. Whereas the previous approaches had to use a single, constant, conservative value for each one of these lower bounds, the unrolling approach is able to take advantage of the branch history of previous iterations in deriving the schedule for the current one. However, there are some drawbacks as well. One handicap that such unrolling schemes have is a lack of control over the greediness of the process of initiating iterations. Starting successive iterations as soon as possible, rather than at a measured rate that is in balance with the completion rate, cannot reduce the average initiation interval but can increase the time to enter the repeating pattern and the length of the repeating pattern. Both contribute to longer compilation times and larger code size. A second problem with unrolling schemes lies in their implementation; recognizing that one has arrived at a previously visited state, to which one can wrap back instead of further expanding the search tree, is quite complicated, especially in the context of finite resources, nonunit latencies, and complex resource usage patterns.

The cyclic scheduling algorithm developed by the IBM VLIW research project [Ebcioğlu and Nakatani 1989; Gasperoni 1989; Moon and Ebcioğlu 1992; Nakatani and Ebcioğlu 1990] might represent a good compromise between the ideal and the practical. Stripped to the essentials, this algorithm applies a cut set, termed a *fence*, to the cyclic graph, which yields an acyclic graph. This reduces the problem to that of scheduling a general, acyclic graph—a simpler problem. Once this is done the fence is moved and the acyclic scheduling is repeated. As this process is repeated, all the cycles in the control flow graph acquire increasingly tight schedules. The acyclic scheduling algorithm used by Ebcioğlu and others is a resource-constrained version of percolation scheduling [Ebcioğlu and Nicolau 1989; Moon and Ebcioğlu 1992].

Software pipelining was also implemented in the compiler for the product line marketed by another minisupercomputer company, Culler Scientific. Unfortunately, we do not believe that any publication describing their implementation of software pipelining exists. Quite recently, software pipelining has been implemented in the compilers for HP's PA-RISC line of computers [Ramakrishnan 1992].

3.2.1.4. Scheduling for RISC and Superscalar Processors. Seemingly conventional scalar processors can sometimes benefit from scheduling techniques. This is due to small amounts of ILP in the form of, for instance, branch delay slots and shallow pipelines. Scheduling for such processors, whether RISC or CISC, has generally been less ambitious and more ad hoc than that for VLIW processors [Auslander and Hopkins 1982; Gross and Hennessy 1982; Hennessy and Gross 1983; Hsu 1987; McFarling and Hennessy 1986]. This was a

direct consequence of the lack of parallelism in those machines and the corresponding lack of opportunity for the scheduler to make a big difference. Furthermore, the limited number of registers in those architectures made the use of aggressive scheduling rather unattractive. As a result, scheduling was viewed as rather peripheral to the compilation process, in contrast to the central position it occupied for VLIW processors and, to a lesser extent, for more highly pipelined processors [Rymarczyk 1982; Sites 1978; Weiss and Smith 1987]. Now, with superscalar processors growing in popularity, the importance of scheduling, as a core part of the compiler, is better appreciated and a good deal of activity has begun in this area [Bernstein and Rodeh 1991; Bernstein et al. 1991; Golumbic and Rainish 1990; Jain 1991; Smotherman et al. 1991], unfortunately, sometimes unaware of the large body of literature that already exists.

3.2.2. Register Allocation. In conventional, sequential processors, instruction scheduling is not an issue. The program's execution time is barely affected by the order of the instruction, only by the number of instructions. Accordingly, the emphasis of the code generator is on generating the minimum number of instructions and using as few registers as possible [Aho and Johnson 1976; Aho et al. 1977a, 1977b; Bruno and Sethi 1976; Sethi 1975; Sethi and Ullman 1970]. However, in the context of pipelined or multiple-issue processors, where instruction scheduling is important, the issue of the phase-ordering between it and register allocation has been a topic of much debate. There are advocates both for performing register allocation before scheduling [Gibbons and Muchnick 1986; Hennessy and Gross 1983; Jain 1991] as well as for performing it after scheduling [Auslander and Hopkins 1982; Chang, Lavery, and Hwu 1991; Goodman and Hsu 1988; Warren 1990]. Each phase-ordering has its advantages and neither one is completely satisfactory.

The most important argument in favor of performing register allocation first is that whereas a better schedule may be desirable, code that requires more registers than are available is just unacceptable. Clearly, achieving a successful register allocation must supersede the objective of constructing a better schedule. The drawback of performing scheduling first, oblivious of the register allocation, is that shorter schedules tend to yield greater register pressure. If a viable allocation cannot be found, spill code must be inserted. At this point, in the case of a statically scheduled processor, the schedule just constructed may no longer be correct. Even if it is, it may be far from the best one possible, for either a VLIW or superscalar machine, since the schedule was built without the spill code in mind. In machines whose load latency is far greater than that of the other operations, the time penalty of the spill code may far exceed the benefits of the better schedule obtained by performing scheduling first.

Historically, the merit of performing register allocation first was that processors had little instruction-level parallelism and few registers, so whereas there was much to be lost by a poor register allocation, there was little to be gained by good scheduling. It was customary, therefore, to perform register allocation first, for instance using graph coloring [Chaitin 1982; Chow and Hennessy 1984, 1990] followed by a postpass scheduling step that considered individual basic blocks [Gibbons and Muchnick 1986; Hennessy and Gross 1983].

From the viewpoint of instruction-level parallel machines, the major problem with performing register allocation first is that it introduces antidependences and output dependences that can constrain parallelism and the ability to construct a good schedule. To some extent

this is inevitable; the theoretically optimal combination of schedule and allocation might contain additional arcs due to the allocation. The real concern is that, when allocation is done first, an excessive number of ill-advised and unnecessary arcs might be introduced due to the insensitivity of the register allocator to the scheduling task. On pipelined machines, whose cache access time is as short as or shorter than the functional unit latencies, the benefits of a schedule unconstrained by register allocation may outweigh the penalties of the resulting spill code.

Scheduling prior to register allocation, known as prepass scheduling, was used in the PL.8 compiler [Auslander and Hopkins 1982]. In evolving this compiler to become the compiler for the superscalar IBM RISC System/6000, the suboptimality of inserting spill code after the creation of the schedule became clear and a second, postpass scheduling step was added after the register allocation [Warren 1990]. During the postpass the scheduler honors all the dependences caused by the register allocation, which in turn was aware of the preferred instruction schedule provided by the prepass scheduler. The IMPACT project at the University of Illinois has demonstrated the effectiveness of this strategy for multiple-issue processors [Chang, Lavery, and Hwu 1991]. Instead of employing the graph coloring paradigm, Hendren and others make use of the richer information present in interval graphs, which are a direct temporal representation of the span of the lifetimes [Hendren et al. 1992]. This assumes that the schedule or, at least, the instruction order has already been determined and that a postpass scheduling step will follow.

Irrespective of which one goes first, a shortcoming of all strategies discussed so far is that the first phase makes its decisions with no consideration of their impact on the subsequent phase. Goodman and Hsu [1988] have addressed this problem by developing two algorithms—one, a scheduler that attempts to keep the register pressure below a limit provided to it, and the second, a register allocation algorithm that is sensitive to its effect on the critical path length of the DAG and thus to the effect on the eventual schedule.

For any piece of code on a given processor, there is some optimal schedule for which register allocation is possible. Scheduling twice, once before and then after register allocation, is an approximation of achieving this ideal. Simultaneous scheduling and register allocation is another strategy for attempting to find a near-optimal schedule and register allocation. Simultaneous scheduling and register allocation is currently understood only in the context of acyclic code, specifically, a single basic block or a linear trace of basic blocks. The essence of the idea is that each time an operation is scheduled, an available register is allocated to hold the result. Also, if this operation constitutes the last use of the contents of one of the source registers, that register is made available once again for subsequent allocation. When no register is available to receive the result of the operation being scheduled, a register must be spilled. The register holding the datum whose use is furthest away in the future is spilled. This approach was used in the FPS-164 compiler at the level of individual basic blocks [Touzeau 1984] as well as across entire traces [Ellis 1985; Freudenberg and Ruttenberg 1992; Lowney et al. 1993]. An important concept developed by the ELI project at Yale and by Multiflow was that of performing hierarchical, profile-driven, integrated global scheduling and register allocation. Traces are picked in decreasing order of frequency and integrated scheduling and allocation are performed on each. The scheduling and allocation decisions made for traces that have been processed form constraints on the corresponding decisions for the remaining code. This is a far more systematic approach

than other ad hoc, priority-based schemes with the same objective. A syntax-based hierarchical approach to global register allocation has been suggested by Callahan and Koblenz [1991].

If a loop is unrolled some number of times and then treated as a linear trace of basic blocks [Fisher et al. 1981], simultaneous trace scheduling and register allocation can be accomplished, but with some loss of performance due to the emptying of pipelines across the back edge. In the case of modulo scheduling, which avoids this performance penalty, no approach has yet been advanced for simultaneous register allocation. Since doing register allocation in advance is unacceptably constraining on the schedule, it must be performed following modulo scheduling. A unique situation encountered with modulo scheduled loops is that the lifetimes are often much longer than the initiation interval. Normally, this would result in a value being overwritten before its last use has occurred. One solution is to unroll the kernel of a modulo scheduled loop a sufficient number of times to ensure that no lifetime is longer than the length of the replicated kernel [Lam 1987, 1988]. This is known as *modulo variable expansion*. In addition to techniques such as graph coloring, the heuristics proposed by Hendren and others [1992] and by Rau and others [1992] may be applied after modulo variable expansion. The other solution for register allocation is to assume the dynamic register renaming provided by the rotating register capability of the Cydra 5. The entity that the register allocator works with are vector lifetimes, that is, the entire sequence of (scalar) lifetimes defined by a particular operation over the execution of the loop [Dehnert and Towle 1993; Dehnert et al. 1989; Rau et al. 1992]. Lower bounds on the number of registers needed for a modulo scheduled loop have been developed by Mangione-Smith and others [1992]. The strategy for recovering from a situation, in which no allocation can be found for the software pipelined loop, is not well understood. Some options have been outlined [Rau et al. 1992], but their detailed implementation, effectiveness, and relative merits have as yet to be investigated.

3.2.3. Other ILP Compiler Topics. Although scheduling and register allocation are at the heart of ILP compilation, a number of other analyses, optimizations, and transformations are crucial to the generation of high-quality code. Currently, schedulers treat a procedure call as a barrier to code motion. Thus, in-lining of intrinsics and user procedures is very important in the high frequency portions of the program [Dehnert and Towle 1993; Linn 1988; Lowney et al. 1993].

Certain loop-oriented analyses and optimizations are specific to modulo scheduling. IF-conversion and the appropriate placement of predicate-setting operations are needed to modulo schedule loops with control flow [Allen et al. 1983; Dehnert and Towle 1993; Dehnert et al. 1989; Park and Schlansker 1991]. The elimination of subscripted loads and stores that are redundant across multiple iterations of a loop can have a significant effect upon both the ResMII and the RecMII [Callahan et al. 1990; Dehnert and Towle 1993; Rau 1992]. This is important for trace scheduling unrolled loops as well [Lowney et al. 1993]. Recurrence back-substitution, and other transformations that reduce the RecMII have a major effect on the performance of all software pipelined loops [Dehnert and Towle 1993]. Most of these transformations and analyses are facilitated by the dynamic single-assignment representation for inner loops [Dehnert and Towle 1993; Rau 1992].

On machines with multiple, identical clusters, such as the Multiflow Trace machines, it is necessary to decide which part of the computation will go on each cluster. This is a

nontrivial task; whereas increased parallelism argues in favor of spreading the computation over the clusters, this also introduces intercluster move operations into the computation, whose latency can degrade performance if the partitioning of the computation across clusters is not done carefully. An algorithm for performing this partitioning was developed by Ellis [1986] and was incorporated into the Multiflow compiler [Lowney et al. 1993].

An issue of central importance to all ILP compilation is the disambiguation of memory references, that is, deciding whether two memory references definitely are to the same memory location or definitely are not. Known as dependence analysis, this has become a very well developed topic in the area of vector computing over the past twenty years [Zima and Chapman 1990]. For vector computers the compiler is attempting to prove that two references in different iterations are *not* to the same location. No benefit is derived if it is determined that they *are* to the same location since such loops cannot be vectorized. Consequently, the nature of the analysis, especially in the context of loops containing conditional branching, has been approximate. This is a shortcoming from the point of view of ILP processors that can benefit both if the two references are or are not to the same location. A more precise analysis than dependence analysis, involving data flow analysis, is required. Also, with ILP processors, memory disambiguation is important outside of loops as well as within them. Memory disambiguation within traces was studied in the ELI project [Ellis 1985; Nicolau 1984] and was implemented in the Multiflow compiler [Lowney et al. 1993]. Memory disambiguation, in the context of innermost loops, was implemented in the Cydra 5 compiler [Dehnert and Towle 1993; Rau 1992] and was studied by Callahan and Koblenz [1991].

4. Available ILP

4.1. Limit Studies and Their Shortcomings

Many experimenters have attempted to measure the maximum parallelism available in programs. The goal of such limit studies is to

throw away all considerations of hardware and compiler practicality and measure the greatest possible amount of ILP inherent in a program.

Limit studies are simple enough to describe: Take an execution trace of the program, and build a data precedence graph on the operations, eliminating false antidependences caused by the write-after-read usage of a register or other piece of hardware storage. The length in cycles of the serial execution of the trace gives the serial execution time on hardware with the given latencies. The length in cycles of the critical path though the data dependence graph gives the shortest possible execution time. The quotient of these two is the available speedup. (In practice, an execution trace is not always gathered. Instead, the executed stream is processed as the code runs, greatly reducing the computation or storage required, or both.)

These are indeed maximum parallelism measures in some sense, but they have a critical shortcoming that causes them to miss accomplishing their stated goal; they do not consider transformations that a compiler might make to enhance ILP. Although we mostly mean

transformations of a yet-unknown nature that researchers may develop in the future, even current state-of-the-art transformations are rarely reflected in limit studies. Thus we have had, in recent years, the anomalies of researchers stating an “upper limit” on available parallelism in programs that is lower than what has already been accomplished with those same programs, or of new results that show the maximum available parallelism to be significantly higher than it was a few years ago, before a new set of code transformations was considered.

There is a somewhat fatuous argument that demonstrates just how imprecise limit studies must be; recalling that infinite hardware is available, we can replace computations in the code with table lookups. In each case we will replace a longer—perhaps very long—computation with one that takes a single step. While this is obviously impractical for most computations with operands that span the (finite, but large) range of integers or floating point numbers representable on a system, it is only impractical in the very sense in which practicality is to be discarded in limit studies. And even on practicality grounds, one cannot dismiss this argument completely; in a sense it really does capture what is wrong with these experiments. There are many instances of transformations, some done by hand, others automatically, that reduce to this concept. Arithmetic and transcendental functions are often sped up significantly by the carefully selected use of table lookups at critical parts of the computation. Modern compilers can often replace a nested set of IF-THEN tests with a single lookup in which hardware does an indirect jump through a lookup table. Limit studies have no way of capturing these transformations, the effect of which could be a large improvement in available ILP.

Even in current practice the effect of ignoring sophisticated compiling is extreme. Transformations such as tree height reduction, loop conditioning, loop exchange, and so forth can have a huge effect on the parallelism available in code. A greater unknown is the research future of data structure selection to improve ILP. A simple example can show this effect. The following code finds the maximum element of a linked list of data:

```
this_ptr = head_ptr;
max_so_far = most_neg_number;
while this_ptr {
    if this_ptr.data > max_so_far
        then max_so_far = this_ptr.data;
    this_ptr = this_ptr.next }
```

From simple observation the list of elements chained from head_ptr cannot be circular. If the compiler had judged it worthwhile, it could have stored these elements in an array and done the comparisons pairwise, in parallel, without having to chase the pointers linearly. This example is not as farfetched as it might seem. Vectorization took 20 years to go from the ability to recognize the simplest loop to the sophisticated vectorizers we have today. There has been virtually no work done on compiler transformations to enhance ILP.

Limit studies, then, are in some sense finding the maximum parallelism available, but in other ways are finding the minimum. In these senses they find the maximum parallelism:

- Disambiguation can be done perfectly, well beyond what is practical.
- There are infinitely many functional units available.
- There are infinitely many registers available.
- Rejoins can be completely unwound.

In other senses, they represent a minimum, or an existence proof that at least a certain amount of parallelism exists, since potentially important processes have been left out:

- Compiler transformations to enhance ILP have not been done.
- Intermediate code generation techniques that boost ILP have not been done.

Perhaps it is more accurate to say that a limit study shows that the maximum parallelism available, in the absence of practicality considerations, is *at least* the amount measured.

4.1.1. Early Experiments. The very first ILP limit studies demonstrated the effect we wrote of above: The experimenters' view of the techniques by which one could find parallelism was limited to the current state of the art, and the experimenters missed a technique that is now known to provide most of the available ILP, the motion of operations between basic blocks of code. Experiments done by Tjaden and Flynn [1970] and by Foster and Riseman [1972] (and, anecdotally, elsewhere) found that there was only a small amount (about a factor of two to three) of improvement due to ILP available in real programs. This was dubbed the *Flynn bottleneck*. By all accounts, these pessimistic and, in a sense, erroneous experiments had a tremendous dampening effect on the progress of ILP research. The experiments were only erroneous in the sense of missing improvements; certainly they did correctly what they said they did.

Interestingly, one of the research teams doing these experiments saw that under the hypothesis of free and infinite hardware, one would not necessarily have to stop finding ILP at basic block boundaries. In a companion paper to the one mentioned above, Riseman and Foster [1972] put forward a hardware-intensive solution to the problem of doing operations speculatively: They measured what would happen if one used duplicate hardware at conditional jumps, and disregarded the one that went in the wrong direction. They found a far larger amount of parallelism: Indeed, they found more than an order of magnitude more than they could when branches were a barrier. Some of the programs they measured could achieve arbitrarily large amounts of parallelism, depending only on data set size. But in an otherwise insightful and visionary piece of work, the researchers lost sight of the fact that they were doing a limit study, and in their tone and abstract emphasized how impractical it would be to implement the hardware scheme they had suggested. (They found that to get a factor-of-ten ILP speedup, one had to be prepared to cope with 16 unresolved branches at the worst point of a typical program. Their scheme would require, then, 2^{16} sets of hardware to do so. Today, as described in most of the papers in this issue, we try to get much of the benefit of the same parallelism without the hardware cost by doing code motions that move operations between blocks and having the code generator make sure that the correct computation is ultimately done once the branches settle.)

4.1.2. Contemporary Experiments. We know of no other ILP limit studies published between then and the 1980s. In 1981 Nicolau and Fisher [1981, 1984] used some of the

apparatuses being developed for the Yale Bulldog compiler to repeat the experiment done by Riseman and Foster, and found virtually the same results.

In the late 1980s architects began to look at superscalar microprocessors and again started a series of limit studies. Interestingly, the most notorious of these [Jouppi and Wall 1989] again neglected the possibility of code motions between blocks. Unsurprisingly, the Flynn bottleneck appeared again, and only the factor of 2-3 parallelism found earlier was found. Two years later Wall [1991] did the most thorough limit study to date and accounted for speculative execution, memory disambiguation, and other factors. He built an elaborate model and published available ILP speedup under a great many scenarios, yielding a wealth of valuable data but no simple answers. The various scenarios allow one to try to bracket what really might be practical in the near future, but are subject to quite a bit of interpretation. In examining the various scenarios presented, we find that settings that a sophisticated compiler might approach during the coming decade could yield speedups ranging from 7 to 60 on the sample programs, which are taken from the SPEC suite and other standard benchmarks. (It is worth noting that Wall himself is much more pessimistic. In the same results he sees an average ceiling of about 5, and the near impossibility of attaining even that much.) Lam and Wilson [1992] did an experiment to measure the effects of different methods of eliminating control flow barriers to parallelism. When their model agreed with Wall's, their results were similar. Butler and Patt [Butler et al. 1991] considered models with a large variety of numbers of functional units and found that with good branch prediction schemes and speculative execution, a wide range of speedup was available.

4.2. Experiments That Measure Attained Parallelism

In contrast to the limit studies, some people have built real or simulated ILP systems and have measured their speedup against real or simulated nonparallel systems. When simulated systems have been involved, they have been relatively realistic systems, or systems that the researchers have argued would abstract the essence of realistic systems in such a way that the system realities should not lower the attained parallelism. Thus the experiments represent something closer to true lower bounds on available parallelism.

Ellis [1986] used the Bulldog compiler to generate code for a hypothetical machine. His model was unrealistic in several aspects, most notably the memory system, but realistic implementations should have little difficulty exploiting the parallelism he found. Ellis measured the speedups obtained on 12 small scientific programs for both a "realistic" machine (corresponding to one under design at Yale) and an "ideal" machine, with limitless hardware and single-cycle functional units. He found speedups ranging from no speedup to 7.6 times speedup for the real model, and a range of 2.7 to 48.3 for the ideal model.

In this issue there are three papers that add to our understanding of the performance of ILP systems. The paper by Hwu and others [1993] considers the effect of a realistic compiler that uses superblock scheduling. Lowney and others [1993] and Schuette and Shen [1993] compare the performance of the Multiflow TRACE 14/300 with current microprocessors from MIPS and IBM, respectively.

Fewer studies have been done to measure the attained performance of software pipelining. Warter and others [1992] consider a set of 30 *doall* loops with branches found in the Perfect

and SPEC benchmark sets. Relative to a single-issue machine without modulo scheduling, they find a 6-time speedup on a hypothetical 4-issue machine and a 10-time speedup on a hypothetical 8-issue machine. Lee and others [1993] combined superblock scheduling and software pipelining for a machine capable of issuing up to seven operations per cycle. On a mix of loop-intensive (e.g., LINPACK) and "scalar" (e.g., Spice) codes, they found an average of one to four operations issued per cycle, with two to seven operations in flight.

5. An Introduction to This Special Issue

In this special issue of *The Journal of Supercomputing* we have attempted to capture the most significant work that took place during the 1980s in the area of instruction-level parallel processing. The intent is to document both the theory and the practice of ILP computing. Consequently, our emphasis is on projects that resulted in implementations of serious scope, since it is this reduction to practice that exposes the true merit and the real problems of ideas that sound good on paper.

During the 1980s the bulk of the advances in ILP occurred in the form of VLIW processing, and this special issue reflects it with papers on Multiflow's Trace family and on Cydrome's Cydra 5. The paper by Lowney and others [1993] provides an overview of the Trace hardware and an in-depth discussion of the compiler. The paper by Schuette and Shen [1993] reports on an evaluation performed by the authors of the TRACE 14/300 and a comparison of it to the superscalar IBM RS/6000. The Cydra 5 effort is documented by two papers: one by Beck, Yen, and Anderson [1993] on the Cydra 5 architecture and hardware implementation, and the other by Dehnert and Towle [1993] on the Cydra 5 compiler. (While reading the descriptions of these large and bulky minisupercomputers, it is worthwhile to bear in mind that they could easily fit on a single chip in the near future!) The only important superscalar product of the 1980s was Astronautics' ZS-1 minisupercomputer. Although we wanted to include a paper on it in this special issue, that did not come to pass. The paper by Hwu and others [1993] reports on IMPACT, the most thorough implementation of an ILP compiler that has occurred in academia.

Notes

1. The first machines of this type that were built in the 1960s were referred to as *look-ahead* processors. Subsequently, machines that performed out-of-order execution, while issuing multiple operations per cycle, came to be termed *superscalar* processors. Since look-ahead processors are only quantitatively different from superscalar processors, we shall drop the distinction and refer to them, too, as superscalar processors.
2. We shall consistently refer to this code generation activity as *scheduling*.

References

- Acosta, R.D., Kjelstrup, J., and Torng, H.C. 1986. An instruction issuing approach to enhancing performance in multiple function unit processors. *IEEE Trans. Comps.*, C-35, 9 (Sept.): 815-828.
- Adam, T.L., Chandy, K.M., and Dickson, J.R. 1974. A comparison of list schedules for parallel processing systems. *CACM*, 17, 12 (Dec.): 685-690.
- Advanced Micro Devices. 1989. *Am29000 Users Manual*. Pub. no. 10620B, Advanced Micro Devices, Sunnyvale, Calif.
- Agerwala, T. 1976. Microprogram optimization: A survey. *IEEE Trans. Comps.*, C-25, 10 (Oct.): 962-973.
- Agerwala, T., and Cocke, J. 1987. High performance reduced instruction set processors. Tech. rept. RC12434 (#55845), IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y.
- Aho, A.V., and Johnson, S.C. 1976. Optimal code generation for expression trees. *JACM*, 23 3 (July): 488-501.
- Aho, A.V., Johnson, S.C., and Ullman, J.D. 1977a. Code generation for expressions with common subexpressions. *JACM*, 24, 1 (Jan.): 146-160.
- Aho, A.V., Johnson, S.C., and Ullman, J.D. 1977b. Code generation for machines with multiregister operations. In *Proc., Fourth ACM Symp. on Principles of Programming Languages*, pp. 21-28.
- Aiken, A., and Nicolau, A. 1988a. Optimal loop parallelization. In *Proc., SIGPLAN'88 Conf. on Programming Language Design and Implementation* (Atlanta, June), pp. 308-317.
- Aiken, A., and Nicolau, A. 1988b. Perfect pipelining: A new loop parallelization technique. In *Proc., 1988 European Symp. on Programming*, Springer Verlag, New York, pp. 221-235.
- Aiken, A., and Nicolau, A. 1991. A realistic resource-constrained software pipelining algorithm. In *Advances in Languages and Compilers for Parallel Processing* (A. Nicolau, D. Gelernter, T. Gross, and D. Padua, eds.), Pitman/MIT Press, London, pp. 274-290.
- Allen, J.R., Kennedy, K., Porterfield, C., and Warren, J. 1983. Conversion of control dependence to data dependence. In *Proc., Tenth Annual ACM Symp. on Principles of Programming Languages* (Jan.): pp. 177-189.
- Anderson D.W., Sparacio, F.J., and Tomasulo, R.M. 1967. The System/360 Model 91: Machine philosophy and instruction handling. *IBM J. Res. and Dev.*, 11, 1 (Jan.): 8-24.
- Apollo Computer. 1988. *The Series 10000 Personal Supercomputer: Inside a New Architecture*. Publication no. 002402-007 2-88, Apollo Computer, Inc., Chelmsford, Mass.
- Arvind and Gostelow, K. 1982. The U-interpreter. *Computer*, 15, 2 (Feb.): 12-49.
- Arvind and Kathail, V. 1981. A multiple processor dataflow machine that supports generalised procedures. In *Proc., Eighth Annual Symp. on Computer Architecture* (May): pp. 291-302.
- Auslander, M., and Hopkins, M. 1982. An overview of the PL-8 compiler. In *Proc., ACM SIGPLAN Symp. on Compiler Construction* (Boston, June), pp. 22-31.
- Bahr, R., Ciavaglia, S., Flahive, B., Kline, M., Mageau, P., and Nickel, D. 1991. The DNI0000TX: A new high-performance PRISM processor. In *Proc., COMPCON '91*, pp. 90-95.
- Baker, K.R. 1974. *Introduction to Sequencing and Scheduling*. John Wiley, New York.
- Beck, G.R., Yen, D.W.L., and Anderson T.L. 1993. The Cydra 5 minisupercomputer: Architecture and implementation. *The J. Supercomputing*, 7, 1/2: 143-180.
- Bell, C.G., and Newell, A. 1971. *Computer Structures: Readings and Examples*. McGraw-Hill, New York.
- Bernstein, D., and Rodeh, M. 1991. Global instruction scheduling for superscalar machines. In *Proc., SIGPLAN '91 Conf. on Programming Language Design and Implementation* (June), pp. 241-255.
- Bernstein, D., Cohen, D., and Krawczyk, H. 1991. Code duplication: An assist for global instruction scheduling. In *Proc., 24th Annual Internat. Symp. on Microarchitecture* (Albuquerque, N.Mex.), pp. 103-113.
- Blanck, G., and Krueger, S. 1992. The SuperSPARC™ microprocessor. In *Proc., COMPCON '92*, pp. 136-141.
- Bloch, E. 1959. The engineering design of the STRETCH computer. In *Proc., Eastern Joint Computer Conf.*, pp. 48-59.
- Bruno, J.L., and Sethi, R. 1976. Code generation for a one-register machine. *JACM*, 23, 3 (July): 502-510.
- Buchholz, W., ed. 1962. *Planning a Computer System: Project Stretch*. McGraw-Hill, New York.
- Butler, M., Yeh, T., Patt, Y., Alsup, M., Scales, H., and Shebanow, M. 1991. Single instruction stream parallelism is greater than two. In *Proc., Eighteenth Annual Internat. Symp. on Computer Architecture* (Toronto), pp. 276-286.
- Callahan, D., and Koblenz, B. 1991. Register allocation via hierarchical graph coloring. In *Proc., SIGPLAN '91 Conf. on Programming Language Design and Implementation* (Toronto, June), pp. 192-203.
- Callahan, D., Carr, S., and Kennedy, K. 1990. Improving register allocation for subscripted variables. In *Proc., ACM SIGPLAN '90 Conf. on Programming Language Design and Implementation*, (White Plains, N.Y., June), pp. 53-65.
- Carpenter, B.E., and Doran, R.W., eds. 1986. *A.M. Turing's ACE Report of 1946 and Other Papers*. MIT Press, Cambridge, Mass.
- Chaitin, G.J. 1982. Register allocation and spilling via graph coloring. In *Proc., ACM SIGPLAN Symp. on Compiler Construction* (Boston, June), pp. 98-105.
- Chang, P.P., and Hwu, W.W. 1988. Trace selection for compiling large C application programs to microcode. In *Proc., 21st Annual Workshop on Microprogramming and Microarchitectures* (San Diego, Nov.), pp. 21-29.

- Chang, P.P., and Hwu, W.W. 1992. Profile-guided automatic inline expansion for C programs. *Software—Practice and Experience*, 22, 5 (May): 349–376.
- Chang, P.P., Lavery, D.M., and Hwu, W.W. 1991. The importance of preprocess code scheduling for superscalar and superpipelined processors. Tech. Rept. no. CRHC-91-18, Center for Reliable and High-Performance Computing, Univ. of Ill., Urbana-Champaign, Ill.
- Chang, P.P., Mahlke, S.A., Chen, W.Y., Warter, N.J., and Hwu, W.W. 1991. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proc., 18th Annual Internat. Symp. on Computer Architecture* (Toronto, May), pp. 266–275.
- Charlesworth, A.E. 1981. An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family. *Computer*, 14, 9: 18–27.
- Chen, T.C. 1971. Parallelism, pipelining, and computer efficiency. *Computer Design*, 10, 1 (Jan.): 69–74.
- Chen, T.C. 1975. Overlap and pipeline processing. In *Introduction to Computer Architecture* (H.S. Stone, ed.), Science Research Associates, Chicago, pp. 375–431.
- Chow, F., and Hennessy, J. 1984. Register allocation by priority-based coloring. In *Proc., ACM SIGPLAN Symp. on Compiler Construction* (Montreal, June), pp. 222–232.
- Chow, F.C., and Hennessy, J.L. 1990. The priority-based coloring approach to register allocation. *ACM Trans. Programming Languages and Systems*, 12 (Oct.): 501–536.
- Coffman, J.R., ed. 1976. *Computer and Job-Shop Scheduling Theory*. John Wiley, New York.
- Coffman, E.G., and Graham, R.L. 1972. Optimal scheduling for two processor systems. *Acta Informatica*, 1, 3: 200–213.
- Cohen, D. 1978. A methodology for programming a pipeline array processor. In *Proc., 11th Annual Microprogramming Workshop* (Asilomar, Calif., Nov.), pp. 82–89.
- Colwell, R.P., Nix, R.P., O'Donnell, J.J., Papworth, D.B., and Rodman, P.K. 1988. A VLIW architecture for a trace scheduling compiler. *IEEE Trans. Comps.*, C-37, 8 (Aug.): 967–979.
- Colwell, R.P., Hall, W.E., Joshi, C.S., Papworth, D.B., Rodman, P.K., and Tornes, J.E. 1990. Architecture and implementation of VLIW supercomputer. In *Proc., Supercomputing '90* (Nov.), pp. 910–919.
- Cotten, L.W. 1965. Circuit implementation of high-speed pipeline systems. In *Proc., AFIPS Fall Joint Computing Conf.*, pp. 489–504.
- Cotten, L.W. 1969. Maximum-rate pipeline systems. In *Proc., AFIPS Spring Joint Computing Conf.*, 581–586.
- Danelutto, M., and Vanneschi, M. 1990. VLIW in-the-large: A model for fine grain parallelism exploitation of distributed memory multiprocessors. In *Proc., 23rd Annual Workshop on Microprogramming and Microarchitecture* (Nov.), pp. 7–16.
- Dasgupta, S., and Tartar, J. 1976. The identification of maximal parallelism in straight-line microprograms. *IEEE Trans. Comps.*, C-25, 10 (Oct.): 986–991.
- Davidson, E.S. 1971. The design and control of pipelined function generators. In *Proc., 1971 Internat. IEEE Conf. on Systems, Networks, and Computers* (Oaxtepec, Mexico, Jan.), pp. 19–21.
- Davidson, E.S. 1974. Scheduling for pipelined processors. In *Proc., 7th Hawaii Conf. on Systems Sciences*, pp. 58–60.
- Davidson, S., Landskov, D., Shriver, B.D., and Mallett, P.W. 1981. Some experiments in local microcode compaction for horizontal machines. *IEEE Trans. Comps.*, C-30, 7: 460–477.
- Davidson, E.S., Shar, L.E., Thomas, A.T., and Patel, J.H. 1975. Effective control for pipelined computers. In *Proc., COMPCON '90* (San Francisco, Feb.), pp. 181–184.
- Dehnert, J.C., and Towle, R.A. 1993. Compiling for the Cydra 5. *The J. Supercomputing*, 7, 1/2: 181–227.
- Dehnert, J.C., Hsu, P.Y.-T., and Bratt, J.P. 1989. Overlapped loop support in the Cydra 5. In *Proc., Third Internat. Conf. on Architectural Support for Programming Languages and Operating Systems* (Boston, Apr.), pp. 26–38.
- DeLano, E., Walker, W., Yetter, J., and Forsyth, M. 1992. A high speed superscalar PA-RISC processor. In *Proc., COMPCON '92* (Feb.), pp. 116–121.
- DeWitt, D.J. 1975. A control word model for detecting conflicts between microprograms. In *Proc., 8th Annual Workshop on Microprogramming* (Chicago, Sept.), pp. 6–12.
- Diefendorff, K., and Allen, M. 1992. Organization of the Motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, 12, 2 (Apr.): 40–63.
- Dongarra, J.J. 1986. A survey of high performance computers. In *Proc., COMPCON '86* (Mar.), pp. 8–11.
- Dwyer, H., and Törng, H.C. 1992. An out-of-order superscalar processor with speculative execution and fast, precise interrupts. In *Proc., 25th Annual Internat. Symp. on Microarchitecture* (Portland, Ore., Dec.), pp. 272–281.
- Ebcioğlu, K. 1988. Some design ideas for a VLIW architecture for sequential-natured software. In *Parallel Processing (Proc., IFIP WG 10.3 Working Conf. on Parallel Processing, Pisa, Italy)* (M. Cosnard, M.H. Barton, and M. Vanneschi, eds.), North-Holland, pp. 3–21.
- Ebcioğlu, K., and Nakatani, T. 1989. A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture. In *Languages and Compilers for Parallel Computing* (D. Gelernter, A. Nicolau, and D. Padua, eds.), Pitman/MIT Press, London, pp. 213–229.
- Ebcioğlu, K., and Nicolau, A. 1989. A global resource-constrained parallelization technique. In *Proc., 3rd Internat. Conf. on Supercomputing* (Crete, Greece, June), pp. 154–163.
- Eckert, J.P., Chu, J.C., Tonik, A.B., and Schmitt, W.F. 1959. Design of UNIVAC-LARC System: I. In *Proc., Eastern Joint Computer Conf.*, pp. 59–65.
- Ellis, J.R. 1986. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, Cambridge, Mass.
- Fawcett, B.K. 1975. Maximal clocking rates for pipelined digital systems. M.S. thesis, Univ. of Ill., Urbana-Champaign, Ill.
- Fernandez, E.B., and Bussel, B. 1973. Bounds on the number of processors and time for multiprocessor optimal schedule. *IEEE Trans. Comps.*, C-22, 8 (Aug.): 745–751.
- Fisher, J.A. 1979. The optimization of horizontal microcode within and beyond basic blocks: An application of processor scheduling with resources. Ph.D. thesis, New York Univ., New York.
- Fisher, J.A. 1980. 2^N -way jump microinstruction hardware and an effective instruction binding method. In *Proc., 13th Annual Workshop on Microprogramming* (Colorado Springs, Colo., Nov.), pp. 64–75.
- Fisher, J.A. 1981. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comps.*, C-30, 7 (July): 478–490.
- Fisher, J.A. 1983. Very long instruction word architectures and the ELI-512. In *Proc., Tenth Annual Internat. Symp. on Computer Architecture* (Stockholm, June), pp. 140–150.
- Fisher, J.A. 1992. Trace Scheduling-2, an extension of trace scheduling. Tech. rept., Hewlett-Packard Laboratories.
- Fisher, J.A., and Freudberger, S.M. 1992. Predicting conditional jump directions from previous runs of a program. In *Proc., Fifth Internat. Conf. on Architectural Support for Programming Languages and Operating Systems* (Boston, Oct.), pp. 85–95.
- Fisher, J.A., Landskov, D., and Shriver, B.D. 1981. Microcode compaction: Looking backward and looking forward. In *Proc., 1981 Nat. Computer Conf.*, pp. 95–102.
- Fisher, J.A., Ellis, J.R., Ruttenberg, J.C., and Nicolau, A. 1984. Parallel processing: A smart compiler and a dumb machine. In *Proc., ACM SIGPLAN '84 Symp. on Compiler Construction* (Montreal, June), pp. 37–47.
- Floating Point Systems. 1979. *FPS AP-120B Processor Handbook*. Floating Point Systems, Inc., Beaverton, Ore.
- Foster, C.C., and Riseman, E.M. 1972. Percolation of code to enhance parallel dispatching and execution. *IEEE Trans. Comps.*, C-21, 12 (Dec.): 1411–1415.
- Franklin, M., and Sohi, G.S. 1992. The expandable split window paradigm for exploiting fine-grain parallelism. In *Proc. 19th Annual International Symp. on Computer Architecture* (Gold Coast, Australia, May), pp. 58–67.
- Freudberger, S.M., and Ruttenberg, J.C. 1992. Phase ordering of register allocation and instruction scheduling. In *Code Generation—Concepts, Tools, Techniques: Proc., Internat. Workshop on Code Generation*, May 1991 (R. Giegerich, and S.L. Graham, eds.), Springer-Verlag, London, pp. 146–172.
- Gasperoni, F. 1989. Compilation techniques for VLIW architectures. Tech. rept. RC 14915, IBM Research Div., T.J. Watson Research Center, Yorktown Heights, N.Y.
- Gibbons, P.B., and Muchnick, S.S. 1986. Efficient instruction scheduling for a pipelined architecture. In *Proc., ACM SIGPLAN '86 Symp. on Compiler Construction* (Palo Alto, Calif., July), pp. 11–16.
- Golumbic, M.C., and Rainish, V. 1990. Instruction scheduling beyond basic blocks. *IBM J. Res. and Dev.*, 34, 1 (Jan.): 93–97.
- Gonzalez, M.J. 1977. Deterministic processor scheduling. *ACM Computer Surveys*, 9, 3 (Sept.): 173–204.
- Goodman, J.R., and Hsu, W.-C. 1988. Code scheduling and register allocation in large basic blocks. In *Proc., 1988 Internat. Conf. on Supercomputing* (St. Malo, France, July), pp. 442–452.
- Grishman, R., and Su, B. 1983. A preliminary evaluation of trace scheduling for global microcode compaction. *IEEE Trans. Comps.*, C-32, 12 (Dec.): 1191–1194.
- Gross, T.R., and Hennessy, J.L. 1982. Optimizing delayed branches. In *Proc., 15th Annual Workshop on Microprogramming* (Oct.), pp. 114–120.

- Gross, T., and Ward, M. 1990. The suppression of compensation code. In *Advances in Languages and Compilers for Parallel Computing* (A. Nicolau, D. Gelernter, T. Gross, and D. Padua, eds.), Pitman/MIT Press, London, pp. 260–273.
- Gurd, J., Kirkham, C.C., and Watson, I. 1985. The Manchester prototype dataflow computer. *CACM*, 28, 1(Jan.): 34–52.
- Hallin, T.G., and Flynn, M.J. 1972. Pipelining of arithmetic functions. *IEEE Trans. Comps.*, C-21, 8 (Aug.): 880–886.
- Hendren, L.J., Gao, G.R., Altman, E.R., and Mukerji, C. 1992. Register allocation using cyclic interval graphs: A new approach to an old problem. ACAPS Tech. Memo 33, Advanced Computer Architecture and Program Structures Group, McGill Univ., Montreal.
- Hennessy, J.L., and Gross, T. 1983. Post-pass code optimization of pipelined constraints. *ACM Trans. Programming Languages and Systems*, 5, 3 (July): 422–448.
- Hennessy, J., Jouppi, N., Baskett, F., Gross, T., and Gill, J. 1982. Hardware/software tradeoffs for increased performance. In *Proc., Symp. on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Calif., Mar.) pp. 2–11.
- Hennessy, J., Jouppi, N., Przybylski, S., Rowen, C., Gross, T., Baskett, F., and Gill, J. 1982. MIPS: A microprocessor architecture. In *Proc., 15th Annual Workshop on Microprogramming* (Palo Alto, Calif., Oct.), pp. 17–22.
- Hintz, R.G., and Tate, D.P. 1972. Control Data STAR-100 processor design. In *Proc., COMPCON '72* (Sept.), pp. 1–4.
- Hsu, P.Y.T. 1986. Highly concurrent scalar processing. Ph.D. thesis, Univ. of Ill., Urbana-Champaign, Ill.
- Hsu, P.Y.T., and Davidson, E.S. 1986. Highly concurrent scalar processing. In *Proc., Thirteenth Annual Internat. Symp. on Computer Architecture*, pp. 386–395.
- Hsu, W.-C. 1987. Register allocation and code scheduling for load/store architectures. Comp. Sci. Tech. Rept. no. 722, Univ. of Wisc., Madison.
- Hu, T.C. 1961. Parallel sequencing and assembly line problems. *Operations Research*, 9, 6: 841–848.
- Hwu, W.W., and Chang, P.P. 1988. Exploiting parallel microprocessor microarchitectures with a compiler code generator. In *Proc., 15th Annual Internat. Symp. on Computer Architecture* (Honolulu, May), pp. 45–53.
- Hwu, W.W., and Patt, Y.N. 1986. HPSm, a high performance restricted data flow architecture having minimal functionality. In *Proc., 13th Annual Internat. Symp. on Computer Architecture* (Tokyo, June), pp. 297–306.
- Hwu, W.W., and Patt, Y.N. 1987. Checkpoint repair for out-of-order execution machines. *IEEE Trans. Comps.*, C-36, 12 (Dec.): 1496–1514.
- Hwu, W.W., Conte, T.M., and Chang, P.P. 1989. Comparing software and hardware schemes for reducing the cost of branches. In *Proc., 16th Annual Internat. Symp. on Computer Architecture* (May), pp. 224–233.
- Hwu, W.W., Mahlke, S.A., Chen, W.Y., Chang, P.P., Warter, N.J., Bringmann, R.A., Ouellette, R.G., Hank, R.E., Kiyohara, T., Haab, G.E., Holm, J.G., and Lavery, D.M. 1993. The superblock: An effective technique for VLIW and superscalar compilation. *The J. Supercomputing*, 7, 1/2: 229–248.
- IBM. 1967. *IBM J. Res. and Dev.*, 11, 1 (Jan.). Special issue on the System/360 Model 91.
- IBM. 1976. *IBM 3838 Array Processor Functional Characteristics*. Pub. no. 6A24-3639-0, file no. S370-08, IBM Corp., Endicott, N.Y.
- IBM. 1990. *IBM J. Res. and Dev.*, 34, 1 (Jan.). Special issue on the IBM RISC System/6000 processor.
- Intel. 1989a. *i860 64-Bit Microprocessor Programmer's Reference Manual*. Pub. no. 240329-001, Intel Corp., Santa Clara, Calif.
- Intel. 1989b. *80960CA User's Manual*. Pub. no. 270710-001, Intel Corp., Santa Clara, Calif.
- Jain, S. 1991. Circular scheduling: A new technique to perform software pipelining. In *Proc., ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation* (June), pp. 219–228.
- Johnson, M. 1991. *Superscalar Microprocessor Design*. Prentice-Hall, Englewood Cliffs, N.J.
- Jouppi, N.P. 1989. The nonuniform distribution of instruction-level and machine parallelism and its effect on performance. *IEEE Trans. Comps.*, C-38, 12 (Dec.): 1645–1658.
- Jouppi, N.P., and Wall, D. 1989. Available instruction level parallelism for superscalar and superpipelined machines. In *Proc., Third Internat. Conf. on Architectural Support for Programming Languages and Operating Systems* (Boston, Apr.), pp. 272–282.
- Kasahara, H., and Narita, S. 1984. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Trans. Comps.*, C-33, 11 (Nov.): 1023–1029.
- Keller, R.M. 1975. Look-ahead processors. *Computing Surveys* 7, 4 (Dec.): 177–196.
- Kleir, R.L. 1974. A representation for the analysis of microprogram operation. In *Proc., 7th Annual Workshop on Microprogramming* (Sept.), pp. 107–118.
- Kleir, R.L., and Ramamoorthy, C.V. 1971. Optimization strategies for microprograms. *IEEE Trans. Comps.*, C-20, 7 (July): 783–794.
- Kogge, P.M. 1973. Maximal rate pipelined solutions to recurrence programs. In *Proc., First Annual Symp. on Computer Architecture* (Univ. of Fla., Gainesville, Dec.), pp. 71–76.
- Kogge, P.M. 1974. Parallel solution of recurrence problems. *IBM J. Res. and Dev.*, 18, 2 (Mar.): 138–148.
- Kogge, P.M. 1977a. Algorithm development for pipelined processors. In *Proc., 1977 Internat. Conf. on Parallel Processing* (Aug.), p. 217.
- Kogge, P.M. 1977b. The microprogramming of pipelined processors. In *Proc., 4th Annual Symp. on Computer Architecture* (Mar.), pp. 63–69.
- Kogge, P.M. 1981. *The Architecture of Pipelined Computers*. McGraw-Hill, New York.
- Kogge, P.M., and Stone, H.S. 1973. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Comps.*, C-22, 8 (Aug.): 786–793.
- Kohler, W.H. 1975. A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems. *IEEE Trans. Comps.*, C-24, 12 (Dec.): 1235–1238.
- Kohn, L., and Margulis, N. 1989. Introducing the Intel i860 64-bit microprocessor. *IEEE Micro*, 9, 4 (Aug.): 15–30.
- Kunkel, S.R., and Smith, J.E. 1986. Optimal pipelining in supercomputers. In *Proc., 13th Annual Internat. Symp. on Computer Architecture* (Tokyo, June), pp. 404–411.
- Labrousse, J., and Slavenburg, G.A. 1988. CREATE-LIFE: A design system for high performance VLSI circuits. In *Proc., Internat. Conf. on Circuits and Devices*, pp. 365–360.
- Labrousse, J., and Slavenburg, G.A. 1990a. A 50 MHz microprocessor with a VLIW architecture. In *Proc., ISSCC '90* (San Francisco), pp. 44–45.
- Labrousse, J., and Slavenburg, G.A. 1990b. CREATE-LIFE: A modular design approach for high performance ASICs. In *Proc., COMPCON '90* (San Francisco), pp. 427–433.
- Lam, M.S.-L. 1987. A systolic array optimizing compiler. Ph.D. thesis, Carnegie Mellon Univ., Pittsburgh.
- Lam, M. 1988. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc., ACM SIGPLAN '88 Conf. on Programming Language Design and Implementation* (Atlanta, June), pp. 318–327.
- Lam, M.S., and Wilson, R.P. 1992. Limits of control flow on parallelism. In *Proc., Nineteenth Internat. Symp. on Computer Architecture* (Gold Coast, Australia, May), pp. 46–57.
- Landskow, D., Davidson, S., Shriver, B., and Mallett, P.W. 1980. Local microcode compaction techniques. *ACM Computer Surveys*, 12, 3 (Sept.): 261–294.
- Lee, J.K.F., and Smith, A.J. 1984. Branch prediction strategies and branch target buffer design. *Computer*, 17, 1 (Jan.): 6–22.
- Lee, M., Tirumalai, P.P., and Ngai, T.-F. 1993. Software pipelining and superblock scheduling: Compilation techniques for VLIW machines. In *Proc., 26th Annual Hawaii Internat. Conf. on System Sciences* (Hawaii, Jan.), vol. 1, pp. 202–213.
- Linn, J.L. 1988. Horizontal microcode compaction. In *Microprogramming and Firmware Engineering Methods* (S. Habib, ed.), Van Nostrand Reinhold, New York, pp. 381–431.
- Lowney, P.G., Freudenberg, S.M., Karzes, T.J., Lichtenstein, W.D., Nix, R.P., O'Donnell, J.S., and Ruttenburg, J.C. 1993. The Multiflow trace scheduling compiler. *The J. Supercomputing*, 7, 1/2: 51–142.
- Mahlke, S.A., Chen, W.Y., Hwu, W.W., Rau, B.R., and Schlansker, M.S. 1992. Sentinel scheduling for VLIW and superscalar processors. In *Proc., Fifth Internat. Conf. on Architectural Support for Programming Languages and Operating Systems* (Boston, Oct.), pp. 238–247.
- Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E., and Bringmann, R.A. 1992. Effective compiler support for predicated execution using the hyperblock. In *Proc., 25th Annual Internat. Symp. on Microarchitecture* (Dec.), pp. 45–54.
- Mallett, P.W. 1978. Methods of compacting microprograms. Ph.D. thesis, Univ. of Southwestern La., Lafayette, La.
- Mangione-Smith, W., Abraham, S.G., and Davidson, E.S. 1992. Register requirements of pipelined processors. In *Proc., Internat. Conf. on Supercomputing* (Washington, D.C., July).
- McFarling, S., and Hennessy, J. 1986. Reducing the cost of branches. In *Proc., Thirteenth Internat. Symp. on Computer Architecture* (Tokyo, June), pp. 396–403.
- Moon, S.-M., Ebcioiglu, K. 1992. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. In *Proc., 25th Annual Internat. Symp. on Microarchitecture* (Portland, Ore., Dec.), pp. 55–71.

- Nakatani, T., and Ebcioglu, K. 1990. Using a lookahead window in a compaction-based parallelizing compiler. In *Proc., 23rd Annual Workshop on Microprogramming and Microarchitecture* (Orlando, Fla., Nov.), pp. 57–68.
- Nicolau, A. 1984. Parallelism, memory anti-aliasing and correctness for trace scheduling compilers. Ph.D. thesis, Yale Univ., New Haven, Conn.
- Nicolau, A. 1985a. Percolation scheduling: A parallel compilation technique. Tech. Rept. TR 85-678, Dept. of Comp. Sci., Cornell, Ithaca, N.Y.
- Nicolau, A. 1985b. Uniform parallelism exploitation in ordinary programs. In *Proc., Internat. Conf. on Parallel Processing* (Aug.), pp. 614–618.
- Nicolau, A., and Fisher, J.A. 1981. Using an oracle to measure parallelism in single instruction stream programs. In *Proc., Fourteenth Annual Microprogramming Workshop* (Oct.), pp. 171–182.
- Nicolau, A., and Fisher, J.A. 1984. Measuring the parallelism available for very long instruction word architectures. *IEEE Trans. Comps.*, C-33, 11 (Nov.): 968–976.
- Nicolau, A., and Potasman, R. 1990. Realistic scheduling: Compaction for pipelined architectures. In *Proc., 23rd Annual Workshop on Microprogramming and Microarchitecture* (Orlando, Fla., Nov.), pp. 69–79.
- Oehler, R.R., and Blasgen, M.W. 1991. IBM RISC System/6000: Architecture and performance. *IEEE Micro*, 11, 3 (June): 14.
- Papadopoulos, G.M., and Culler, D.E. 1990. Monsoon: An explicit token store architecture. In *Proc., Seventeenth Internat. Symp. on Computer Architecture* (Seattle, May), pp. 82–91.
- Park, J.C.H., and Schlansker, M.S. 1991. On predicated execution. Tech. Rept. HPL-91-58, Hewlett Packard Laboratories.
- Patel, J.H. 1976. Improving the throughput of pipelines with delays and buffers. Ph.D. thesis, Univ. of Ill., Urbana-Champaign, Ill.
- Patel, J.H., and Davidson, E.S. 1976. Improving the throughput of a pipeline by insertion of delays. In *Proc., 3rd Annual Symp. on Computer Architecture* (Jan.), pp. 159–164.
- Patterson, D.A., and Sequin, C.H. 1981. RISC I: A reduced instruction set VLSI computer. In *Proc., 8th Annual Symp. on Computer Architecture* (Minneapolis, May), pp. 443–450.
- Peterson, C., Sutton, J., and Wiley, P. 1991. iWarp: A 100-MOPS, LIW microprocessor for multicomputers. *IEEE Micro*, 11, 3 (June): 26.
- Popescu, V., Schultz, M., Spracklen, J., Gibson, G., Lightner, B., and Isaman, D. 1991. The Metaflow architecture. *IEEE Micro*, 11, 3 (June): 10.
- Radin, G. 1982. The 801 minicomputer. In *Proc., Symp. on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Calif., Mar.), pp. 39–47.
- Ramakrishnan, S. 1992. Software pipelining in PA-RISC compilers. *Hewlett-Packard J.* (July): 39–45.
- Ramamoorthy, C.V., and Gonzalez, M.J. 1969. A survey of techniques for recognizing parallel processable streams in computer programs. In *Proc., AFIPS Fall Joint Computing Conf.*, pp. 1–15.
- Ramamoorthy, C.V., and Tsuchiya, M. 1974. A high level language for horizontal microprogramming. *IEEE Trans. Comps.*, C-23: 791–802.
- Ramamoorthy, C.V., Chandy, K.M., and Gonzalez, M.J. 1972. Optimal scheduling strategies in a multiprocessor system. *IEEE Trans. Comps.*, C-21, 2 (Feb.): 137–146.
- Rau, B.R. 1988. Cydra 5 Directed Dataflow architecture. In *Proc., COMPCON '88* (San Francisco, Mar.), pp. 106–113.
- Rau, B.R. 1992. Data flow and dependence analysis for instruction level parallelism. In *Fourth Internat. Workshop on Languages and Compilers for Parallel Computing* (U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, eds.), Springer-Verlag, pp. 236–250.
- Rau, B.R., and Glaeser, C.D. 1981. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proc., Fourteenth Annual Workshop on Microprogramming* (Oct.), pp. 183–198.
- Rau, B.R., Glaeser, C.D., and Greenawalt, E.M. 1982. Architectural support for the efficient generation of code for horizontal architectures. In *Proc., Symp. on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Calif., Mar.), pp. 96–99.
- Rau, B.R., Glaeser, C.D., and Picard, R.L. 1982. Efficient code generation for horizontal architectures: Compiler techniques and architectural support. In *Proc., Ninth Annual Internat. Symp. on Computer Architecture* (Apr.), pp. 131–139.
- Rau, B.R., Lee, M., Tirumalai, P., and Schlansker, M.S. 1992. Register allocation for software pipelined loops. In *Proc., SIGPLAN '92 Conf. on Programming Language Design and Implementation* (San Francisco, June 17–19), pp. 283–299.
- Rau, B.R., Yen, D.W.L., Yen, W., and Towle, R.A. 1989. The Cydra 5 departmental supercomputer: Design philosophies, decisions and trade-offs. *Computer*, 22, 1 (Jan.): 12–34.
- Riseman, E.M., and Foster, C.C. 1972. The inhibition of potential parallelism by conditional jumps. *IEEE Trans. Comps.*, C-21, 12 (Dec.): 1405–1411.
- Ruggiero, J.F., and Coryell, D.A. 1969. An auxiliary processing system for array calculations. *IBM Systems J.*, 8, 2: 118–135.
- Russell, R.M. 1978. The CRAY-1 computer system. *CACM*, 21: 63–72.
- Rymarczyk, J. 1982. Coding guidelines for pipelined processors. In *Proc., Symp. on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Calif., Mar.), pp. 12–19.
- Schmidt, U., and Caesar, K. 1991. Datawave: A single-chip multiprocessor for video applications. *IEEE Micro*, 11, 3 (June): 22.
- Schneck, P.B. 1987. *Supercomputer Architecture*. Kluwer Academic, Norwell, Mass.
- Schutte, M.A., and Shen, J.P. 1993. Instruction-level experimental evaluation of the Multiflow TRACE 14/300 VLIW computer. *The J. Supercomputing*, 7, 1/2: 249–271.
- Sethi, R. 1975. Complete register allocation problems. *SIAM J. Computing*, 4, 3: 226–248.
- Sethi, R., and Ullman, J.D. 1970. The generation of optimal code for arithmetic expressions. *JACM*, 17, 4 (Oct.): 715–728.
- Sites, R.L. 1978. Instruction ordering for the CRAY-1 computer. Tech. rept. 78-CS-023, Univ. of Calif., San Diego.
- Smith, J.E. 1981. A study of branch prediction strategies. In *Proc., Eighth Annual Internat. Symp. on Computer Architecture* (May), pp. 135–148.
- Smith, J.E. 1982. Decoupled access/execute architectures. In *Proc., Ninth Annual Internat. Symp. on Computer Architecture* (Apr.), pp. 112–119.
- Smith, J.E. 1989. Dynamic instruction scheduling and the Astronautics ZS-1. *Computer*, 22, 1 (Jan.): 21–35.
- Smith, J.E., and Pleszakun, A.R. 1988. Implementing precise interrupts in pipelined processors. *IEEE Trans. Comps.*, C-37, 5 (May): 562–573.
- Smith, J.E., Dermer, G.E., Vanderwarrn, B.D., Klinger, S.D., Roszewski, C.M., Fowler, D.L., Scidmore, K.R., and Laudon, J.P. 1987. The ZS-1 central processor. In *Proc., Second Internat. Conf. on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Calif., Oct.), pp. 199–204.
- Smith, M.D., Horowitz, M., and Lam, M. 1992. Efficient superscalar performance through boosting. In *Proc., Fifth Internat. Conf. on Architectural Support for Programming Languages and Operating Systems* (Boston, Oct.), pp. 248–259.
- Smith, M.D., Lam, M.S., and Horowitz, M.A. 1990. Boosting beyond static scheduling in a superscalar processor. In *Proc., Seventeenth Internat. Symp. on Computer Architecture* (June), pp. 344–354.
- Smotherman, M., Krishnamurthy, S., Aravind, P.S., and Hunnicutt, D. 1991. Efficient DAG construction and heuristic calculation for instruction scheduling. In *Proc., 24th Annual Internat. Workshop on Microarchitecture* (Albuquerque, N.M., Nov.), pp. 93–102.
- Sohi, G.S., and Vajapayem, S. 1987. Instruction issue logic for high-performance, interruptable pipelined processors. In *Proc., 14th Annual Symp. on Computer Architecture* (Pittsburgh, June), pp. 27–36.
- Su, B., and Ding, S. 1985. Some experiments in global microcode compaction. In *Proc., 18th Annual Workshop on Microprogramming* (Asilomar, Calif., Nov.), pp. 175–180.
- Su, B., and Wang, J. 1991a. GURPR*: A new global software pipelining algorithm. In *Proc., 24th Annual Internat. Symp. on Microarchitecture* (Albuquerque, N.M., Nov.), pp. 212–216.
- Su, B., and Wang, J. 1991b. Loop-carried dependence and the general URPR software pipelining approach. In *Proc., 24th Annual Hawaii Internat. Conf. on System Sciences* (Hawaii, Jan.).
- Su, B., Ding, S., and Jin, L. 1984. An improvement of trace scheduling for global microcode compaction. In *Proc., 17th Annual Workshop on Microprogramming* (New Orleans, Oct.), pp. 78–85.
- Su, B., Ding, S., and Xia, J. 1986. URPR—An extension of URCR for software pipelining. In *Proc., 19th Annual Workshop on Microprogramming* (New York, Oct.), pp. 104–108.
- Su, B., Ding, S., Wang, J., and Xia, J. 1987. GURPR—A method for global software pipelining. In *Proc., 20th Annual Workshop on Microprogramming* (Colorado Springs, Colo., Dec.), pp. 88–96.

- Thistle, M.R., and Smith, B.J. 1988. A processor architecture for Horizon. In *Proc., Supercomputing '88*, (Orlando, Fla., Nov.), pp. 35–41.
- Thomas, A.T., and Davidson, E.S. 1974. Scheduling of multiconfigurable pipelines. In *Proc., 12th Annual Allerton Conf. on Circuits and Systems Theory* (Allerton, Ill.), pp. 658–669.
- Thornton, J.E. 1964. Parallel operation in the Control Data 6600. In *Proc., AFIPS Fall Joint Computer Conf.*, pp. 33–40.
- Thornton, J.E. 1970. *Design of a Computer—The Control Data 6600*. Scott, Foresman, Glenview, Ill.
- Tirumalai, P., Lee, M., and Schlansker, M.S. 1990. Parallelization of loops with exits on pipelined architectures. In *Proc., Supercomputing '90* (Nov.), pp. 200–212.
- Tjaden, G.S., and Flynn, M.J. 1970. Detection and parallel execution of parallel instructions. *IEEE Trans. Comps.*, C-19, 10 (Oct.): 889–895.
- Tjaden, G.S., and Flynn, M.J. 1973. Representation of concurrency with ordering matrices. *IEEE Trans. Comps.*, C-22, 8 (Aug.): 752–761.
- Tokoro, M., Tamura, E., and Takizuka, T. 1981. Optimization of microprograms. *IEEE Trans. Comps.*, C-30, 7 (July): 491–504.
- Tokoro, M., Takizuka, T., Tamura, E., and Yamaura, I. 1978. A technique of global optimization of microprograms. In *Proc., 11th Annual Workshop on Microprogramming* (Asilomar, Calif., Nov.), pp. 41–50.
- Tokoro, M., Tamura, E., Takase, K., and Tamaru, K. 1977. An approach to microprogram optimization considering resource occupancy and instruction formats. In *Proc., 10th Annual Workshop on Microprogramming* (Niagara Falls, N.Y., Nov.), pp. 92–108.
- Tomasulo, R.M. 1967. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. and Dev.*, 11, 1 (Jan.): 25–33.
- Touzeau, R.F. 1984. A FORTRAN compiler for the FPS-164 scientific computer. In *Proc., ACM SIGPLAN '84 Symp. on Compiler Construction* (Montreal), pp. 48–57.
- Tsuchiya, M., and Gonzalez, M.J. 1974. An approach to optimization of horizontal microprograms. In *Proc., Seventh Annual Workshop on Microprogramming* (Palo Alto, Calif.), pp. 85–90.
- Tsuchiya, M., and Gonzalez, M.J. 1976. Toward optimization of horizontal microprograms. *IEEE Trans. Comps.*, C-25, 10 (Oct.): 992–999.
- Uht, A.K. 1986. An efficient hardware algorithm to extract concurrency from general-purpose code. In *Proc., Nineteenth Annual Hawaii Conf. on System Sciences* (Jun.), pp. 41–50.
- Wall, D.W. 1991. Limits of instruction-level parallelism. In *Proc., Fourth Internat. Conf. on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, Calif., Apr.), pp. 176–188.
- Warren, H.S. 1990. Instruction scheduling for the IBM RISC System/6000 processor. *IBM J. Res. and Dev.*, 34, 1 (Jan.): 85–92.
- Warter, N.J., Bockhaus, J.W., Haab, G.E., and Subramanian, K. 1992. Enhanced modulo scheduling for loops with conditional branches. In *Proc., 25th Annual Internat. Symp. on Microarchitecture* (Portland, Ore., Dec.), pp. 170–179.
- Watson, W.J. 1972. The TI ASC—A highly modular and flexible super computer architecture. In *Proc., AFIPS Fall Joint Computer Conf.*, pp. 221–228.
- Wedig, R.G. 1982. Detection of concurrency in directly executed language instruction streams. Ph.D. thesis, Stanford Univ., Stanford, Calif.
- Weiss, S., and Smith, J.E. 1984. Instruction issue logic for pipelined supercomputers. In *Proc., 11th Annual Internat. Symp. on Computer Architecture*, pp. 110–118.
- Weiss, S., and Smith, J.E. 1987. A study of scalar compilation techniques for pipelined supercomputers. In *Proc., Second Internat. Conf. on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Calif., Oct.), pp. 105–109.
- Wilkes, M.V. 1951. The best way to design an automatic calculating machine. In *Proc., Manchester Univ. Comp. Inaugural Conf.* (Manchester, England, July), pp. 16–18.
- Wilkes, M.V., and Stringer-J.B. 1953. Microprogramming and the design of the control circuits in an electronic digital computer. In *Proc., The Cambridge Philosophical Society, Part 2* (Apr.), pp. 230–238.
- Wolfe, A., and Shen, J.P. 1991. A variable instruction stream extension to the VLIW architecture. In *Proc., Fourth Internat. Conf. on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, Calif., Apr.), pp. 2–14.
- Wood, G. 1978. On the packing of micro-operations into micro-instruction words. In *Proc., 11th Annual Workshop on Microprogramming* (Asilomar, Calif., Nov.), pp. 51–55.
- Wood, G. 1979. Global optimization of microprograms through modular control constructs. In *Proc., 12th Annual Workshop on Microprogramming* (Hershey, Penn.), pp. 1–6.
- Yau, S.S., Schow, A.C., and Tsuchiya, M. 1974. On storage optimization of horizontal microprograms. In *Proc., Seventh Annual Workshop on Microprogramming* (Palo Alto, Calif.), pp. 98–106.
- Yeh, T.Y., and Patt, Y.N. 1992. Alternative implementations of two-level adaptive branch prediction. In *Proc., Nineteenth Internat. Symp. on Comp. Architecture* (Gold Coast, Australia, May), pp. 124–134.
- Zima, H., and Chapman, B. 1990. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley, Reading, Mass.

ROCK: A HIGH-PERFORMANCE SPARC CMT PROCESSOR

Rock, Sun's third-generation chip-multipathing processor, contains 16 high-performance cores, each of which can support two software threads. Rock uses a novel checkpoint-based architecture to support automatic hardware scouting under a load miss, speculative out-of-order retirement of instructions, and aggressive dynamic hardware parallelization of a sequential instruction stream. It is also the first processor to support transactional memory in hardware.

Shailender Chaudhry

Robert Cypher

Magnus Ekman

Martin Karlsson

Anders Landin

Sherman Yip

Håkan Zeffler

Marc Tremblay

Sun Microsystems

• • • • • Designing an aggressive chip-multipath (CMT) processor¹ involves many tradeoffs. To maximize throughput performance, each processor core must be highly area and power efficient, so that many cores can coexist on a single die. Similarly, if the processor is to perform well on a wide spectrum of applications, per-thread performance must be high so that serial code also executes efficiently, minimizing performance problems related to Amdahl's law.^{2,3} Rock, Sun's third-generation chip-multipathing processor, uses a novel resource-sharing hierarchical structure tuned to maximize performance per area and power.⁴

Rock uses an innovative checkpoint-based architecture to implement highly aggressive speculation techniques. Each core has support for two threads, each with one checkpoint. The hardware support for threads can be used two different ways: a single core can run two application threads, giving each thread hardware support for *execute ahead* (EA) under cache misses; or all of a core's resources can adaptively be combined to run one application thread with

even more aggressive *simultaneous speculative threading* (SST), which uses two checkpoints. EA is an area-efficient way of creating a large virtual issue window without the large associative structures. SST dynamically extracts parallelism, letting execution proceed in parallel at two different points in the program.

These schemes let Rock maximize throughput when the parallelism is available, and focus the hardware resources to boost per-thread performance when it is at a premium. Because these speculation techniques make Rock less sensitive to cache misses, we have been able to reduce on-chip cache sizes and instead add more cores to further increase performance. The high core-to-cache ratio requires high off-chip bandwidth. Rock-based systems use large caches in the off-chip memory controllers to maintain a relatively low access rate to DRAM; this maximizes performance and optimizes total system power.

To help the programmability of multiprocessor systems with hundreds of threads and to improve synchronization-related performance, Rock supports transactional

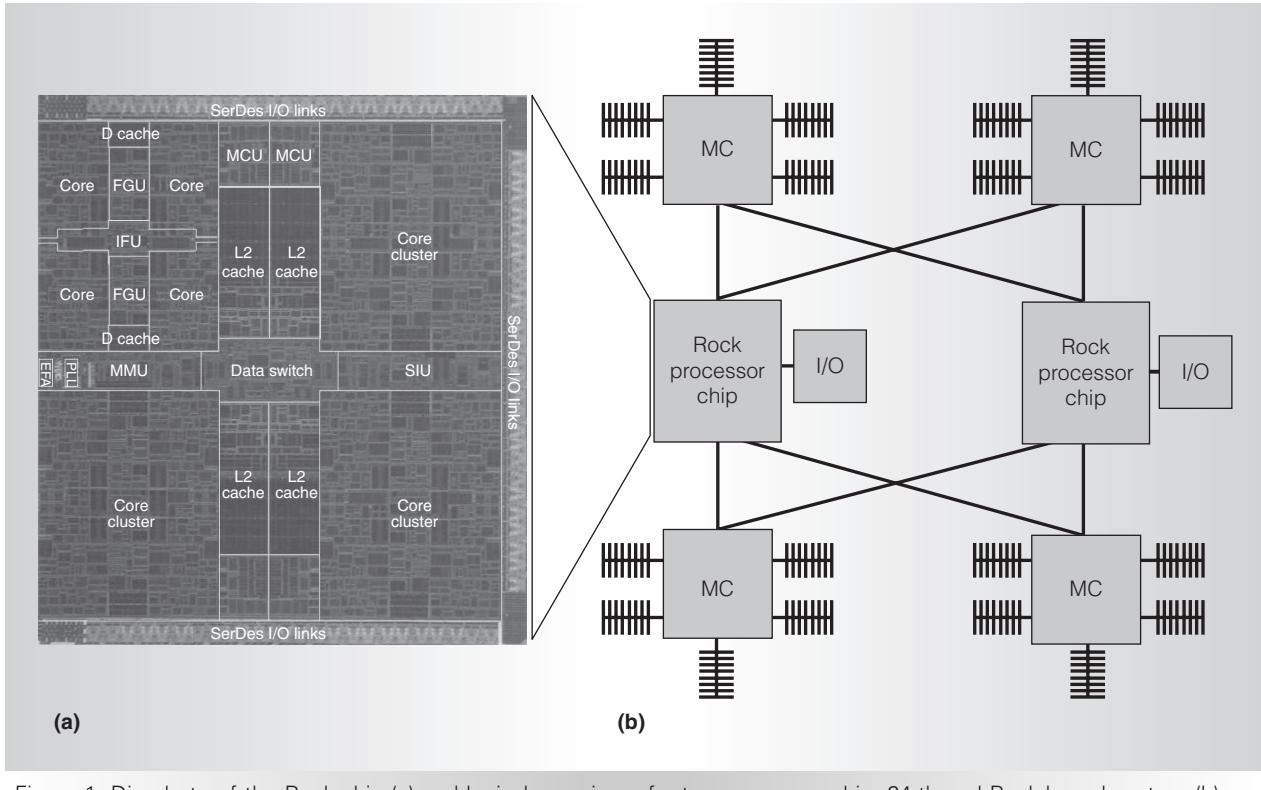


Figure 1. Die photo of the Rock chip (a) and logical overview of a two-processor-chip, 64-thread Rock-based system (b).

memory⁵ in hardware; it is the first commercial processor to do so. TM lets a Rock thread efficiently execute all instructions in a transaction as an atomic unit, or not at all. Rock leverages the checkpointing mechanism with which it supports EA to implement the TM semantics with low additional hardware overhead.

System overview

Figure 1 shows a die photo of the Rock processor and a logical overview of a Rock-based system using two processor chips. Each Rock processor has 16 cores, each configurable to run one or two threads; thus, each chip can run up to 32 threads. The 16 cores are divided into core clusters, with four cores per cluster. This design enables resource sharing among cores within a cluster, thus reducing the area requirements. All cores in a cluster share an instruction fetch unit (IFU) that includes the level-one (L1) instruction cache. We decided that all four cores in a cluster should share the IFU because it is relatively simple to fetch a large

number of instructions per cycle. Thus, four cores can share one IFU in a round-robin fashion while maintaining full fetch bandwidth. Furthermore, the shared instruction cache enables constructive sharing of common code, as is encountered in shared libraries and operating system routines.

Each core cluster contains two L1 data caches (D cache) and two floating-point units (FGU), each of which is shared by a pair of cores. As Figure 1a shows, these structures are relatively large, so sharing them provides significant area savings. A crossbar connects the four core clusters with the four level-two banks (L2 cache), the second-level memory-management unit (MMU), and the system interface unit (SIU). Rock's L2 cache is 2 Mbytes, consisting of four 512-Kbyte 8-way set-associative banks. A relatively large fraction of the die area is devoted to core clusters as opposed to L2 cache. We selected this design point after simulating designs with different ratios of core clusters to L2 cache. It provides the highest

throughput of the different options and yields good single-thread performance.

Each Rock chip is directly connected to an I/O bridge ASIC and to multiple memory controller ASICs via high-bandwidth serializer/deserializer (SerDes) links. This system topology provides truly uniform access to all memory locations. The memory controllers implement a directory-based MESI coherence protocol (which uses the states *modified*, *exclusive*, *shared*, and *invalid*) supporting multiprocessor configurations. We decided not to support the *owned* cache state (O) because there are no direct links between processors, and so accesses to a cache line in the O state in another processor would require a four-hop transaction, which would increase the latency and the link bandwidth requirements.

To further reduce link bandwidth requirements, Rock systems use hardware to compress packets sent over the links between the processors and the memory controllers. Each memory controller has an 8-Mbyte level-three cache, with all the L3 caches in the system forming a single large, globally shared cache. This L3 cache reduces both the latency of memory requests and the DRAM bandwidth requirements. Each memory controller has $4 + 1$ fully buffered dual in-line memory module (FB-DIMM) memory channels, with the fifth of these used for redundancy.

Pipeline overview

A *strand* consists of the hardware resources needed to execute a software thread. Thus, in a Rock system, eight strands share a fetch unit, which uses a round-robin scheme to determine which strand gets access to the instruction cache each cycle. The 32-Kbyte shared instruction cache is four-way set associative, and virtual-to-physical translation is provided by a 64-entry, four-way-associative instruction translation look-aside buffer (ITLB). Because fetch logic in the IFU keeps track of when a fetch request can cross a page boundary, most fetch requests can bypass the ITLB access and thereby save ITLB power and fetch latency. Each strand has a sequential next-cache-line prefetch mechanism that automatically requests the next cache line, thereby

exploiting the great spatial locality typical in instruction data. The fetch unit fetches a full cache line of 16 instructions and forwards it to a strand-specific fetch buffer.

Rock's branch predictor is a 60-Kbit (30K two-bit saturating counters) gshare branch predictor, enhanced to take advantage of the software prediction bit provided in Sparc V9 branches. In each cycle, two program-counter-relative (PC-relative) branches are predicted. Indirect branches are predicted by either a 128-entry branch-target buffer or a 128-entry return-address predictor.

As Figure 2 shows, the fetch stages are connected to the decode pipeline through an instruction buffer. The decode unit, which serves one strand per cycle, has a helper ROM that it uses to decompose complex instructions into sequences of simpler instructions. Once decoded, instructions proceed to the appropriate strand-specific instruction queue. A round-robin arbiter decides which strand will have access to the issue pipeline the next cycle. On every cycle, up to four instructions can be steered from the instruction queue to the issue first-in, first-out buffers (FIFOs). There are five issue FIFOs (labeled A0, A1, BR, FP, and MEM in Figure 2); each is three entries deep and directly connected to its respective execution pipe. An issue FIFO will stall an instruction if there is a read-after-write (RAW) hazard. However, such a stall does not necessarily affect instructions in the other issue FIFOs. Independent younger instructions in the other issue FIFOs can proceed to the execution unit out of order. Instructions that can't complete their execution go into the deferred instruction queue (DQ) for later reexecution. The DQ is an SRAM-based structure with only a single read/write port, so it is area and power efficient.

A two-level register file provides register operands. The first-level register file is a highly ported structure to support the source operand requirements of the maximum issue width.⁶ It contains one register window, whereas the second-level register file contains all of the eight register windows that Rock supports. The second-level register file has a single read/write port and is made out of SRAM. For each architectural register in



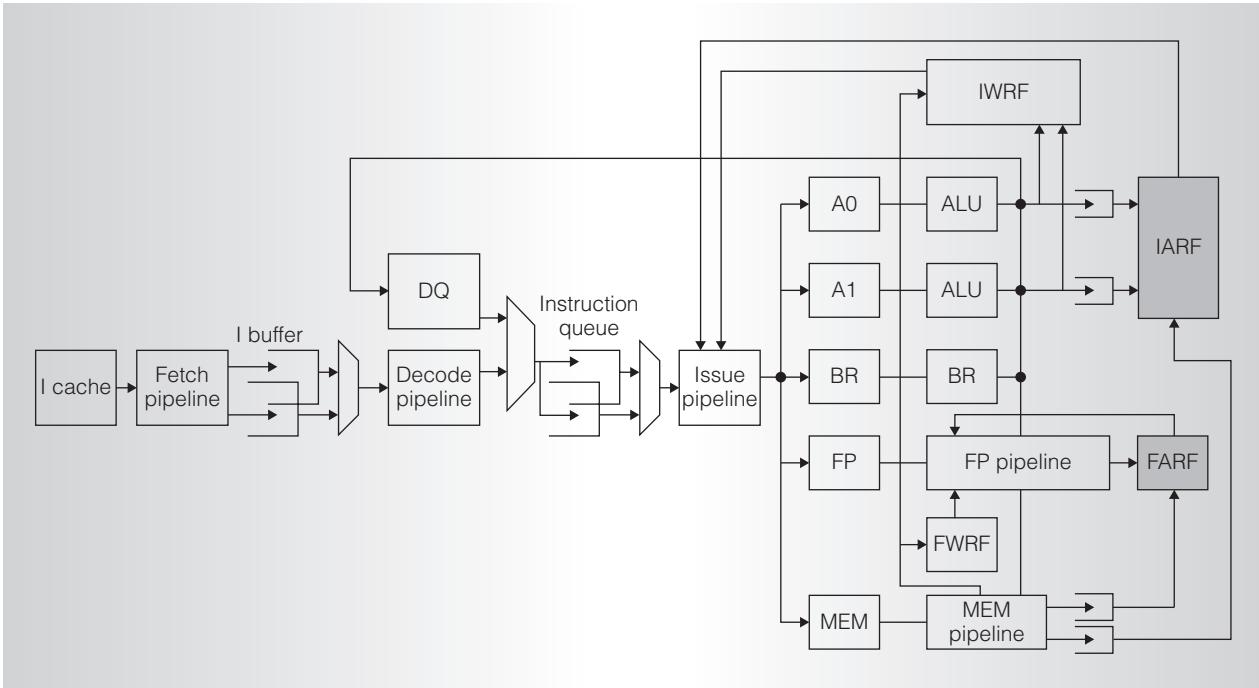


Figure 2. Pipeline overview showing five execution units (ALU, ALU, BR, FP pipeline, MEM pipeline), the integer architectural register file (IARF), the integer working register file (IWRF), the floating-point architectural register file (FARF), and the floating-point working register file (FWRF).

the second-level register file, there is both a main and a speculative copy to support checkpointing and speculation. Although the second-level register file holds about 16 times as much data as the first-level register file, because it is constructed of SRAM, the two structures are almost the same size. To keep track of which register copy to write to—main or speculative—each architectural register has a one-bit pointer associated with it.

The execution unit, which can execute up to five instructions per cycle, consists of five pipelines: simple ALU (A0), complex ALU (A1), branch (BR), floating-point (FP), and memory (MEM). The complex ALU provides all the functionality of the simple ALU and can also handle multicycle instructions such as leading-zero detect. The memory pipeline accepts one load or store instruction per cycle.

Two cores share each 32-Kbyte four-way set-associative data cache. The data cache is dual ported, so both cores sharing it can access it in each cycle. The two cores accessing the same data cache each have a private tag

array, which enables a three-cycle load-use latency. The data cache is accompanied by a 32-entry, two-way skewed-associative way-predicted micro data translation look-aside buffer (micro-DTLB) with full least recently used (LRU) replacement. The way-predictor has a conflict avoidance mechanism that removes conflicts by duplicating entries. Stores are handled by a 32-entry store buffer in each core. The store buffer, which is two-way banked for timing reasons, supports store merging. Stores that hit in the data cache update both the data cache and the L2 cache. Stores that miss in the data cache update the L2 cache and do not allocate in the data cache. To increase memory-level parallelism, stores generate prefetch requests to the L2 and can update the L2 out of program order while maintaining the total store order (TSO)⁷ memory model.

To conserve area, two cores share each floating-point unit in Rock. A floating-point unit contains a floating-point multiply-accumulate unit and a graphics and logical instructions unit that executes the Sparc VIS floating-point instructions. The unit

computes divides and square roots through the iterative Newton-Raphson method. Rock implements new instructions to allow register values to be moved between integer and floating-point registers. The floating-point unit further implements a new Sparc V9 instruction that produces random values with high statistical entropy.

In the pipeline's back end, the commit unit maintains architectural state and controls some of the checkpointed state required for speculation. The commit unit can retire up to four instructions per cycle while the pipeline is not speculating. An unlimited number of instructions can be committed upon successful completion of speculation.

Checkpoint-based architecture

Rock implements a checkpoint-based architecture,^{8–10} in which a checkpoint of the architectural state can be taken on an arbitrary instruction. Once a strand takes a checkpoint, it operates in a speculative mode, in which each write to a register updates a “shadow” speculative copy of the register. If the speculation succeeds, the strand discards the checkpoint, and the speculative copies become the new architectural copies; we call this operation a *join*. If the speculation fails, the strand discards speculative copies of registers and resumes execution from the checkpoint. In addition to enabling EA, SST, and TM, the checkpoint mechanism simplifies trap processing and increases the issue rate by relaxing the issue rules.

Execute ahead

When a Rock system strand encounters a long-latency instruction, it takes a checkpoint and initiates an EA phase. A data cache miss, a micro-DTLB miss, and a divide are examples of long-latency events triggering an EA phase. The destination register of the long-latency instruction is marked as *not available* (NA) in a dedicated NA scoreboard, and the long-latency instruction goes into the DQ for later reexecution. For every subsequent instruction that has at least one source operand that is NA, its destination register is marked as NA and the instruction goes into the DQ. Instructions for which none of the operands is NA are executed, their results are written to

the speculative copy of the destination register, and the NA bit of the destination register is cleared. When the long-latency operation completes, the strand transitions from the EA phase to a replay phase, in which the deferred instructions are read out of the DQ and reexecuted.

In Figure 3a, the strand executes in a non-speculative phase until instruction 1 experiences a data cache miss, at which point the strand takes a checkpoint, defers instruction 1 to the DQ, and marks destination register r8 as NA. When a subsequent instruction that uses r8 as a source register (instruction 2) enters the pipeline, it is also deferred because of the NA bit for r8. As a result, its destination register, r10, is also marked as NA. An independent instruction (3) executes and writes its result into the speculative copy of the register. Because source register r10 for instruction 4 has been marked NA, that instruction also goes to the DQ.

When instruction 1's data is returned, the strand transitions from an EA to a replay phase. The instruction queue stops accepting instructions from the instruction buffer and starts accepting instructions from the DQ. Because instruction 1 is now a cache hit, r8 is no longer marked NA. The chain of dependent instructions (2 and 4) can also execute and write the results to the speculative copies of the registers. With all deferred instructions successfully reexecuted, the strand initiates a join operation to promote the speculative values as the new architectural state. The join operation merely involves manipulating the one-bit pointers that indicate which of the two register copies holds the architectural state values. It is thus a fast and power-efficient operation. The store buffer is notified that the EA phase has joined, and the buffered stores (if any) are allowed to start draining to the L2 cache. The join also notifies the instruction queue to resume accepting instructions from the instruction buffer, and the strand begins a nonspeculative phase.

In the EA process, only the long-latency instruction and its dependents are stored in the DQ; independent instructions update the speculative copies of their destination registers, and these instructions are speculatively retired. As a result, independent

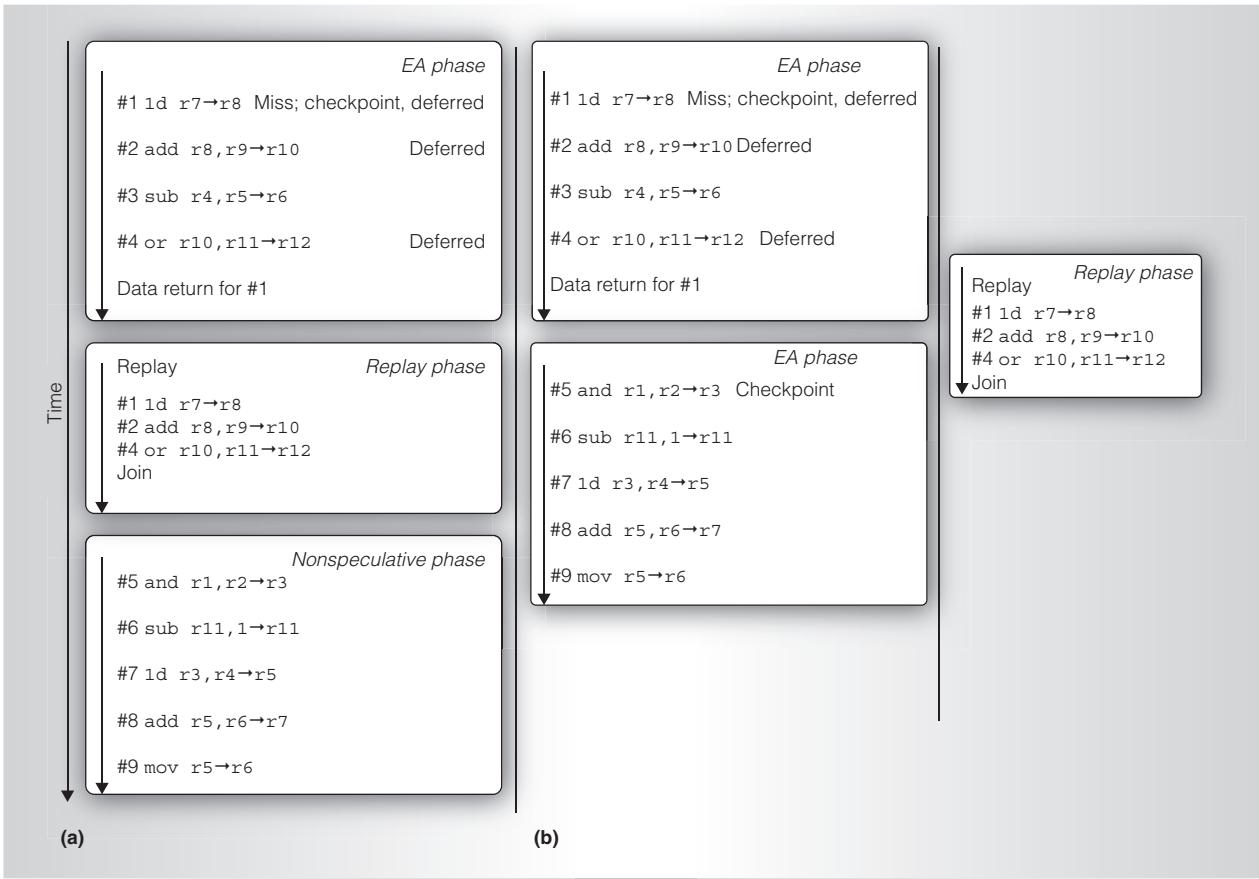


Figure 3. Examples of execute ahead (a) and simultaneous speculative threading (b). The nonspeculative phase in the EA example (a) is replaced in the SST example (b) by an EA phase that executes in parallel with the replay phase.

instructions require no additional architectural resources, so there is no strict architectural limit on the number of independent instructions executed. Therefore, with a relatively small DQ, an EA phase can have more than a thousand instructions in flight.¹¹

In the example in Figure 3a, all instructions that went into the DQ can execute successfully in a single replay phase. However, if additional cache misses or other long-latency instructions arise in the EA phase, some instructions might have one or more NA source registers when they reexecute in the replay phase. In this case, these instructions will return to the DQ, and additional replay phases will run.

If an overflow of the store buffer or DQ occurs during an EA phase, the EA phase continues execution until the long-latency instruction completes. At this point, the EA phase fails and execution resumes from the

checkpoint. In this case, the EA phase execution acts as a hardware scout thread that can encounter additional cache misses, thus providing memory-level parallelism.^{12,13} In addition, the branch predictors are updated so as to reduce branch mispredictions when execution returns to the checkpoint. (Chaudhry et al. have described the performance gains from the hardware scout feature.¹⁴)

Because the EA and replay phases execute instructions out of program order, they must address several register hazards. In particular, write-after-read hazards could occur in which an older instruction reexecuting in a replay phase reads a register that was written by a younger instruction in the preceding EA phase. To prevent such hazards (and similar hazards between successive replay phases), whenever an instruction goes into the DQ, all of the instruction's operands that are

available (not NA) go into the DQ with the instruction.

In addition, write-after-write register hazards could exist in which an older instruction that is reexecuted in a replay phase overwrites a register written by a younger instruction in the preceding EA phase. To prevent this type of hazard (and similar hazards between successive replay phases), older instructions are prevented from writing the speculative copy of a register that has already been written by a younger instruction (although all instructions are allowed to write the copy of their destination register in the first-level register file to provide data forwarding).

The EA and replay phases can speculatively execute loads out of program order. However, to support the TSO memory model, it is essential that no other thread can detect that the loads executed out of order. Thus, a bit per cache line per strand, called an *s-bit*, is added to the data cache. Every load in an EA or replay phase sets the appropriate *s-bit*. If the cache line containing a set *s-bit* is invalidated or evicted, the speculation fails and execution resumes from the checkpoint. Once an EA phase joins or fails, the strand's *s-bits* are cleared.

Simultaneous speculative threading

Rock's ability to dedicate the resources of two strands to a single software thread enables SST, a more aggressive form of speculation. In Figure 3a, when the long-latency instruction (1) completes, the strand stops fetching new instructions and instead replays instructions from the DQ. In contrast, with SST, two instruction queues are dedicated to the single software thread, so one instruction queue continues to receive new instructions from the fetch unit while the other replays instructions from the DQ. As a result, with SST it is possible to start a new EA phase in parallel with execution of the replay phase.

Figure 3b shows an example of SST: When the data returns for instruction 1, one strand begins a replay phase to complete the deferred instructions 1, 2, and 4. In parallel, the other strand takes an additional checkpoint and begins a new EA phase in which it executes new instructions 5 through 9.

In Figure 3b, the second EA phase completed successfully without deferring any of the instructions (5 through 9) to the DQ. However, if one or more of these instructions were deferred (for example, because of a cache miss within this EA phase or a dependence on a cache miss from the earlier EA phase), the strand that performed the first replay phase would then start a new replay phase to complete these deferred instructions. In this manner, it is possible to repeatedly initiate new EA phases with one strand, while performing replay phases for deferred instructions with the other strand.

In addition to enabling the parallel execution of an EA phase and a replay phase, SST reduces the EA phase failures due to resource limitations. Specifically, if the store queue or DQ is nearly full during a first EA phase, the core takes a second checkpoint and a new EA phase begins. In this manner, the first EA phase can complete successfully even if the store queue or DQ overflows during the second EA phase.

In terms of on-chip real estate, no extra state is added to support SST. The bookkeeping structures needed for two-thread EA are simply reused differently to allow for SST. Hence, SST requires virtually no extra area.

Transactional memory

To support TM, Rock uses two new instructions that have been added to the Sparc instruction set—*checkpoint* and *commit*. The *checkpoint* instruction denotes the beginning of a transaction, whereas *commit* denotes the end of the transaction. The *checkpoint* instruction has one parameter, called the *fail-pc*. If the transaction fails for any reason, the instructions within the transaction have no effect on the architectural state (with the exception of nontransactional stores, described later), and the program continues execution from the *fail-pc*. If the transaction succeeds, the architectural state includes the effects of all instructions within the transaction, the loads and stores within the transaction are atomic within the logical memory order, and execution continues after the *commit* instruction.

Rock's implementation of TM relies heavily on the same mechanisms that enable

EA operation. In particular, a strand views the checkpoint instruction as a long-latency instruction (analogous to a load miss) that checkpoints the register state and the fail-pc. If the transaction fails, the strand discards the speculative register updates performed within the transaction, restores the checkpointed state, and continues execution from the fail-pc. If the transaction succeeds, the speculative register updates are committed to architectural state, and the checkpoint is discarded.

Loads within the transaction are executed speculatively and thus set s-bits on the cache lines that are read. The invalidation or replacement of a line with the thread's s-bit set prior to committing the transaction will fail the transaction. As a result, the same mechanism that allows reordering of speculative loads during EA lets the strand make transactional loads appear atomic. Stores within the transaction are placed in the store queue in program order. The addresses of stores are sent to the L2 cache, which then tracks conflicts with loads or stores from other threads. If the L2 cache detects such a conflict, it reports the conflict to the core, which then fails the transaction. When the commit instruction executes, the L2 locks all lines being written by the transaction. Locked lines cannot be read or written by any other threads. This is the point at which other threads view the transaction's loads and stores as being performed, thus guaranteeing atomicity. The stores then drain from the store queue and update the lines, with the last store to each line releasing the lock on that line. The support for locking lines stored to by a committed transaction is the primary hardware mechanism that we added to Rock to implement TM.

Rock's TM support primarily targets efficient execution of moderately sized transactions that fit within the hardware resources. As an example, all stores in a transaction are buffered in the store queue, so the store queue size places a limit on the size of a successful transaction.

Along with the checkpoint and commit instructions, Rock provides additional registers and instructions to aid in debugging and to provide greater control over a transaction's execution. Specifically, Rock provides

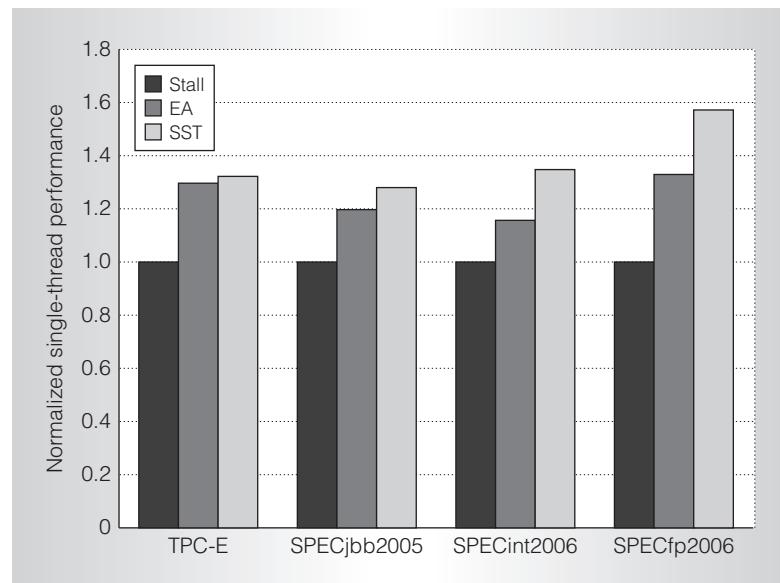


Figure 4. Single-thread performance improvements over stalling in-order processor achieved by speculative techniques execute ahead (EA) and simultaneous speculative threading (SST).

a checkpoint status register that can be read at the fail-pc to determine the cause of a failed transaction. It also provides nontransactional loads and stores.

Nontransactional loads are logically not part of the transaction; they do not cause failure when another thread stores to the location that was read nontransactionally. As a result, nontransactional loads can be used to synchronize the committing of a transaction. For example, it is possible to use nontransactional loads to repeatedly read a *flag* variable until it has been set by another thread.

Nontransactional stores, similarly, are logically not part of the transaction; they update memory even if the transaction fails. Therefore, it is possible to use nontransactional stores to determine what fraction of a transaction executed prior to a failure, thus aiding debugging.

Performance results

Figure 4 shows simulation results for how the speculation techniques we've described improve single-thread performance. We validated the simulator against hardware. The results—for the benchmarks TPC-E, SPECjbb, SPECint2006, and SPECfp2006—are normalized to a stalling

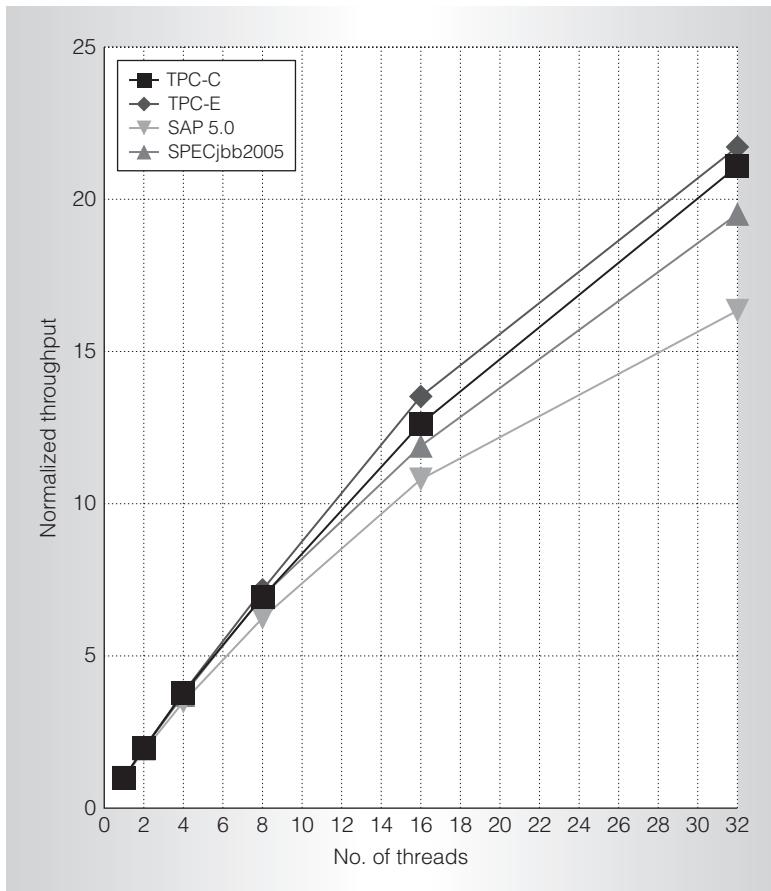


Figure 5. Rock chip throughput for various thread counts, performance normalized to that of a single EA thread.

in-order processor. All of the benchmarks gain significant performance from EA, ranging from 18 percent for SPECint to 35 percent for SPECfp. The two commercial workloads fall in between, with improvements of 20 and 30 percent.

The performance gains due to EA come from the combination of two effects. One is the increased memory-level parallelism that comes from uncovering more cache misses under the original miss. This effect is most pronounced in the two commercial benchmarks (TPC-E and SPECjbb) because of their large working sets and resulting high miss rate in the on-chip caches. The other effect is the overlapping of cache misses with the execution of independent instructions, which is most important in SPECint and SPECfp because of their relatively low miss rate in the shared L2 cache.

In Figure 4, the rightmost bar for each benchmark indicates the performance gain from SST. The main advantage of this technique is that a new EA phase can execute in parallel with the replay phase. As with EA, the gains from SST are most pronounced in the low-miss-rate benchmarks.

Per-thread performance also depends on how heavily the chip resources are shared with other threads. The miss rate increases as more threads share caches, and as a result Rock's speculation techniques become more important. To obtain the results shown in Figure 5, we placed the threads on the chip so as to minimize interference between threads. Four or fewer active threads run in different core clusters, and the only shared resource is the L2 cache. When eight threads are active, threads also share an instruction cache and an instruction fetch unit. When 16 threads are active, all cores are used, and hence threads also share a data cache. When we enabled all 32 threads, two threads share each core.

Figure 5 shows the throughput at various thread counts, with performance normalized to that of a single EA thread (with no other threads running on the chip). Despite the sharing of multiple resources, the chip throughput increases up to 22× as the thread count increases from 1 to 32. Thus, the per-thread performance remains high.

Rock is the first commercial processor to use a checkpoint-based architecture to enable speculative, out-of-order retirement of instructions, and the first commercial processor to implement hardware TM. These novel features raised many unique challenges during the processor's design and verification.

To accommodate Rock's checkpointing and speculation, we made several changes to the Sparc verification infrastructure. We verified the processor's operation using a Sparc functional simulator as a reference model. Unlike other processors, Rock speculatively retires instructions out of order, and then either performs a join or fails speculation. As a result, the intermediate results produced by the speculatively retired instructions could not be directly verified by single-stepping the reference model. Instead, we

had to step the reference model tens or hundreds of times to capture the state after a join, and only at this point could we compare the architectural state to the reference model.

The aggressive speculative design complicated not only correctness debugging, but also performance debugging, because slight changes to the implementation could cause speculation to succeed or fail, thus significantly impacting performance. As a result, accurately modeling the speculation failure conditions in the performance model was essential. Furthermore, failed speculation also complicated correctness debugging by hiding the effects of rare, corner-case bugs that appeared in EA phases, which often failed.

Finally, to verify the TM implementation, we implemented a logical reference memory model that tracked the values of loads and stores from multiple threads. We used this logical reference model to verify the atomicity of the loads and stores within a transaction by applying all of them to the reference model when the transaction logically commits. The greatest challenge was identifying precisely when the transaction logically commits with respect to loads and stores from other threads.

With its novel organization of on-chip resources and key speculation mechanisms, Rock delivers leading-edge throughput performance in combination with high per-thread performance, with a good performance/power ratio. Its hardware implementation of TM will help programmability of multiprocessor systems and improve synchronization-related performance.

MICRO

3. M.D. Hill and M.R. Marty, "Amdahl's Law in the Multicore Era," *Computer*, vol. 41, no. 7, 2008, pp. 33-38.
4. M. Tremblay and S. Chaudhry, "A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread SPARC Processor," *Proc. Int'l Solid-State Circuits Conf. Digest of Technical Papers (ISSCC 08)*, IEEE Press, 2008, pp. 82-83.
5. M. Herlihy and J.E.B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," *Proc. IEEE/ACM Int'l Symp. Computer Architecture (ISCA 93)*, ACM Press, 1993, pp. 289-300.
6. M. Tremblay, B. Joy, and K. Shin, "A Three Dimensional Register File for Superscalar Processors," *Proc. Hawaii Int'l Conf. System Sciences (HICSS 95)*, IEEE CS Press, 1995, pp. 191-201.
7. SPARC Int'l, *The SPARC Architecture Manual (version 9)*, Prentice-Hall, 1994.
8. H. Akkary, R. Rajwar, and S.T. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors," *Proc. IEEE/ACM Int'l Symp. Microarchitecture (MICRO 03)*, IEEE CS Press, 2003, pp. 423-434.
9. W.W. Hwu and Y.N. Patt, "Checkpoint Repair for Out-of-Order Execution Machines," *Proc. IEEE/ACM Int'l Symp. Computer Architecture (ISCA 87)*, ACM Press, 1987, pp. 18-26.
10. S.T. Srinivasan et al., "Continual Flow Pipelines: Achieving Resource-Efficient Latency Tolerance," *IEEE Micro*, vol. 24, no. 6, 2004, pp. 62-73.
11. A. Cristal et al., "Out-of-Order Commit Processors," *Proc. IEEE Int'l Symp. High-Performance Computer Architectures (HPCA 04)*, IEEE CS Press, 2004, pp. 48-59.
12. J. Dundas and T. Mudge, "Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss," *Proc. Int'l Conf. Supercomputing (ICS 97)*, ACM Press, 1997, pp. 68-75.
13. O. Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors," *Proc. IEEE Int'l Symp. High-Performance Computer Architecture (HPCA 03)*, IEEE CS Press, 2003, pp. 129-140.
14. S. Chaudhry et al., "High-Performance Throughput Computing," *IEEE Micro*, vol. 25, no. 3, 2005, pp. 32-45.

Acknowledgments

It is an honor to present this work on behalf of the talented Rock team. Rock would not have been possible without their devotion and hard work.

References

1. P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro*, vol. 25, no. 2, 2005, pp. 21-29.
2. G.M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *Proc. AFIPS Conf.*, vol. 30, AFIPS Press, 1967, pp. 483-485.
3. M.D. Hill and M.R. Marty, "Amdahl's Law in the Multicore Era," *Computer*, vol. 41, no. 7, 2008, pp. 33-38.
4. M. Tremblay and S. Chaudhry, "A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread SPARC Processor," *Proc. Int'l Solid-State Circuits Conf. Digest of Technical Papers (ISSCC 08)*, IEEE Press, 2008, pp. 82-83.
5. M. Herlihy and J.E.B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," *Proc. IEEE/ACM Int'l Symp. Computer Architecture (ISCA 93)*, ACM Press, 1993, pp. 289-300.
6. M. Tremblay, B. Joy, and K. Shin, "A Three Dimensional Register File for Superscalar Processors," *Proc. Hawaii Int'l Conf. System Sciences (HICSS 95)*, IEEE CS Press, 1995, pp. 191-201.
7. SPARC Int'l, *The SPARC Architecture Manual (version 9)*, Prentice-Hall, 1994.
8. H. Akkary, R. Rajwar, and S.T. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors," *Proc. IEEE/ACM Int'l Symp. Microarchitecture (MICRO 03)*, IEEE CS Press, 2003, pp. 423-434.
9. W.W. Hwu and Y.N. Patt, "Checkpoint Repair for Out-of-Order Execution Machines," *Proc. IEEE/ACM Int'l Symp. Computer Architecture (ISCA 87)*, ACM Press, 1987, pp. 18-26.
10. S.T. Srinivasan et al., "Continual Flow Pipelines: Achieving Resource-Efficient Latency Tolerance," *IEEE Micro*, vol. 24, no. 6, 2004, pp. 62-73.
11. A. Cristal et al., "Out-of-Order Commit Processors," *Proc. IEEE Int'l Symp. High-Performance Computer Architectures (HPCA 04)*, IEEE CS Press, 2004, pp. 48-59.
12. J. Dundas and T. Mudge, "Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss," *Proc. Int'l Conf. Supercomputing (ICS 97)*, ACM Press, 1997, pp. 68-75.
13. O. Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors," *Proc. IEEE Int'l Symp. High-Performance Computer Architecture (HPCA 03)*, IEEE CS Press, 2003, pp. 129-140.
14. S. Chaudhry et al., "High-Performance Throughput Computing," *IEEE Micro*, vol. 25, no. 3, 2005, pp. 32-45.

Shailender Chaudhry is a distinguished engineer at Sun Microsystems, where he is chief architect of the Rock processor. His research interests include processor architecture, transactional memory, algorithms, and protocols. Chaudhry has an MS in computer engineering from Syracuse University.

Robert Cypher is a distinguished engineer and a chief system architect at Sun Microsystems. His research interests include parallel computer architecture, processor architecture, transactional memory, and parallel programming. He has a PhD in computer science from the University of Washington.

Magnus Ekman is a staff engineer at Sun Microsystems, where he is a performance architect. His research interests include processor architecture and memory system design. Ekman has a PhD in computer engineering from Chalmers University of Technology. He also has an MS in financial economics from Gothenburg University.

Martin Karlsson is a staff engineer at Sun Microsystems, where he is a core architect working closely with the design team on Rock's microarchitecture. His research interests include transactional memory, instruction scheduling, and checkpoint-based

architectures. He has a PhD in computer science from Uppsala University.

Anders Landin is a distinguished engineer and a chief system architect at Sun Microsystems. His research interests include multiprocessor system architecture, processor architecture, and application and system performance modeling. He has an MS in computer science and engineering from Lund University, Sweden.

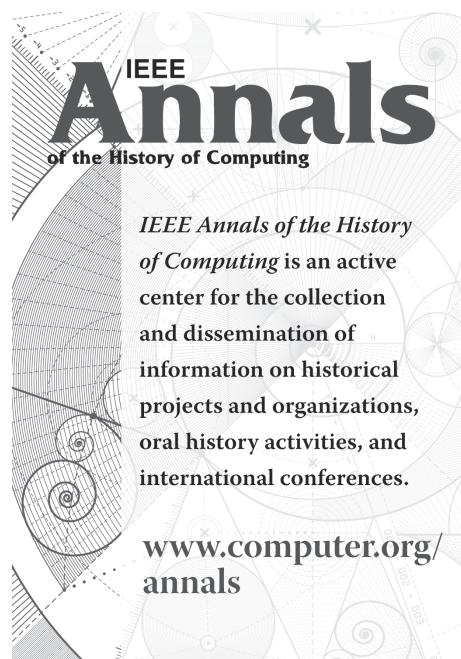
Sherman Yip is a staff engineer at Sun Microsystems, where he is a core architect working closely with the design team on Rock's microarchitecture. His research interests include evaluating new processor features and efficient implementation of architectures that use deep speculation. Yip has a BS in computer science from the University of California at Davis.

Håkan Zeffe is a staff engineer at Sun Microsystems, where he is a system and performance architect. His research interests include parallel computer architecture, coherence, memory consistency, and system software. He has a PhD in computer science from Uppsala University.

Marc Tremblay is a Sun Fellow, senior vice president, and chief technology officer for Sun's Microelectronics Group. In this role, Tremblay sets future directions for Sun's processor and system roadmap. His mission has been to move the entire Sparc processor product line to the throughput computing paradigm, incorporating techniques he has helped develop over the years—including multicores, multithreading, speculative multithreading, scout threading, and transactional memory.

Direct questions and comments about this article to Martin Karlsson, Sun Microsystems, 4180 Network Circle, Mailstop SCA18-211, Santa Clara, CA 95054; martin.karlsson@sun.com.

For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/csdl>.



Performance Evaluation of Intel® Transactional Synchronization Extensions for High-Performance Computing

Richard M. Yoo[†]
richard.m.yoo@intel.com

Konrad Lai[‡]
konrad.lai@intel.com

[†]Parallel Computing Laboratory
Intel Labs
Santa Clara, CA 95054

Christopher J. Hughes[†]
christopher.j.hughes@intel.com

Ravi Rajwar[‡]
ravi.rajwar@intel.com

[‡]Intel Architecture Development Group
Intel Architecture Group
Hillsboro, OR 97124

ABSTRACT

Intel has recently introduced Intel® Transactional Synchronization Extensions (Intel® TSX) in the Intel 4th Generation Core™ Processors. With Intel TSX, a processor can dynamically determine whether threads need to serialize through lock-protected critical sections. In this paper, we evaluate the first hardware implementation of Intel TSX using a set of high-performance computing (HPC) workloads, and demonstrate that applying Intel TSX to these workloads can provide significant performance improvements. On a set of real-world HPC workloads, applying Intel TSX provides an average speedup of 1.41x. When applied to a parallel user-level TCP/IP stack, Intel TSX provides 1.31x average bandwidth improvement on network intensive applications. We also demonstrate the ease with which we were able to apply Intel TSX to the various workloads.

Categories and Subject Descriptors

B.8.2 [Hardware]: Performance and Reliability—*performance analysis and design aids*; C.1.4 [Computer Systems Organization]: Processor Architectures—*parallel architectures*; D.1.3 [Software]: Programming Techniques—*concurrent programming*

General Terms

Performance, Measurement

Keywords

Transactional Memory, High-Performance Computing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SC13, November 17–21, 2013, Denver, CO, USA.

Copyright is held by the owner/author(s). Publication rights licensed to ACM. ACM 978-1-4503-2378-9/13/11 ...\$15.00.
<http://dx.doi.org/10.1145/2503210.2503232>

1. INTRODUCTION

Due to limits in technology scaling, software developers have come to rely on thread-level parallelism to obtain sustainable performance improvement. However, except for the case where the computation is massively parallel (e.g., data-parallel applications), performance of threaded applications is often limited by how inter-thread synchronization is performed. For example, using coarse-grained locks can limit scalability, since the execution of lock-guarded critical sections is inherently serialized. Using fine-grained locks, in contrast, may provide good scalability, but increases locking overheads, and can often lead to subtle bugs.

Various proposals have been made over the years to address the limitations of lock-based synchronization. Lock-free algorithms support concurrent updates to data structures and do not require mutual exclusion through a lock. However, such algorithms are very difficult to write and may not perform as well as their lock-based counterparts. Hardware transactional memory [11] and Oklahoma Update Protocol [26] propose hardware support to simplify the implementation of lock-free data structures. They rely on mechanisms other than locks to ensure forward progress. Speculative Lock Elision [22] proposes hardware support to expose concurrency in lock-based synchronization—the hardware would optimistically execute critical sections without serialization and serialize execution only when necessary. In spite of these proposals, writing correct, high-performance multi-threaded programs remains quite challenging.

Intel has introduced Intel® Transactional Synchronization Extensions (Intel® TSX) in the Intel 4th Generation Core™ Processors [12] to improve the performance of critical sections. With Intel TSX, the hardware can dynamically determine whether threads need to serialize through lock-protected critical sections. Threads perform serialization only if required for correct execution. Hardware can thus expose concurrency that would have been hidden due to unnecessary synchronization.

In this paper we apply Intel TSX to a set of workloads in the high-performance computing (HPC) domain and present the first evaluation of the performance benefits when running on a processor with Intel TSX support. The evaluation incorporates a broad spectrum of workloads, ranging from kernels and benchmark suites to a set of real-world

workloads and a parallel user-level TCP/IP stack. Some of the workloads were originally written to stress test a throughput-oriented processor [24], and have been optimized for the HPC domain. Nevertheless, applying Intel TSX to these workloads provides an average speedup of 1.41x. Applying Intel TSX to a user-level TCP/IP stack provides an average bandwidth improvement of 1.31x on a set of network intensive applications. These results are in contrast to prior work on other commercial implementations that show little to no performance benefits [23, 29], or are limited to small kernels and benchmarks [20, 6, 5].

We demonstrate multiple sources of performance gains. The dynamic avoidance of unnecessary serialization allows more concurrency and improves scalability. In other cases, we reduce the cost of uncontended synchronization operations, and achieve performance gains even in single thread executions. Much of the gain is achieved with changes just in the synchronization library: In some cases, localized changes in the application code results in additional gains.

Section 2 presents a brief overview of Intel TSX. We describe the experimental setup in Section 3 and outline how we apply Intel TSX to the various workloads. Section 4 characterizes Intel TSX using a suite of benchmarks. We evaluate Intel TSX for these benchmarks without any source code changes. In Section 5, we evaluate Intel TSX performance on a set of real-world workloads. We also demonstrate two key techniques to further improve performance: *lockset elision* and *transactional coarsening*. These techniques are useful if one can modify the source code to optimize for performance. In Section 6, we apply Intel TSX to a large-scale software system using a user-level TCP/IP stack and identify some of the challenges, such as condition variables. We discuss related work in Section 7 and conclude in Section 8.

2. INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS

Intel TSX provides developers an instruction set interface to specify critical sections for transactional execution¹. The hardware executes these developer-specified critical sections transactionally, and without explicit synchronization and serialization. If the transactional execution completes successfully (*transactional commit*), then memory operations performed during the transactional execution appear to have occurred instantaneously, when viewed from other processors. However, if the processor cannot complete its transactional execution successfully (*transactional abort*), then the processor discards all transactional updates, restores architectural state, and resumes execution. The execution may then need to serialize through locking if necessary, to ensure forward progress. The mechanisms to track transactional states, detect data conflicts, and commit atomically or rollback transactional states are all implemented in hardware.

Intel TSX provides two software interfaces to specify critical sections. The Hardware Lock Elision (HLE) interface is a legacy compatible instruction set extension (XACQUIRE and XRELEASE prefixes) for programmers who would like to run HLE-enabled software on legacy hardware, but would also like to take advantage of the new transactional execu-

¹Full specifications for Intel TSX can be found in [12]. Enabling and optimization guidelines can also be found in [13]. Additional resources for Intel TSX can be found at <http://www.intel.com/software/tsx>.

tion capabilities on hardware with Intel TSX support. Restricted Transactional Memory (RTM) is a new instruction set extension (comprising the XBEGIN and XEND instructions) for programmers who prefer a more flexible interface than HLE. When an RTM region aborts, architectural state is recovered, and execution restarts non-transactionally at the fallback address provided with the XBEGIN instruction.

Intel TSX does not guarantee that a transactional execution will eventually commit. Numerous architectural and microarchitectural conditions can cause aborts. Examples include data conflicts, exceeding buffering capacity for transactional states, and executing instructions that may always abort (e.g., system calls). Software using RTM instructions should not rely on the Intel TSX execution alone for forward progress. The fallback path not using Intel TSX support must ensure forward progress, and it must be able to run successfully without Intel TSX. Additionally, the transactional path and the fallback path must co-exist without incorrect interactions.

Software using the RTM instructions for lock elision must test the lock during the transactional execution to ensure correct interaction with another thread that may or already has explicitly acquired the lock non-transactionally, and should abort if not free. The software fallback handler should define a policy to retry transactional execution if the lock is not free, and to explicitly acquire the lock if necessary.

When using the Intel TSX instructions to implement lock elision, whether through the HLE or RTM interface, the changes required to enable the use of these instructions are limited to synchronization libraries, and do not require application software changes.

The first implementation of Intel TSX on the 4th Generation Core™ microarchitecture uses the first level (L1) data cache to track transactional states. All tracking and data conflict detection are done at the granularity of a cache line, using physical addresses and the cache coherence protocol. Eviction of a transactionally written line from the data cache will cause a transactional abort. However, evictions of lines that are only transactionally read do not cause an abort; they are moved into a secondary structure for tracking, and may result in an abort at some later time.

3. EXPERIMENTAL SETUP

We use an Intel 4th Generation Core™ processor with Intel TSX support. The processor has 4 cores with 2 Hyper-Threads per core, for a total of 8 threads. Each core has a 32 KB L1 data cache. We use Intel C/C++ compiler for most of our studies, but for those applications utilizing OpenMP, we also use GCC with `libgomp` to precisely control the number of threads. We use inline assembly to emit bytes for Intel TSX instructions, but intrinsics are also available through compiler header files (e.g., `<immintrin.h>`).

Unless otherwise noted, we use thread affinity to bind threads to cores so that as many cores are used as possible—e.g., a 4 thread run will use a single thread on each of the 4 cores, while an 8 thread run will also use 4 cores, but with 2 threads per core. A minimum of 10 executions are averaged to derive statistically meaningful results.

The workloads we use in this paper include transactional memory benchmark suites (CLOMP-TM [23], STAMP [19], and RMS-TM [16]), real-world applications from the HPC domain, and a large-scale software system with a TCP/IP stack running network intensive applications.

These workloads use *synchronization libraries* to coordinate accesses to shared data. An application may either directly call these libraries, or invoke them indirectly through macros or pragmas. These underlying libraries provide multiple mechanisms for synchronizing accesses to shared data. If the shared data being updated is a single memory location (an atomic operation), then the library can achieve this through the use of an atomic instruction (such as LOCK-prefixed instructions in the Intel 64 architecture). For more complex usages, lock-protected critical sections are used.

We apply Intel TSX to the underlying synchronization library, and do not *require* application source changes or annotations. Specifically, in this paper we use the RTM-based interface to elide the relevant critical section locks specified by the synchronization library, and execute the critical section transactionally. If the transactional execution is unsuccessful, then the lock may be explicitly acquired to ensure forward progress. The decision to acquire the lock explicitly is based on the number of times the transactional execution has been tried but failed; for our hardware and workloads, 5 gave the best overall performance. To ensure correct interaction of the transactional execution with other threads that may or already has explicitly acquired the lock, the state of the lock is tested during the transactional execution.

4. EVALUATION ON TRANSACTIONAL MEMORY BENCHMARKS

We start by using CLOMP-TM [23] microbenchmark to characterize Intel TSX performance, and then use the STAMP benchmark suite [19] to see how such performance translates to workload performance. We also apply Intel TSX to RMS-TM [16], and observe how it compares to fine-grained locking, and how it interacts with system calls during a transactional execution.

These transactional memory (TM) benchmark suites use macros and pragmas to invoke the underlying TM library. In addition to a TM implementation, the library also provides a lock-based critical section implementation, equivalent to a conventional lock-based execution model using a global lock. We apply Intel TSX to elide the global lock in the critical section implementation.

4.1 CLOMP-TM Results

In this section we characterize Intel TSX performance using the CLOMP-TM benchmark 1.6 [23]. CLOMP-TM is a synthetic memory access generator that emulates the synchronization characteristics of HPC applications; an unstructured mesh is divided into *partitions*, where each partition is subdivided into *zones*. Threads concurrently modify these zones to update the mesh.

Specifically, each zone is pre-wired to deposit a value to a set of other zones, *scatter zones*, which involves (1) reading the coordinate of a scatter zone, (2) doing some computation, and (3) depositing the new value back to the scatter zone. Since threads may be updating the same zone, value deposits need to be synchronized. Conflict probability can be adjusted by controlling how the zones are wired; and by changing the number of scatters per zone, the amount of work done in a critical section can be adjusted.

To compare Intel TSX performance against existing synchronization methods, we use the benchmark to reproduce the experiment conducted in [23]. Here, threads do not

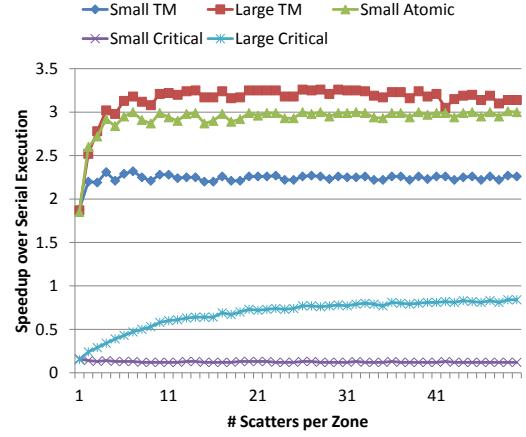


Figure 1: CLOMP-TM benchmark results for 4 threads. Intel TSX version (**Large TM**) outperforms atomic instruction-based version (**Small Atomic**) when at least 3 or 4 scatter zone updates are batched.

contend for memory locations, and to avoid artifacts from L1 data cache sharing among threads, we disable Hyper-Threading (i.e., we use 4 threads).

Figure 1 shows the results. In the figure, **Small Atomic** denotes the case where a LOCK-prefixed instruction is used to enforce atomicity on a single scatter zone value update; this is equivalent to using `#pragma omp atomic`. Likewise, **Small Critical** denotes the use of a lock, equivalent to `#pragma omp critical`, for each scatter zone update. **Large Critical** denotes the case where for each zone, we batch the scatter zone updates (and the accompanying index and value computation code) under a critical section guarded by a single lock. **Small TM** and **Large TM** map the lock-guarded critical sections in **Small Critical** and **Large Critical** into calls into the Intel TSX-enabled synchronization library.

The X-axis denotes the number of scatters for each zone, and at each scatter count, the speedup is against the execution time of the corresponding serial version.

When we synchronize on each scatter zone update, while the LOCK prefix-based version (**Small Atomic**) is the fastest, Intel TSX version (**Small TM**) is not too much worse. The version that uses lock (**Small Critical**), however, performs a lot worse. In contrast, batching a set of scatter zone updates into a single critical section allows better amortization of the synchronization costs. Especially, Intel TSX with batching (**Large TM**) outperforms even **Small Atomic** once we batch at least 3~4 updates. Batching with lock (**Large Critical**), however, suffers from lock contentions, and remains slow.

Compared to the results presented in [23], which requires 5 to 10 updates to be batched before its transactional execution outperforms atomic updates, Intel TSX exhibits lower overhead. However, the scale at which the transactional execution is implemented on [23] is different (16 cores per chip, 4 threads per core). Therefore, a direct comparison cannot be made.

4.2 STAMP Results

STAMP [19] is a benchmark suite extensively used by the transactional memory community. Compared to CLOMP-TM, its workloads are much closer to a realistic application. We use the benchmark suite (0.9.10) to see how Intel TSX performance translates into application performance.

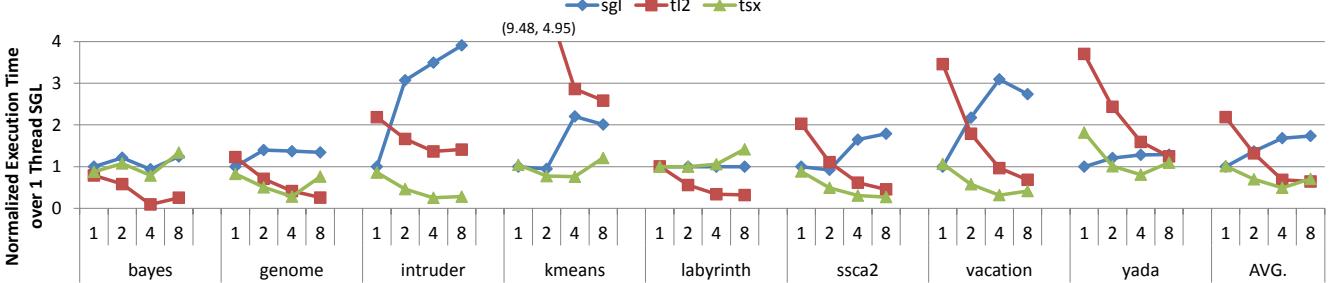


Figure 2: STAMP benchmark results. Intel TSX provides low single thread overhead, while outperforming a software transactional memory (TL2 [7]) in many cases.

Workload	1 thread		2 threads		4 threads		8 threads	
	tl2	tsx	tl2	tsx	tl2	tsx	tl2	tsx
bayes	0	64	1	91	2	89	6	94
genome	0	6	0	11	1	19	1	88
intruder	0	6	32	11	50	31	57	74
kmeans	0	0	15	26	35	71	55	96
labyrinth	0	87	4	95	8	100	16	97
ssca2	0	0	0	1	0	1	0	1
vacation	0	38	2	51	6	52	9	99
yada	0	46	46	68	58	84	65	92

Table 1: Transactional abort rates (%) for the STAMP benchmark suite. Figures of particular interest are highlighted.

Specifically, some STAMP workloads use critical sections with medium/large memory footprint. Memory accesses within a critical section that are required for synchronization correctness have been manually annotated for use by software transactional memory (STM) implementations. STMs rely on instrumenting memory accesses within a transactional region to track transactional reads and writes, and such annotation allows STMs to only track necessary accesses. When using a lock-based execution, these annotated accesses get mapped to regular loads and stores, and are synchronized using the underlying locking mechanism.

Figure 2 shows the execution time of different synchronization schemes implemented by the underlying TM library. We use the *native* input with high contention configuration.

The execution time in the figure is normalized to the single thread execution time of the **sgl** version. **sgl** represents the case where the TM library implements transactional regions as critical sections protected through a single global lock. This scheme forces all transactional regions to serialize, and thus prevents scaling if critical sections comprise a significant fraction of an application’s execution. As expected, with increasing thread count, workloads do not scale.

tl2 represents the performance where the TM library implements transactional regions using the STM included in the benchmark distribution, called TL2 [7]. Overall, by leveraging the annotations to only track crucial memory accesses, STM provides good scalability. However, except for **labyrinth**, it suffers significant single thread overhead. This is because STM has to instrument the annotated memory accesses within a transactional region. On a single-threaded execution, it still pays this overhead, but cannot exploit concurrency to make up for the performance loss.

Intel TSX, however, does not require any instrumentation. In the figure, **tsx** represents the performance where we apply Intel TSX to transactionally elide the single global lock in **sgl**. As can be seen, the Intel TSX-enhanced library shows radically improved single thread performance. Specifically, the performance is comparable to single global lock. With more threads, however, Intel TSX scales significantly

better than single global lock, and in many cases, outperforms STM. With both good single-thread performance and good scalability, a programmer may elect to apply Intel TSX over coarse-grained locks, instead of the conversion effort to fine-grained locks or suffering the high overheads of STM.

Although we provide results on all the workloads for completeness, results on **bayes** and **kmeans** should be discounted, because their execution is strongly dependent on the *order* of various parallel computations—thus, a slower synchronization scheme may result in faster benchmark execution, and vice versa. Specifically, **bayes** utilizes a hill-climbing strategy that combines local and global search [19]. We notice that executions with STM consistently get stuck in local minima, terminating the search earlier but returning inferior results. Similarly, **kmeans** iterates its algorithm until the cluster search converges; we notice that an implementation using Intel TSX always converges faster than STM. We suspect both cases are related to how this specific STM implementation handles floating point variables, and are currently investigating the issue.

Table 1 shows transactional abort percentage that gives more insight into TL2 and Intel TSX behavior. We collect Intel TSX statistics through Linux `perf`. First to note is the non-trivial abort rate of Intel TSX with only one thread. These aborts are mostly due to the effective capacity limit of the set-associative L1 data cache for medium/large critical sections. Hyper-Threading, on the other hand, increases the pressure on the L1, compounding the capacity issue. Thus, in the table, Intel TSX sees significantly higher transactional abort rates with 8 threads than with 4 threads.

Overall, while STAMP tries to cover diverse transactional characteristics, we see that some workloads stopped critical section refinement at medium/large footprint; this would not have been a problem for STMs with virtually unlimited buffer size. STMs also manage to avoid capacity issues through their heavy use of selective annotation (e.g., for **labyrinth**, a 14 MB copy of a global structure to thread-local memory is not annotated). Such manual annotation requires significant effort, especially in a large-scale software system [10], and is not possible with high-level transactional programming constructs [27].

However, due to its low overhead, Intel TSX provides speedup over STM in many cases where its capacity-induced abort rate is reasonable.

4.3 RMS-TM Results

The STAMP benchmark suite is written from the ground-up specifically to evaluate transactional memory implementations. In contrast, RMS-TM [16] adapts a set of existing workloads to use transactional memory. As a result,

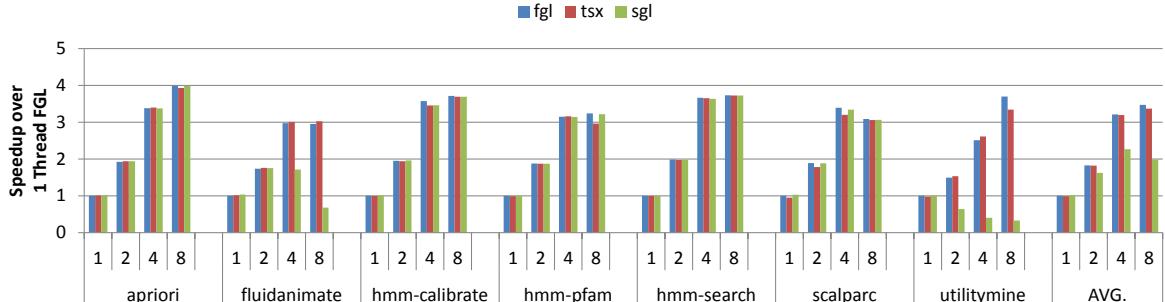


Figure 3: RMS-TM benchmark results. Intel TSX provides comparable performance to fine-grained locking, even when system calls are made during a transactional execution.

Workload	Description	Threading	Sync	Txn Technique		
				Lockset	StatC	DynC
graphCluster	Performs min-cut graph clustering. Kernel 4 of SSCA2 [1].	OpenMP	locks	✓		✓
ua	Unstructured Adaptive (UA) from NAS Parallel Benchmarks suite [9]. Solves a set of heat equations on an adaptive mesh.	OpenMP	atomics		✓	✓
physicsSolver	Uses PSOR to solve a set of 3-D force constraints on groups of blocks.	PThread	locks	✓		✓
nufft	Non-uniform FFT. Baseline reported in [15].	OpenMP	locks			✓
histogram	Parallel image histogram construction.	PThread	atomics			✓
canneal	VLSI router from PARSEC [2]. Performs simulated annealing.	PThread	lock-free			

Table 2: Real-world workloads used in this study. Sync denotes the synchronization mechanism used by the original code. Txn Technique represents the transactional optimization techniques we apply (Lockset = Lockset Elision, StatC = Static Coarsening, and DynC = Dynamic Coarsening).

workloads in RMS-TM exhibit different characteristics from STAMP. Specifically, compared to the medium/large transactions used by STAMP, RMS-TM utilizes fine-grained locks. Therefore, the critical sections exhibit moderate footprint, and as in high-level transactional programming languages [27], no manual annotation is performed. On the other hand, the workloads perform (non-transactional) memory allocation and I/O within critical sections.

We use the RMS-TM benchmark suite to observe how Intel TSX-based synchronization fares in scenarios that are (1) either already optimized (through fine-grained locks) or are (2) not always friendly for transactional execution (i.e., memory allocation and I/O within critical sections). Specifically, we disable the TM-MEM and TM-FILE flags to perform native memory management and file operations within transactional regions, and use the larger input set provided by the benchmark. Figure 3 shows the results.

We compare the speedup of Intel TSX (**tsx**) to fine-grained locking (**fgl**), relative to fine-grained locking with a single thread. With fine-grained locking, RMS-TM workloads scale reasonably well. Using Intel TSX provides comparable performance, demonstrating that memory allocation and I/O within a transactional region do not require special handling, nor necessarily impact performance to a significant degree. As long as such a condition is detected early and the lock is acquired, system calls may not be a performance issue. We also observe that Hyper-Threading has less performance impact on Intel TSX, primarily because the data footprints are moderate as compared to some STAMP workloads.

Figure 3 also shows the performance when we use single global lock (**sgl**) to synchronize all critical sections. Here, macros that mark critical sections are mapped to acquire and release a single global lock, instead. Therefore, the code section that is being synchronized is the same as Intel TSX.

Guarding the critical sections with fine-grained locks or a single global lock does not make significant performance differences, except in **fluidanimate** with lots of small critical sections, and **utilitymine** with more than 30% of execution

spent in critical sections [16]. Here, single global lock fails to scale, while Intel TSX effectively exploits the parallelism, providing comparable performance to fine-grained locking.

5. EVALUATION ON REAL-WORLD WORKLOADS

In this section, we apply and evaluate Intel TSX on a set of real-world workloads. These applications use different types of synchronization mechanisms: lock-based critical sections, atomic operations, and lock-free data structures. Applying Intel TSX to the lock-based critical sections is straightforward. However, we modified the source code so that we could also apply Intel TSX to code regions that use atomic operations and lock-free data structures.

For each workload, we start with a straightforward translation, and then consider optimizations to improve the performance of transactional synchronization.

5.1 Workloads

Table 2 shows the workloads we use for this study. These workloads cover various threading and synchronization schemes, and some represent computations typically found in the HPC domain. In fact, **physicsSolver** and **histogram** were used to stress test a throughput-oriented processor [24].

Specifically, **graphCluster** is Kernel 4 of the SSCA2 benchmark [1]. The **ssca2** workload in STAMP, in contrast, re-implements Kernel 1 for transactional memory from the ground-up. **graphCluster** partitions a graph into clusters while minimizing edge cut costs. Vertices are observed in parallel, and based on the neighbors, they may be added/removed from the cluster. The original application uses per-vertex locks to synchronize updates on the vertex status.

ua is the Unstructured Adaptive workload from NAS Parallel Benchmarks suite [9]. To handle the adaptively refined mesh, **ua** utilizes the Mortar Element Method [9], where thread-local computations performed on *collocation points* are dynamically gathered (i.e., reduced) to *mortars* on a

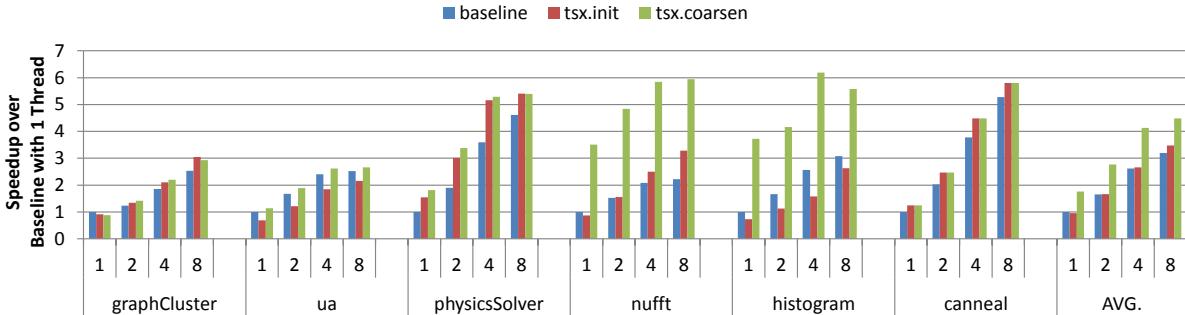


Figure 4: Intel TSX performance on real-world workloads.

global grid. Since the grid dynamically changes, gathers on each mortar require synchronization—the original application uses atomic operations. Reduced values are later scattered back to collocation points.

physicsSolver iteratively resolves constraints between pairs of objects, computing the force exerted on each other to prevent inter-penetration. A key critical section updates the total force exerted on both objects in a given pair. Since each object may be involved in multiple pair-wise interactions, the original application acquires a pair of locks to resolve each constraint, one lock for each object.

nufft performs 3-D non-uniform FFT. We use the baseline version reported in [15]. Specifically, we focus on the *adjoint NUFFT* operator, which reduces a set of non-uniformly spaced spectral indices onto a uniform spectral grid. Since the reduction combines an unpredictable set of non-uniform indices for each grid point, it requires synchronization. The original application uses an array of locks for this.

histogram is an image histogram construction workload. Multiple threads directly update the shared histogram; thus, the updates require synchronization. The original application uses an atomic operation for each bin update. While simple, histogram comprises the core compute of many HPC workloads, such as the two-point correlation function in astrophysics [3], and radix sort [17].

Lastly, **canneal** is a routing workload from PARSEC [2]. It performs simulated annealing, where each thread tries to randomly swap two elements to improve solution quality. To perform this swap in an atomic fashion, the original application implements sophisticated lock-free synchronization.

5.2 Unoptimized Performance Results

For these workloads, we map the lock-based critical sections to calls into the Intel TSX-enhanced synchronization library. The library converts these into critical sections protected by a single global lock, and applies Intel TSX to elide that lock. For atomic operations, we convert the LOCK-prefixed operation into a regular operation, and protect the update using a global lock-based critical section. This is then mapped into a call to the Intel TSX-enabled synchronization library. For the lock-free algorithms, we replace the entire algorithm to use global lock-based critical sections, discarding the atomic instructions and version checking codes from the original algorithm. Intel TSX-based synchronization library is then applied.

Figure 4 shows the results. In the figure, **baseline** represents the performance of the original, lock- and atomics-based code. **tsx.init** represents the performance of the Intel TSX-enabled version. We discuss **tsx.coarsen** later.

Even with straightforward porting, Intel TSX provides a

```

myLock = omp_test_lock(&vLock[w]);
if (myLock) {
    // Non-blocking path
    ... update graph ...
    omp_unset_lock(&vLock[w]);
} else {
    // Blocking path
    omp_set_lock(&vLock[w]);
    ... update graph ...
    omp_unset_lock(&vLock[w]);
}

```

Listing 1: *graphCluster* code example.

noticeable performance improvement. For example, **nufft** has significant concurrency within a critical section hidden under lock contention, which we exploit with transactional execution. For **canneal**, we confirm the observation in [5]: Replacing the complicated lock-free algorithm with a transactional region not only makes the code much simpler, but since some atomic read-time checks can now be removed, provides significant performance improvement as well.

5.2.1 Lockset Elision

For **graphCluster** and **physicsSolver**, on the other hand, we find *lockset elision* to be the key reason for performance improvement. On these workloads, for some critical sections, a set of locks need to be acquired before we can enter the section. Lock acquisitions typically involve costly atomic operations, and the overhead of acquiring a set of locks can be even higher. By replacing a *set of lock acquisitions* with a single transactional begin, we can reduce the overheads—we call this *lockset elision*. We similarly replace the set of lock releases with a single transactional commit.

For example, for **physicsSolver**, we substitute a single transactional begin for a set of two lock acquisitions, one for each object in the pair that is being processed.

Lockset elision can be more subtle. Listing 1 shows an example from **graphCluster**. To synchronize when updating a vertex, the original code utilizes two critical sections, a non-blocking path and a blocking path. Using `omp_test_lock()`², a thread first tries to acquire the lock in a non-blocking fashion. If it succeeds, the thread enters the non-blocking path. If the non-blocking lock acquisition fails, the thread enters the blocking path, and calls `omp_set_lock()` to invoke the blocking lock acquisition code.

When there is little contention for the locks, if the OpenMP implementation has lower lock acquisition overhead for `omp_test_lock()` than `omp_set_lock()`, this code will be

²The OpenMP specification [21] defines that ‘These routines attempt to set an OpenMP lock but do not suspend execution of the task executing the routine.’ That is, the routine is a *try-lock* operation, despite the name.

```

// Compute collocation point indices
i11, i12, i13, i14 = ...;

// Compute mortar indices
ig1, ig2, ig3, ig4 = ...;

#pragma omp atomic
    tmor[ig1] += tx[i11]*third;
#pragma omp atomic
    tmor[ig2] += tx[i12]*third;
#pragma omp atomic
    tmor[ig3] += tx[i13]*third;
#pragma omp atomic
    tmor[ig4] += tx[i14]*third;

```

Listing 2: **ua** code example.

```

for (int y = start_row; y < end_row; y++) {
    for (int x = 0; x < width; x++) {
        if (x % TXN_GRAN == 0)
            TM_BEGIN();
        // Update histogram bin
        UPDATE_BIN(x);
        if (x % TXN_GRAN == TXN_GRAN - 1)
            TM_END();
    }
    src_ptr += width;
}

```

Listing 3: **histogram** code example.

more efficient than using `omp_set_lock()` everywhere. However, under high contention, the code performs an additional lock check just to find that the non-blocking path cannot be taken. These two lock checks can be replaced with a single transactional begin, reducing overheads.

5.2.2 Transactional Coarsening

In contrast, our initial **ua** and **histogram** ports to transactional execution are slower than the original code; these workloads use LOCK prefix-based atomics to perform individually synchronized updates to shared data structures. As seen in Section 4.1, using a critical section to perform an update of a single memory location, whether using Intel TSX or not, has higher overhead than atomics (in Figure 1, compare **Small TM** and **Small Atomic**). On the other hand, batching the updates can amortize overheads; for CLOMP-TM, batching just a few updates allows transactional execution to outperform atomics (i.e., **Large TM** performs better than **Small Atomic** at that point).

We consider *transactional coarsening*, or applying batching to our applications. Specifically, we apply two techniques: *static* and *dynamic coarsening*. *Static coarsening* merges different critical sections (or atomic updates) into one transactional region, at the source code level. Listing 2 shows a code example from **ua**. The code snippet shows the gather phase, where atomics are used to synchronize the reduction on the dynamically changing grid. Our original port places each atomic update into its own transactional region, and removes the now-unnecessary `atomic` pragmas. To amortize the synchronization overhead, we now place all of these updates in a single transactional region.

In contrast, *dynamic coarsening* combines multiple dynamic *instances* of the same transactional region. Listing 3 shows a **histogram** code section amenable to dynamic coarsening. Basically, the code skips some XBEGIN and XEND instances based on the loop index, to combine TXN_GRAN updates into a single transactional region. We explore the best value for TXN_GRAN later.

We expect that dynamic coarsening could be easily ap-

plied with compiler loop unrolling, or through lightweight runtimes. Static coarsening requires more complex static analysis, but is still amenable to automation.

5.3 Optimized Performance Results

We selectively apply transactional coarsening to our workloads. Table 2 shows the specific techniques we apply. In Figure 4, **tsx.coarsen** demonstrates the performance after coarsening. We see that for those two workloads where Intel TSX performed worse than the baseline (i.e., **ua** and **histogram**), Intel TSX now provides a significant speedup. Other workloads benefit from transactional coarsening as well: On average, with 8 threads, Intel TSX provides 1.41x speedup over the baseline. Transactional coarsening may not change scalability noticeably, but by reducing the overhead even in single thread, absolute performance is higher for all thread counts.

5.4 Discussion

5.4.1 Alternative Optimizations

Implementing the equivalent of lockset elision or transactional coarsening using traditional synchronization constructs is difficult. The only static method is to increase the granularity of a critical section. While this could improve performance at low thread counts, coarse-grained locks may not scale. Techniques like multiple granularity locking could be used to dynamically adapt locking granularity, but this is challenging for the programmers to use. Further, it can sometimes be nontrivial to determine prior to a critical section which set of shared locations will be accessed.

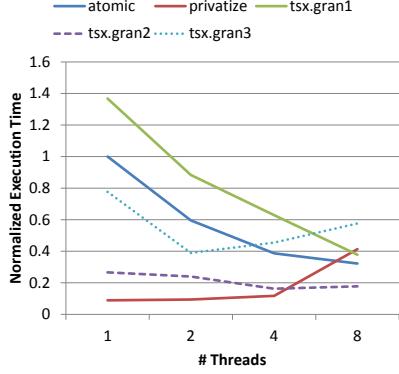
In contrast, with Intel TSX-based synchronization, one can readily convert multiple lock acquisitions or critical sections into a single transactional region, to better amortize the overhead while not significantly sacrificing scalability. Similarly, workloads that perform synchronized updates on multiple memory locations using atomic operations will also benefit by merging them into a single transactional region. In these situations, Intel TSX can improve performance over even fine-grained locking.

5.4.2 Conflict-Free Approaches

Lock- or atomics-based synchronization assumes that concurrent threads perform possibly conflicting updates directly to a shared data structure, and serializes the conflicting accesses. Some HPC applications instead use a *conflict-free* approach, where concurrent updates are *pre-arranged* to be independent. Arranging those updates requires effort and causes overheads, but can skip synchronization for each update. Privatization and barrier-based synchronization are two popular mechanisms used.

Under privatization, each thread (or processor, or node) maintains a *local copy* of the shared data structure on which it performs modifications. Once the updates are done, the application *reduces* the copies, merging the updates. Creating the copies and performing the reduction, however, are overheads that increase with the size of the shared data structure. Privatization therefore scales only if the number of updates is large, relative to the size of the data structure.

With barriers, we eliminate the possibility of conflicts by separating updates that access the same part of the shared data structure into different groups. The application then executes one group of updates at a time in parallel, with a



(a) *histogram* performance results.

Figure 5: Comparison of different synchronization schemes (atomic and mutex, respectively) with a single thread. *tsx.gran** represent different transactional granularities.

barrier between each group to enforce any dependencies between groups. Here, the barriers and formation of the groups are overheads. Also, if the groups are not large enough, this scheme may introduce load imbalance.

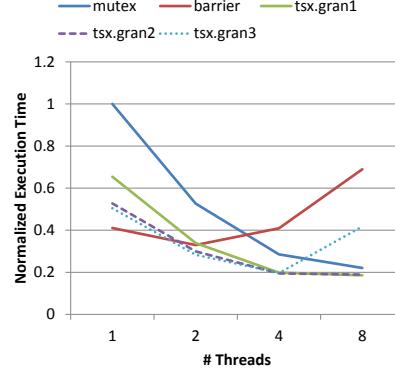
Figure 5 demonstrates two cases where the overheads of these conflict-free schemes outweigh the benefits (i.e., smaller number of synchronizations). In Figure 5a, **atomic** denotes the baseline version of **histogram** that uses atomics for each update. In contrast, **privatize** denotes an alternate version that privatizes the histogram. In Figure 5b, **mutex** denotes the performance of the baseline **physicsSolver**, and **barrier** denotes an implementation that uses barrier-based synchronization. For this workload, we omit the time for forming the groups of independent tasks, since those groups are used repeatedly, amortizing the overhead. In both graphs, **tsx.gran*** denote the performance of Intel TSX across varying transactional granularities. Execution time is normalized to that of the baseline with one thread.

In these experiments, while both privatization and barriers provide good performance with low thread counts, they do not scale. For **histogram**, the number of histogram bins is large relative to the number of items being binned. Therefore, the reduction overhead eventually dominates the execution time. For **physicsSolver**, the input scene has a few objects with many updates, causing large load imbalance.

In these cases, an approach with more synchronization is faster. At 8 threads, even locks and atomics outperform the conflict-free approaches. This may not be true in all applications, nor for all inputs and input parameters—the best performing scheme will depend on several factors, including the overhead for a single synchronization operation. However, as Figure 5 (and our earlier results) demonstrates, Intel TSX can reduce such overheads; and as the overhead decreases, conflict-free approaches look relatively worse. Combined with the promise of significantly easier parallelization, Intel TSX makes an approach with more synchronization more attractive.

5.4.3 Choosing the Right Granularity

Figure 5 also shows workload performance sensitivity to transactional coarsening (the **tsx.gran*** lines—a larger number indicates coarser transactional regions). In general, coarsening improves performance by better amortizing the transactional overheads. However, as the transactional region/footprint gets larger, it becomes more prone to conflicts. Thus, we expect (and observe) a performance inflection point



(b) *physicsSolver* performance results.

Execution time is normalized to baseline (*atomic* and *mutex*, respectively) with a single thread. *tsx.gran** represent different transactional granularities.

as we increase transactional granularity; e.g., in Figure 5b, at 8 threads, largest granularity (**tsx.gran3**) does not provide the best performance. A hardware or runtime-assisted approach to dynamically adjust transactional coarsening could be necessary.

6. EVALUATION IN LARGE-SCALE SOFTWARE SYSTEMS

In the previous section, we focused on improving individual workload performance through Intel TSX. In this section we explore how Intel TSX can be applied in a large software framework, to benefit a larger set of workloads that utilize the framework. In particular, we apply Intel TSX to a parallel user-level TCP/IP stack, and measure the performance of some network intensive applications. Findings should be beneficial to OS, hypervisor, and library development.

6.1 Case Study: User-Level TCP/IP Stack

Version 3.0 of the PARSEC benchmark suite³ includes a multithreaded user-level TCP/IP stack, which is a user-level port of a BSD network stack. In the stack, all the synchronization constructs—locks, condition variables, etc.—and routines are implemented in a single locking module, which acts as a wrapper to the underlying PThread library.

We replace the PThread library with an Intel TSX-enabled synchronization library that uses RTM *instructions* to elide the locks; in this scenario, we use the existing critical section locks in the stack. Instruction-based specification of a transactional region does not require lock acquisition and release to be in the same code scope. As reported in [28, 25], however, using scoped transactional programming constructs to achieve the same can be non-trivial. By enhancing the locking module with Intel TSX, all the workloads utilizing this TCP/IP stack can take advantage of Intel TSX without any changes to the workload code.

One significant challenge we faced, however, is the interaction with condition variables. For reference, Listings 4 and 5 show the PThread condition variable wait and signal routines, respectively. Since we substitute `pthread_mutex_lock()` with XBEGIN, in Listing 4, when a thread finds it needs to call `pthread_cond_wait()`, it does not hold a PThread lock to call the function with.

With Intel TSX, a relatively straightforward workaround would be to just unconditionally abort the transactional exe-

³<http://parsec.cs.princeton.edu/parsec3-doc.htm>

```

pthread_mutex_lock(&lock);

while (monitor state not true) {
    // Wait till condition met
    pthread_cond_wait(&lock, &cond);
}

pthread_mutex_unlock(&lock);

```

Listing 4: PThread condition variable wait routine.

```

pthread_mutex_lock(&lock);
... update monitor state to true ...
// Signal waiting thread
pthread_cond_signal(&cond);

pthread_mutex_unlock(&lock);

```

Listing 5: PThread condition variable signal routine.

cution upon encountering `pthread_cond_wait()`, and to acquire the lock. Once a thread acquires the lock in the fallback handler, it could use the lock to manipulate the condition variable. However, transactional aborts could limit the performance benefits from Intel TSX-based synchronization.

The signaling thread, on the other hand, experiences similar issues in Listing 5, since calling `pthread_cond_signal()` may lead to a system call, which would abort the transactional execution.

Since PThread condition variables are tightly coupled with the locking mechanism, applying Intel TSX to locks requires handling the condition variables as well. Such issues with condition variables have also been reported in other case studies of large-scale software systems [30, 28, 25].

Therefore, we implement a transactional execution-aware condition variable [8]. Instead of the PThread condition variables, this implementation uses Linux `futex`, which does not require holding a lock. Specifically, to reduce transactional aborts, a thread tries to commit partial results when it finds the need to wait on a condition variable. Once it commits, the thread calls `futex` to atomically put itself on the waiters list. The signaling thread, in contrast, registers a callback if it finds the need to signal a condition variable. Upon a transactional commit, the thread will execute the callback to update the `futex`. Once the waiting thread resumes, it starts the transactional execution again. We next compare the performance of the abovementioned implementation options for condition variables.

6.2 Performance Results and Analysis

We evaluate different implementations of the TCP/IP stack on three PARSEC workloads that leverage the stack. These workloads are organized in a client-server fashion, where the client sends the input data over the network, and the server compresses or analyzes the data.

Figure 6 shows the performance of the workloads, normalized to `mutex`, the original TCP/IP stack implementation using PThread locks and condition variables. In the graph, we report the server-side read bandwidth, since it lies on the critical path of the execution. For accurate measurements, workloads with pipeline parallelism (i.e., `netferret` and `netdedup`) have been set to execute the input stages in full before executing the rest of the pipeline.

Our first Intel TSX-based implementation, `tsx.abort`, is the version where we apply Intel TSX to transactionally execute critical sections, but abort the transactional execution

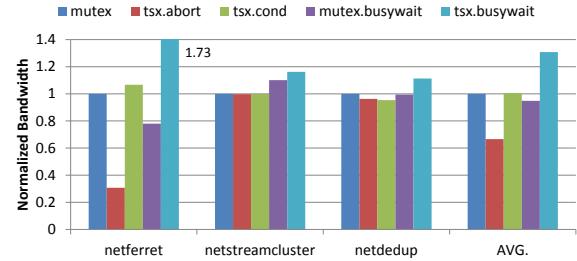


Figure 6: Intel TSX performance on user-level TCP/IP stack. Reports server-side read bandwidth.

```

pthread_mutex_lock(&lock);

while (monitor state not true) {
    // Busy-wait till condition met
    pthread_mutex_unlock(&lock);
    pthread_mutex_lock(&lock);
}

pthread_mutex_unlock(&lock);

```

Listing 6: Busy-wait substitution for conditional wait.

whenever we have to update a condition variable. The performance drops drastically on `netferret`, since the workload sends/receives many small packets over the network.

Our second implementation, `tsx.cond`, uses the transactional execution-aware condition variable. This implementation has much better performance on `netferret` than `tsx.abort`, and even provides some benefit over `mutex`. However, the other workloads observe little benefit, so the average performance is very similar to `mutex`.

With performance analysis tools, we identify there exists certain delay to putting a thread to sleep and waking it up, and while Intel TSX speeds up the rest of the code, this delay dominates the critical path of the network stack. As a last resort, we replace the wait routine in Listing 4 with busy waiting, as shown in Listing 6. While crude, this waiting conforms to the blocking monitor semantics [18].

In Figure 6, `mutex.busywait` and `tsx.busywait` show the performance of such busy waiting with PThread locks and Intel TSX, respectively. As can be seen, the Intel TSX-enabled stack with busy waiting provides significant performance improvement on all the workloads, albeit with some wasted CPU cycles and energy. This demonstrates that increased speculation via *redundant execution*, i.e., spinning, can translate into application level performance.

7. RELATED WORK

Other commercial designs for transactional execution exists. Sun Microsystems® announced transactional execution capability in its Rock [6] processor. However, it was not made commercially available. IBM® introduced transactional support to its Blue Gene®/Q line of supercomputers [29], later extending it to System z® mainframes [14]. Vega processors from Azul Systems® also had hardware support for transactional execution. This was used to elide locks in the Java™ stack [4]. In general, industrial implementations (1) detect transactional conflicts at the time of access, and (2) buffer modifications on the on-chip storage (e.g., caches). Such designs incur the least modifications to the existing cache coherence and core designs.

These designs, however, differ in where they buffer speculative updates, and whether they provide register check-

pointing. For example, Blue Gene/Q utilizes the L2 cache, Rock [6] and System z utilize store queues (store caches), and the first Intel TSX implementation uses the L1 data cache to buffer speculative writes. The choice of buffering location has microarchitectural implications, such as the capacity for speculative states, latency of commits, and messaging between caches. Intel TSX, Blue Gene/Q, and System z have sufficient buffering capacity to handle moderate-sized transactional regions, except for Rock, which can hold 32 lines. For Rock, System z, and Intel TSX, hardware provides register checkpointing, while Blue Gene/Q relies on software. Such difference may have been a factor in the reported Blue Gene/Q performance results [23].

8. CONCLUSION

We describe Intel Transactional Synchronization Extensions and show that the first implementation has significant performance potential. Using a set of transactional memory benchmark suites running on a processor with Intel TSX support, we first demonstrate that Intel TSX has low overheads. Next, we show that on a set of real-world, high-performance computing workloads, Intel TSX provides 1.41x average speedup over lock- and atomics-based implementations. Finally, we apply Intel TSX-based synchronization to a parallel user-level TCP/IP stack. We observe an average of 1.31x bandwidth improvement on a set of network intensive applications.

Through our work with the benchmarks and applications, we also developed techniques to best utilize Intel TSX. In particular, lockset elision and transactional coarsening provide significant benefits. We also encountered a significant challenge in making condition variables transactional execution-aware. While we present a solution that provides good performance, library-level support for transactional condition variables may be necessary.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. We also thank Pradeep Dubey, Ronak Singhal, Joseph Curley, Justin Gottschlich, and Tatiana Shpeisman for their feedback on the paper.

10. REFERENCES

- [1] D. A. Bader and K. Madduri. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In *Proceedings of the 12th International Conference on High Performance Computing*, pages 465–476, 2005.
- [2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.
- [3] J. Chhugani, C. Kim, H. Shukla, J. Park, P. Dubey, J. Shalf, and H. D. Simon. Billion-particle SIMD-friendly two-point correlation on large-scale HPC cluster systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1:1–1:11, 2012.
- [4] C. Click. Azul’s experience with hardware transactional memory. In *HP Labs Bay Area Transactional Memory Workshop*, 2009.
- [5] D. Dice, Y. Lev, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 325–334, 2010.
- [6] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–168, 2009.
- [7] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 2006 International Conference on Distributed Computing*, pages 194–208.
- [8] P. Dudnik and M. M. Swift. Condition variables and transactional memory: Problem or opportunity? In *the 4th ACM SIGPLAN Workshop on Transactional Computing*, 2009.
- [9] H. Feng, R. F. V. der Wijngaart, R. Biswas, and C. Mavriplis. Unstructured Adaptive (UA) NAS parallel benchmark, version 1.0. Technical report, NASA Technical Report NAS-04-006, 2004.
- [10] V. Gajinov, F. Zyulkyarov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. QuakeTM: Parallelizing a complex sequential application using transactional memory. In *Proceedings of the 23rd International Conference on Supercomputing*, pages 126–135, 2009.
- [11] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 1993 Annual International Symposium on Computer Architecture*, pages 289–300.
- [12] Intel Corporation. Intel architecture instruction set extensions programming reference. Chapter 8: Intel transactional synchronization extensions. 2012.
- [13] Intel Corporation. Intel 64 and IA-32 architectures optimization reference manual. Chapter 12: Intel TSX recommendations. 2013.
- [14] C. Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for IBM System z. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 25–36, 2012.
- [15] D. Kalamkar, J. Trzasko, S. Sridharan, M. Smelyanskiy, D. Kim, A. Manduca, Y. Shu, M. Bernstein, B. Kaul, and P. Dubey. High performance non-uniform FFT on modern x86-based multi-core systems. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium*, pages 449–460, 2012.
- [16] G. Kestor, V. Karakostas, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. RMS-TM: A comprehensive benchmark suite for transactional memory systems. In *Proceedings of the 2nd Joint WOSP/SIPEW International Conference on Performance Engineering*, pages 335–346, 2011.
- [17] C. Kim, J. Park, N. Satish, H. Lee, P. Dubey, and J. Chhugani. CloudRAMSort: Fast and efficient

- large-scale distributed RAM sort on shared-nothing cluster. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 841–850.
- [18] B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Commun. ACM*, 23(2):105–117, 1980.
 - [19] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the 2008 IEEE International Symposium on Workload Characterization*, pages 35–46.
 - [20] M. Mitran and V. Vokhshoori. Evaluating the zEC12 transactional execution facility. *IBM Systems Magazine*, 2012.
 - [21] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 3.1, 2011.
 - [22] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 294–305, 2001.
 - [23] M. Schindewolf, B. Bihari, J. Gyllenhaal, M. Schulz, A. Wang, and W. Karl. What scientific applications can benefit from hardware transactional memory? In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 90:1–90:11, 2012.
 - [24] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 papers*, pages 18:1–18:15.
 - [25] A. Skyrme and N. Rodriguez. From locks to transactional memory: Lessons learned from porting a real-world application. In *the 8th ACM SIGPLAN Workshop on Transactional Computing*, 2013.
 - [26] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. Multiple reservations and the Oklahoma update. *Parallel Distributed Technology: Systems Applications, IEEE*, 1(4):58–71, 1993.
 - [27] Transactional Memory Specification Drafting Group. Draft specification of transactional language constructs for C++, version: 1.1. 2012.
 - [28] T. Vyas, Y. Liu, and M. Spear. Transactionalizing legacy code: An experience report using GCC and memcached. In *the 8th ACM SIGPLAN Workshop on Transactional Computing*, 2013.
 - [29] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 127–136, 2012.
 - [30] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. S. Lee. Kicking the tires of software transactional memory: Why the going gets tough. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 265–274, 2008.

Notice and Disclaimers

Intel and Intel Core are trademarks of Intel Corporation in the U.S. and/or other countries. Sun Microsystems and Java are registered trademarks of Oracle and/or its affiliates. IBM, Blue Gene/Q, and System z are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Azul Systems is a trademark of Azul Systems, Inc. in the United States and other countries. Other names and brands may be claimed as the property of others.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to <http://www.intel.com/performance>.

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804.

Design and Performance Evaluation of NUMA-Aware RDMA-Based End-to-End Data Transfer Systems

Yufei Ren

Stony Brook University

Stony Brook, New York 11790

yufei.ren@stonybrook.edu

Tan Li

Stony Brook University

Stony Brook, New York 11790

tan.li@stonybrook.edu

Dantong Yu

Brookhaven National

Laboratory

Upton, New York 11973

dtyu@bnl.gov

Shudong Jin

Stony Brook University

Stony Brook, New York 11790

shujin@stonybrook.edu

Thomas Robertazzi

Stony Brook University

Stony Brook, New York 11790

thomas.robertazzi@stonybrook.edu

ABSTRACT

Data-intensive applications place stringent requirements on the performance of both back-end storage systems and front-end network interfaces. However, for ultra high-speed data transfer, for example, at 100 Gbps and higher, the effects of multiple bottlenecks along a full end-to-end path, have not been resolved efficiently. In this paper, we describe our implementation of an end-to-end data transfer software at such high-speeds. At the back-end, we construct a storage area network with the iSCSI protocols, and utilize efficient RDMA technology. At the front-end, we design network communication software to transfer data in parallel, and utilize NUMA techniques to maximize the performance of multiple network interfaces. We demonstrate that our system can deliver the full 100 Gbps end-to-end data transfer throughput. The software product is tested rigorously and demonstrated applicable to supporting various data-intensive applications that constantly move bulk data within and across data centers.

Categories and Subject Descriptors

C.2.2 [Computer Communication Networks]: Network Protocols—*applications*; D.4.8 [Operating Systems]: Performance—*measurements*

General Terms

Design, Performance.

Keywords

Network Protocols, Storage Area Network, Remote Direct Memory Access, Multi-Core Architecture, Non-Uniform Memory Access

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

SC'13 November 17-21, 2013, Denver, CO, USA

Copyright 2013 ACM 978-1-4503-2378-9/13/11 ...\$15.00.

<http://dx.doi.org/10.1145/2503210.2503260>.

1. INTRODUCTION

Various data-intensive applications require ultra high-speed data transfer capability, such as those in data centers, cloud-computing environments, and distributed scientific computing. They frequently need data transfer software to support true end-to-end data and file delivery, i.e., between the storage systems attached to the source and the destination hosts. Figure 1 shows our intuitive example from the Department of Energy's (DOE's) Magellan cloud data centers [4] that are interconnected by the 100 Gbps links of the DOE's Advanced Network Initiative (ANI). Such an architectural layout is often found in the DOE's National Laboratories, for example, three leadership computing facilities hosted at Argonne National Laboratory, Oak Ridge National Laboratory, and the National Energy Research Scientific Computing Center, respectively, and the tier-1 Large Hadron Collider computing facilities at Brookhaven National Laboratory and Fermilab that play a vital role in searching through petascale to exascale experimental data for scientific insights and discoveries [19]. The science programs (climate simulation, astrophysics, high-energy physics, material science, and system biology) at these DOE Laboratories frequently rely on high-performance supercomputers and server clusters, along with back-end storage systems encompassing hundreds of petabyte disk and tape storage, to run computing and data intensive applications, and to move data from experiments and simulations between computing and storage infrastructures and frequently across wide-area networks. Our primary goal is to design and deliver an efficient, extremely high-performance data transfer tool for these computing facilities that share an infrastructural layout similar to that depicted in Figure 1. To scale up data transfer to 100 Gbps and higher, we must overcome at least three different types of bottlenecks along the end-to-end paths that consist of hosts, networks, and storage systems.

First, to overcome the processing bottleneck of individual hosts, multi-core hosts often are employed for ultra high-speed data transfers. As the number of CPU sockets and cores per CPU die grows in the multi-core architecture of modern computer hosts, it becomes increasingly difficult and inefficient to have the same latency in memory access across all CPU cores. A state-of-the-art CPU architecture integrates a memory controller as a core component within

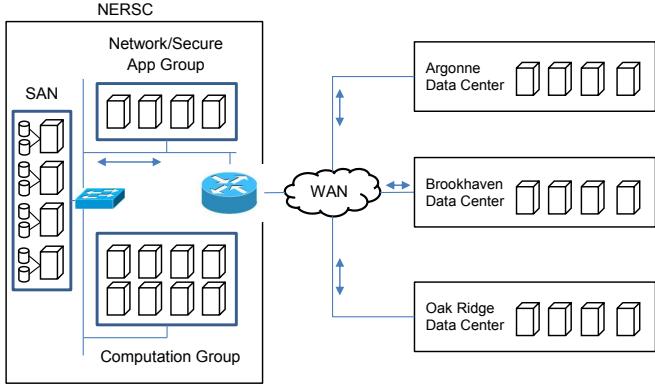


Figure 1: Data transfer and synchronization between data centers. This is an example from the Department of Energy’s (DOE’s) Magellan cloud data centers that are interconnected by the 100 Gbps links of the DOE’s Advanced Network Initiative.

the CPU die, and discards the external memory controller hub, a component that might become a bottleneck in a multi-core architecture [3]. Memory banks in different locations of a motherboard are attached to their corresponding CPUs. Therefore, the access latencies from a specific CPU core to different memory banks are no longer same. With green computing restricting volume and power consumption, vendors turn to the Non-uniform Memory Access (NUMA) model to achieve higher resource density [12, 10, 7]. Although the high-speed connectivity between CPUs greatly facilitates arbitrary memory access, for example QuickPath Interconnect [3] and Hyper Transport [26], an application tuned for local memory access always performs much better than those that are not.

Second, advanced network technologies and protocols are employed to fully utilize the bare-metal bandwidth of ultra high-speed networks, at 100 Gbps and higher, and to eliminate network performance bottlenecks. Remote direct memory access (RDMA) [20] is one of these promising technologies because it can boost the performance of high-speed networks significantly. By enabling network adapters to transfer bulk application memory blocks to or from remote ones, and eliminating data copies in protocol stacks, RDMA achieves low latency and high bandwidth. InfiniBand [14, 13], the original RDMA implementation, dominates the technology market of intra-data center interconnections, while RDMA over Converged Ethernet (RoCE) [9] extends RDMA’s capabilities to the networks between data centers that might be thousands of miles apart. Consequently, RDMA offers an opportunity to assure that large data synchronization and movement within or between data centers for applications to accomplish their routine tasks in a highly efficient manner.

Third, back-end storage systems within a server often become a severe bottleneck due to the low bandwidth of traditional magnetic disks or even recent flash solid-state disks (SSDs). One alternative to overcome this bottleneck of back-end storage systems is to build storage area networks wherein one assembles multiple storage components to provide aggregated bandwidth commensurate with a host’s processing speed and its bare-metal network bandwidth. To configure and adapt high-performance RDMA networking

technology into storage area networks, researchers [11] implemented an iSCSI extension for RDMA (iSER) [16], to enable SCSI commands and objects to be transferred over RDMA-based networks, such as InfiniBand and RoCE.

In this paper, we describe the design, tuning, and performance evaluation of a novel high-speed data transfer system for delivering data at 100 Gbps in an end-to-end fashion. The system utilizes a pair of multi-core front-end hosts (sender and receiver). Our research includes the follows. First, our back-end storage systems use the standard iSER protocol that is configured for high-speed data access. The protocol enables InfiniBand based data delivery from the back-end storage systems to the front-end hosts. This design allows us to eliminate the back-end storage bottleneck with the scalable InfiniBand. Second, between the front-end hosts with multiple network connections, we integrate our RDMA-based file transfer protocol, RFTP [23, 21, 22], into the end-to-end data transfer system, and optimize its performance to maximize bandwidth throughput and minimize host processing overhead. Third, for all hosts along an end-to-end path, we optimize their performance via NUMA tuning. Thus, we minimize the impact of host processing overhead. We note in the current implementation of iSER or RFTP, the NUMA factor is not considered, and we have observed the performance benefit of simple NUMA tuning in this paper. To summarize, our design is the first to achieve 100 Gbps and higher end-to-end real data file transfers between one pair of commodity hosts, and to do so, we have overcome several aforementioned bottlenecks. We evaluate our system comprehensively, using the testbeds that closely resemble the production environments, common in large national laboratories and commercial cloud computing providers. Furthermore, more tests were performed with inter-data center data transfers along long-haul high bandwidth links of over 4000 miles long.

The rest of this paper is organized as follows. In Section 2, we present the background information, and the motivations of our research. We describe our system design in Section 3, and comprehensively evaluate the entire end-to-end system in Section 4. Finally, we offer our conclusions and highlight our contributions.

2. BACKGROUND AND MOTIVATIONS

In this section, we present evidences to show that the advances in hardware technology improve bare-metal performance, but existing software is not developed to take advantage of these advances. Consequently, multiple bottlenecks and issues still exist along end-to-end data transfer paths. Among them, special efforts are needed to improve the efficiency of memory access in multi-core systems, along with techniques for hardware acceleration to maximize the capacity of network protocols. A clear understanding of these advances and a subsequent holistic approach to tackle these new issues are necessary since they are not available in the existing software systems.

2.1 Memory Access in NUMA Multi-core Systems

The stubborn speed disparity between the CPU and memory, named the “Memory Wall”, common in the previous single-core architecture era, will continue to exist and even deteriorate with multi-core architecture. As detailed in [29], latency in memory access will be a major bottleneck in the

computer system. Pursuing higher CPU frequency is not sustainable due to the power wall: increasing transistor current leakage leads to uncontrollable power consumption and generates excessive heat that is hard to dissipate. From system architecture aspect, memory latency might partially negate a high CPU clock rate and the associated computing power. As a result, chip designers might well turn to exploring multi-core architectures and pack more cores into a single CPU die. Consequently, the speed imbalance between fast-growing number of CPU cores and memory will become more severe in the multi-core architecture.

The state-of-the-art NUMA architecture introduces a non-uniform hierarchy of memory latency. Most operating systems often provide only standard scheduling methods and shift to applications the burden of NUMA-related scheduling and tuning. Within this paradigm, applications with high performance requirements must be aware of the physical locations of main memory and even peripheral devices, and implement location-aware mapping functions to co-schedule CPU cores, memory, and devices for application threads with the overall goal of reducing the latency and increasing bandwidth in memory access. The NUMA architecture is not proposed to overcome the memory wall problem. However, it offers applications a hardware platform so as to improve their aggregate performance in a multi-core environment via a suitable policy of memory allocation.

2.2 Protocol Offloading

Another technological advance is the hardware protocol offloading to reduce the processing cost of network protocols in computers. For example, there are at least two memory copies for each data packet sent/received by TCP/IP applications. One is between applications and operating system (kernel), and the other is between operating system (kernel) and network interfaces. For high performance computing, data copies limit a system’s overall performance due to inefficient utilization of memory bandwidth and high CPU consumption. Recently, the bandwidth for a single network adapter has reached 40, 56, or even 100 Gbps [5]. Furthermore, a high-end server is often equipped with multiple adapters for load balancing and fault tolerance. Thus, traditional TCP/IP applications may hit the memory wall problem due to the performance penalty resulted from multiple data copies long before reaching the limit of bare-metal network bandwidth. Consequently, adding network capacity does not improve the actual data transfer performance.

For end-to-end data-transfer systems, both back-end storage network and front-end data movement components must reduce data copy operations and avoid the associated performance penalties. The RDMA protocol and its zero-copy techniques efficiently satisfy this requirement since it off-loads network protocol processing directly into hardware and avoids data copies from/to the kernel space. For example, to build a back-end storage system using SAN, Dalessandro *et al.* [11] implemented iSCSI extensions for RDMA (iSER) [16].

2.3 A Motivating Experiment

To illustrate the importance of the aforementioned technology advances to data transfer applications, we describe a simple experiment carried out in our testbed with multi-core NUMA technology. Two IBM X3650 M4 hosts are connected by three pairs of 40 Gbps RoCE connections (RDMA over

Converged Ethernet). Each RoCE adapter is installed into an eight-lane Peripheral Component Interconnect (PCI) Express 3.0 slot. The theoretical maximum bandwidth of the bi-directional network of such a system is 240 Gbps.

First, we measured the maximum memory bandwidth of our hosts. We compiled STREAM [18], the de facto memory bandwidth benchmark, with the OpenMP option enabled to support multi-threaded test. The *Triad* function showed that the peak memory bandwidth for two NUMA nodes is 50 GB/s, or 400 Gbps. For socket-based network applications, there are two data copies for each network operation at each end of a TCP/IP session. Therefore, the maximum TCP/IP bandwidth that the system can support is 200 Gbps.

Then, we tested TCP/IP stack performance via *iperf* [1] to assess the maximum bi-directional end-to-end bandwidth offered by this testbed. With the default setting, iperf uses only a small chunk of memory, and reuses the same data in the memory chunk. Since the data is always cached within CPU, and it avoids one memory read access. Under these conditions, the result of iperf’s performance matches that of RDMA-based data transfer because it has the same number of memory accesses as RDMA. However, such a test does not reflect real data transfer applications that need to continuously refill data from memory and back-end storage. To eliminate this cache effect, we purposely enlarged the sender’s buffer to exceed the size of the CPU cache. Since iperf is lightweight in user space, and it only transfers memory data to or receives it from network interfaces, most CPU cycles are spent on processing the TCP/IP protocol stack in the kernel space. We captured the percentage of CPU cycles in the kernel through *perf* [6], a Linux kernel profiling tool. During a ten-minute test with the Linux default scheduling policy, the average aggregate bandwidth was 83.5 Gbps. The kernel space and the user space memory copy routines, viz., the *copy_user_generic_string*, consumed about 35% of the overall CPU usage.

For comparison, we optimized “iperf” by tuning the NUMA locality, and repeated this same test. The aggregate bandwidth increased to 91.8 Gbps, about 10% higher than the previous iperf test with the default Linux scheduler.

The experiment and results afford two observations. First, the TCP/IP protocol stack requires multiple data copies and incurs a significant amount of processing overhead that further complicates multi-core memory access, and increases the severity of the memory wall problem. Consequently, the bottleneck of an end-to-end path is host processing operations, rather than network bandwidth. Second, the NUMA memory access incurs additional hardware (CPU cores) cost; for example, latency for synchronization with remote cores can further aggravate the ensuing bottleneck.

The main objective of our work thus is to carefully design an end-to-end data transfer system to eliminate the bottlenecks along a data path. These bottlenecks can comprise transferring hosts, back-end storage systems, and front-end host-to-host network communication channels. We have designed, implemented, and evaluated our RDMA-based system for wide-area data intensive applications [23, 21]. We are aware of several other studies [17, 24, 27] on integrating the RDMA capability into data-transfer applications and evaluating the resulted systems. However, those studies have not yet been validated along the entire end-to-end path, including high performance back-end systems and wide-area network links.

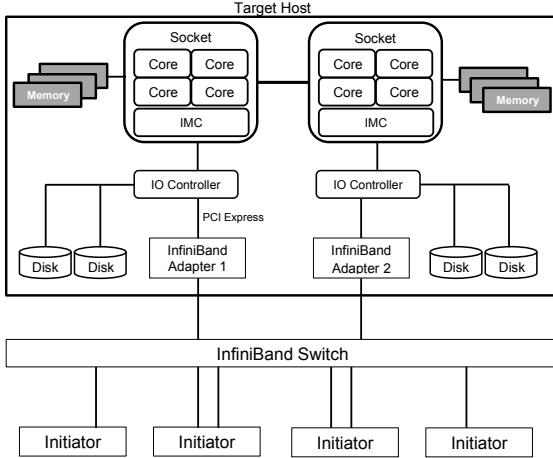


Figure 2: iSER tuning in NUMA architecture with multiple adapters.

3. CHARACTERIZATION OF SYSTEM DESIGN AND NETWORK APPLICATION

In this section, we describe the design of our end-to-end data transfer system. It encompasses one back-end storage system designed as a storage area network, one pair of sending and receiving front-end hosts, and a data transfer application over the entire infrastructure. We detail each component and analyze their performance.

3.1 Back-End Storage Area Network Design

We use the iSER protocol for data communication between a pair of the front-end client and back-end storage server within a storage area network. In this protocol, we follow the definitions in the iSCSI architecture, and call this pair of client and server “initiator” and “target”, respectively. An initiator starts the data transfer process by sending I/O requests to the target that then proactively transfers the data. For example, to handle a read block I/O request sent by the initiator, the target will compose an RDMA *Write* work request to send data to the initiator, while a write I/O request triggers an RDMA *Read* from the target to fetch data from the initiator.

The default target process has a multi-threaded implementation that takes advantages of multi-core architecture to handle multiple I/O requests simultaneously to assure a high throughput. However, with the default setting, the NUMA factor and locations of the Peripheral Component Interconnect (PCI) devices are not considered. There are two possible methods to integrate the NUMA technology into a target. One is to use the *numactl* utility to bind a dedicated target process to each logical NUMA node; the other is to integrate the *libnuma* [15] programming interface into the target implementation. The former needs an explicit, static NUMA policy, while the latter relies on scheduling algorithm for each I/O request. Redesign of iSCSI with the *libnuma* API and libraries is beyond the scope of this paper. We only implement the former solution and here present its effect on NUMA.

We use Linux *tmpfs* as the back-end storage in our prototype system. By adjusting the location of the memory file

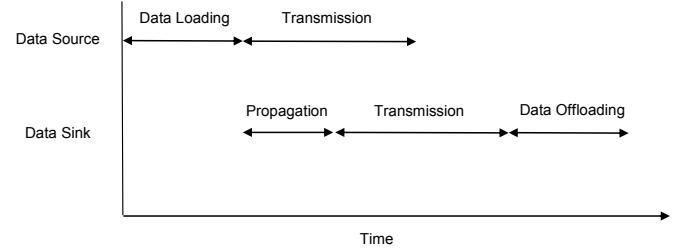


Figure 3: Data block transfer delay breakdown.

with the *mpol* and *remount* options [2], we pin each file into a specified NUMA node memory. Thereafter, we assign each NUMA node to a dedicated target process to handle local I/O requests. Therefore, all NUMA nodes have low latency in accessing local memory, and thereby, the best throughput performance. For a system with multiple adapters, as shown in Figure 2, this choice of design ensures that different I/O requests are handled via different links, hence resulting in the best aggregate performance. The effectiveness of this design will be evaluated by our experiments later.

3.2 RDMA Application Protocol: Cost Analysis and Implementation

Three major components are involved in an end-to-end data transfer application: Data loading, data transmission, and data offloading. Figure 3 shows these components at data source and data sink. Depending on the type of data storage (such as local disks and SANs) and transmission networks (LAN and WAN), the throughput and latency of each component may vary. Any one of the three components can become a bottleneck.

We use our RDMA-based file-transfer protocol, RFTP, to move data within our system. RFTP supports pipelining and parallel operations and configures itself efficiently to utilize system resources and raw network bandwidth. To confirm the benefits of RFTP’s performance, we break down its cost and compare it with the traditional TCP-based data transfer protocols.

To gain insights into the performance of data transfer applications and the efficiency of protocol offloading, we undertake a five-minute test in our local test environment. The data source loads data from `/dev/zero`, and then transfers it over a 40 Gbps RoCE link to the data sink. The latter will dump data into `/dev/null`, i.e., simply discard the received data. Both RDMA-based RFTP and TCP-based iperf accomplish this task at the transfer rate of 39 Gbps. To attain RFTP’s the resource usage of each data transfer thread, we call Linux `getrusage` interface on both the client side and server side. We again analyze the cost of iperf data transfer using the Linux perf tool.

As shown in Figure 4, our RDMA based solution consumes a total of 122% CPU, among which the user space protocol processing uses 56%.¹ In contrast, TCP needs a total of 642% CPU consumption; its kernel space protocol processing accounts for 311%. RDMA’s data-copy overhead is 0% because of zero-copy, while TCP pays 213% on copying data between user space and kernel space. Since the data

¹Note, we use absolute CPU time in the measurement, for example, 122% CPU consumption means the total CPU usage is equivalent to $1.22 \times$ one fully utilized CPU core.

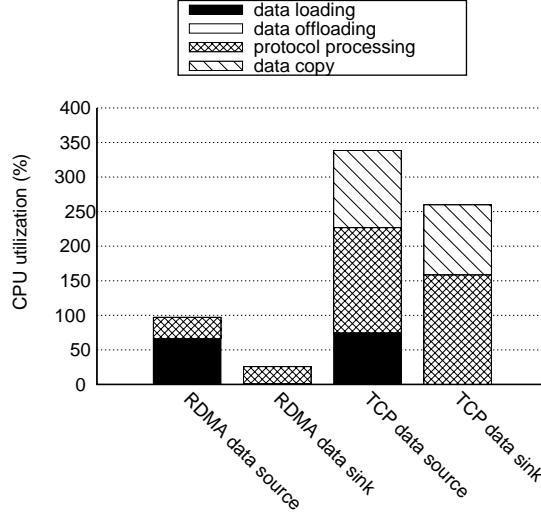


Figure 4: The breakdown of data transfer cost at 40 Gbps rate.

sources load data from `/dev/zero`, the kernel must flush the user memory block with 0s without involving any DMA. In both the RDMA-based and TCP cases, the data sources require about 70% CPU cycles of one core to accomplish the task. Dumping data into `/dev/null` involves less than 1% overhead for the RFTP, while iperf does not offload data. In both cases, the overhead from offloading can be omitted in this experiment. To summarize, through this cost evaluation, RDMA demonstrates its efficient protocol offloading and zero-copy in high-speed data transfer environment.

4. EXPERIMENTAL RESULTS

In this section, we validated the end-to-end data throughput of our software and its CPU consumption, and experimentally confirmed the effectiveness and efficiency of our innovation of NUMA awareness and RDMA hardware offloading. We undertook comprehensive experiments on both the LAN and WAN testbeds. We first describe the testbed’s configurations that consist of the RDMA implementation with both InfiniBand and RoCE interconnection. We evaluated our developed system in three different scenarios: First, the back-end system’s performance with NUMA-aware tuning; second, the application performance in an end-to-end LAN setting; and third, the network performance over a 40 Gbps RoCE long distance path in wide-area networks.

4.1 Testbed Setup

As shown in Figure 5, the LAN experimental system consists of back-end and front-end sections. At the back-end, each initiator or target has two Mellanox InfiniBand adapters, each of which is fourteen data rate (FDR, 56 Gbps) and connected to a Mellanox FDR InfiniBand switch. Therefore, the maximum bandwidth for loading and offloading data is 112 Gbps. At the front-end, three pairs of quad data rate (QDR) 40 Gbps RoCE network cards connect the RFTP client and server with a maximum aggregate bandwidth of 120 Gbps.

In our iSER software setup, we deployed open-iscsi utility version 2.0-872.41 at the initiator host, and SCSI target daemon with version 1.0.31 on the target host. As discussed

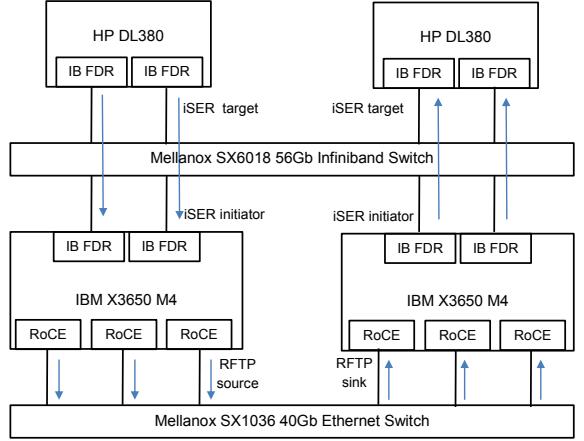


Figure 5: RDMA-based end-to-end system connectivity in LAN.

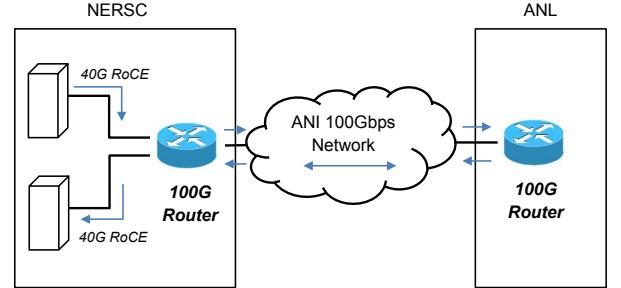


Figure 6: The DOE’s ANI 40 Gbps RoCE WAN between NERSC and ANL. This 4000-mile link is a loopback network from NERSC to ANL and then back to NERSC. The RTT of the link is about 95 milliseconds.

in the previous section, the target incurs the largest fraction of the cost of the iSER protocol processing among all the hosts in our iSER configuration. We first investigated the characteristics of the processes in the target host, including the impact of the NUMA’s binding configuration and the block size of I/O requests.

Initially, we attempted to set up a back-end storage based on Fusion IO’s PCI-based SSD flash drives. However, we found that when applications read or wrote 100 gigabytes data or more continuously to the SSD drive, the thermal-throttling technology of SSDs proactively took actions to throttle the system’s performance to prevent overheating the on-board circuits. These preventive operations degraded the I/O’s performance to about 500MB/s, a severe bottleneck that made our performance evaluation impossible at the speed of 100 Gbps. Thus, in our experiments, we built back-end storage in the main memory of target hosts that dissipates heat much quickly, and so performs consistently over a wider range (from air-conditioned data centers to laboratory environment with normal temperature) of operational temperature. We created six logical units (LUNs) with on the target host, split and load-balanced all I/O requests between the two available InfiniBand links. Each LUN’s size

Table 1: Testbed Configuration

	Front-end LAN	Back-end LAN	Front-end WAN
CPU * Cores	Intel Xeon E5-2660 2.20GHz 16 Cores	Intel Xeon E5-2650 2.00GHz 16 Cores	Intel Xeon E5-2670 2.90GHz 12 Cores
NUMA nodes	2	2	2
Mem(GBytes)	128	384	64
Network Adapters	40 Gbps RoCE QDR	56 Gbps IB FDR	40 Gbps RoCE QDR
OS	CentOS 6.3	CentOS 6.3	Fedora release 17
Kernel Version	2.6.32-279	2.6.32-279	3.4.3-1
OFED Version	MLNX OFED 1.5.3-3.1.0	MLNX OFED 1.5.3-3.1.0	OFED 1.5.4
TCP Congestion Control Algorithm	cubic	cubic	cubic
MTU Size	9000 (RoCE link)	65520 (IB link)	9000
RTT(ms)	0.166	0.144	95

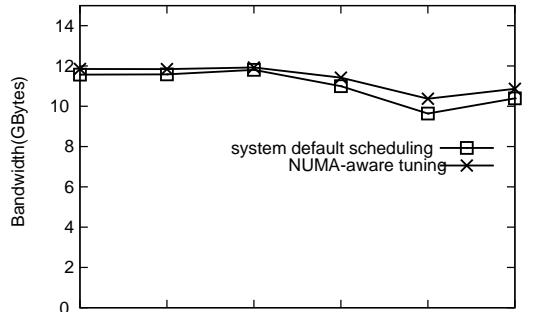
was 50 gigabytes, and the total size of the dataset was 300 gigabytes. The large size memory (we borrowed 768G byte dual in-line memory modules from HP vendors for building the high performance backend storage in our testbed) can be formatted to host any data files and represent a real-world storage solution with low latency and high throughput.

In addition to the testbed configuration within the local network, we also utilized the WAN testbed provided by the DOE's Advanced Networking Initiative (ANI). For these tests, the DOE's ANI supplied a 40 Gbps RoCE wide-area network to evaluate data transfer with RTP over a long-haul 40 Gbps link. This 4000-mile link is a loopback network from the National Energy Research Scientific Computing Center (NERSC) in Oakland, CA to Argonne National Laboratory near Chicago, IL and then back to the NERSC. As shown in Figure 6, the two hosts are connected to wide area networks by a 100 Gbps router, the Alcatel-Lucent Model SR 7750 border. The corresponding routing records on NERSC router are configured as a loopback via ANL's 100 Gbps router [28]. Table 1 lists the detailed configurations for all the hosts, in both the LAN and WAN testbeds described.

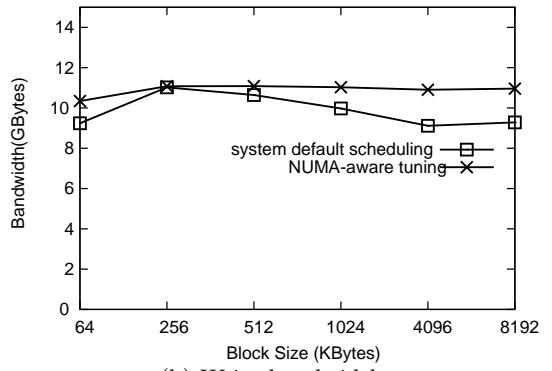
4.2 Evaluation of Memory-Based Storage System Performance

In this set of experiments, we evaluated the improvement in the iSER's performance with our NUMA-aware tuning policy and compared it with the Linux's default scheduling policy.

The iSCSI target host (the back-end storage) is based on Linux tmpfs file system created out of the system's main memory to eliminate the inefficient magnetic disk or SSD bottleneck. The system's memory is large enough (300 GB) to be comparable to a regular SAS disk in terms of the volume, whilst offering a performance a hundred of times faster than that of a magnetic disk. We created six logical units to spread parallel IO requests into different banks of the main memory. To minimize the overhead from the file system, the target exported the back-end memory file as raw devices to allow the initiator to choose any type of file systems as appropriate. We choose the flexible I/O tester [8], fio, as the benchmark software, and each test case lasted five minutes.



(a) Read bandwidth

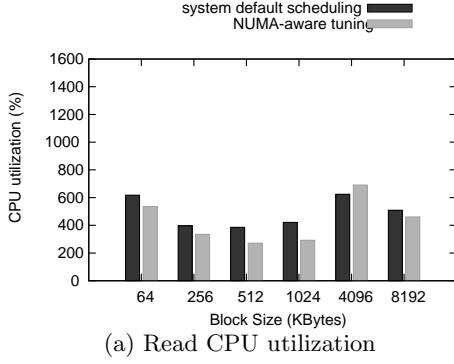


(b) Write bandwidth

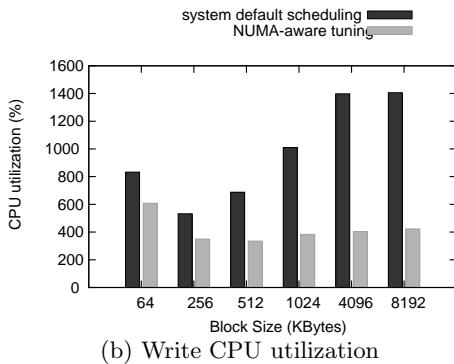
Figure 7: iSER bandwidth comparison between default scheduling and NUMA-tuning.

Figures 7 and 8, respectively, show the bandwidth and CPU consumption, in these tests. In each figure, we separately illustrate the data read and data write performance. To achieve the best performance, multiple I/O threads run simultaneously against each LUN. The gain in performance levels off once the number of threads reaches a certain threshold. Beyond that, too many I/O threads would introduce more contention, and impact the overall performance. In our testbed, we found that the optimal configuration is to use four threads for each LUN.

For read operations, the bandwidth improvement is merely 7.6% with NUMA binding, and neither is the saving on CPU consumption significant. However, for write operations, we observed an improvement in bandwidth up to 19% for the block size larger than 4 megabytes, and using the default Linux binding policy increases CPU consumption threefold. We argue that the main reason of this disparity between read and write operations is the significant overhead of cache coherency and synchronization that is needed for write operations. We note that we performed the read and write operations on a memory-based tmpfs file system. A write request essentially is a memory-write operation, and if it is executed without NUMA-aware tuning, one such operation will invalidate all other data copies in the caches at other NUMA nodes. With NUMA-aware tuning, this invalidation occurs only locally and thus, the overhead is low. When read requests are executed, with or without NUMA-aware tuning, the data copies are always "cached" or "shared" instead of



(a) Read CPU utilization



(b) Write CPU utilization

Figure 8: iSER CPU utilization comparison between default scheduling and NUMA-tuning.

“modified”, and hence, the overhead from cache coherency is minimal.

We also notice that the bandwidth performance of serving read requests out of iSER is slightly better by 7.5% than that of serving write requests. We believe this reflects the better performance of RDMA Write (used by read requests) than RDMA Read (used by write requests). This difference in performance between RDMA Read and Write was revealed in a previous study [23]. When an iSER initiator sends a read request to a target, the target would use RDMA write to write (send) data directly to the memory of the initiator for the actual transfer of data; thus, the observed performance of a read request has a better bandwidth.

4.3 End-to-End Data Transfer Performance

In this section, we describe our tests and our validation of data-transfer applications in an integrated testbed environment and gather its performance with an end-to-end perspective. Our goal is to gain insight into how our application can be adapted to day-to-day real transfer scenarios.

In mimicking a realistic data transfer scenario, we adopted one wherein a transfer application first interacts with file systems via the POSIX interfaces rather than via raw devices. The details of implementing the functions of different file systems are hidden by the POSIX interfaces. Furthermore, our preliminary tests in our testbed demonstrated that the throughput differences among the raw block devices exported by target via the iSER protocol, Linux universal ext4 file system, and the XFS [25] built over the exported block devices via the iSER protocol, are comparable. Since the XFS file system particularly is efficient for parallel I/O

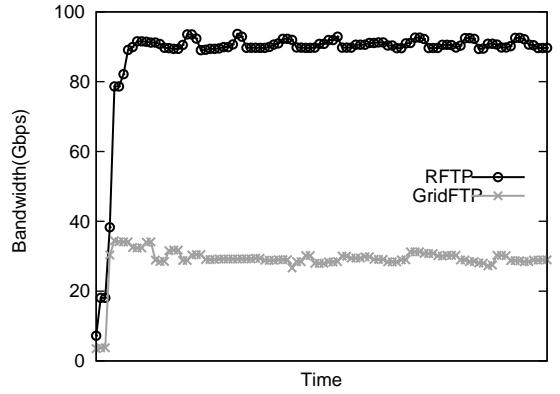


Figure 9: Throughput of end-to-end data transfer over 25 minutes.

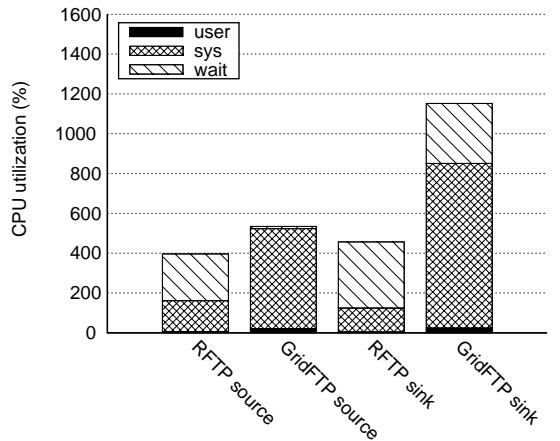


Figure 10: CPU utilization breakdown for RFTP and GridFTP.

and better aligned with our testing requirements, without losing generality, we chose XFS over other file types and formatted the exported block device with XFS from the initiator side.

We combined the back-end system and front-end system to show the end-to-end performance between RFTP and GridFTP, a widely used data transfer tool in high performance computing. To assure a fair comparison between GridFTP and RFTP, and to assure the achievement of the best performance of both applications, and minimize the penalty of remote memory access for each of them, we used numactl to bind the RFTP and GridFTP processes to a specified NUMA node. Figure 9 shows the two applications’ performance during 25-minute-long tests. Our prior “fio” tests revealed that the narrowest section along the end-to-end data transfer path resides on file I/O write operation; its performance is 94.8 Gbps. Therefore, the best performance of the testbed for end-to-end data transfer is 94.8 Gbps. RFTP achieved 91 Gbps, i.e., 96% of the effective bandwidth of the overall system, while GridFTP obtained 29 Gbps, i.e., only 30% of the bandwidth for the following reasons. First, TCP stack processing incurs a substantial overhead, such as data copy between kernel space and user

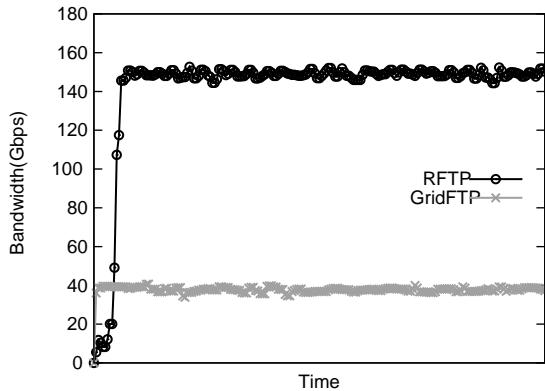


Figure 11: Throughput of bi-directional end-to-end data transfer over 50 minutes.

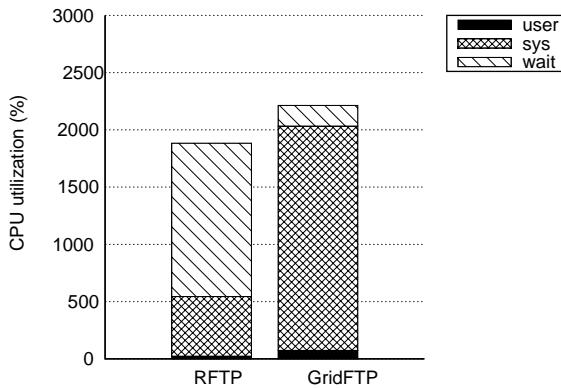


Figure 12: CPU utilization breakdown for RFTP and GridFTP bi-directional test.

space and interrupt handling for an I/O-intensive application. The high “sys” CPU in GridFTP, shown in Figure 10, reveals the high TCP/IP stack-processing cost of GridFTP. Second, GridFTP has a single-threaded design that causes the network to be in an idle state when this thread performs I/O request with long waiting time. Consequently, GridFTP is not able to fully utilize the resources of the whole I/O system. Running multiple processes simultaneously may alleviate this problem, but at the price of higher CPU consumption. Third, without support for direct I/O, GridFTP suffers the I/O cache effect at the front-end sender and receiver hosts.

To further clarify the best end-to-end host performance in terms of bandwidth and CPU consumption, we performed bi-directional data transfer tests with RFTP and GridFTP. In this experiment, we initiated data transfer simultaneously from each end of the end-to-end path to the other end. The configurations are the same as in the previous experiment. We expected that the aggregate bandwidth in the bi-directional tests would have been twice the performance in the unidirectional experiment due to the full duplex property of each component in the transfer path. However, the experimental results did not match with our expectation. Contention for resources increased for bi-directional tests because of more intensive parallel I/O requests to the back-

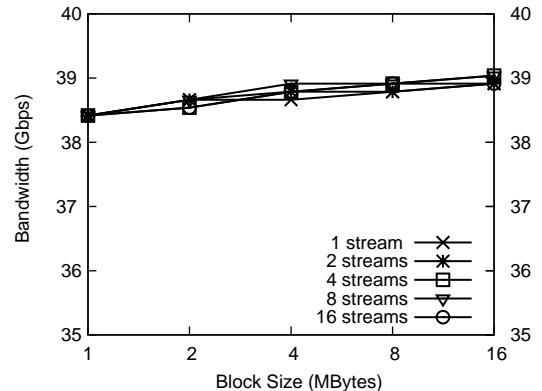


Figure 13: RFTP bandwidth with various block sizes and numbers of streams.

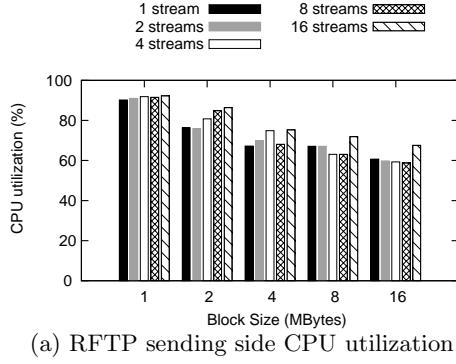
end hosts, memory copies, and higher protocol processing overhead at the front-end hosts.

As shown in Figure 11, our RFTP demonstrated an impressive 83% bandwidth improvement in the bi-directional experiments versus the unidirectional ones, and almost doubles the unidirectional performance (17% less). On the other hand, GridFTP demonstrated only about a 33% improvement due to its high CPU contention, as shown in Figure 12.

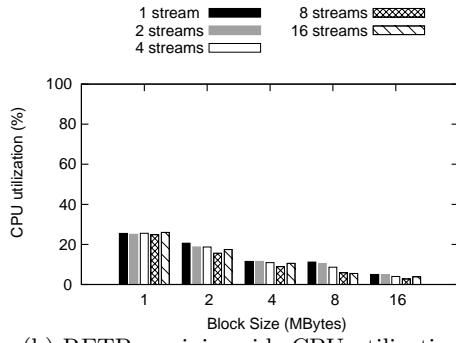
4.4 Experimental Results over 40 Gbps WAN RoCE Link

Long-haul fat links introduce much higher latency than local area networks, and have a large bandwidth delay product (BDP). It is challenging for traditional network protocols (between front-end hosts) to fill up the network pipe at the speed of 100 Gbps and beyond. Here, we evaluate the effectiveness of RFTP in eliminating this bottleneck effect. We ran RFTP over a RoCE link in DOE’s ANI testbed. This link has an RTT of about 95 milliseconds, and the BDP is close to 500 megabytes. We cannot relocate our entire testbed system to the point of presence (POP) site of the DOE’s ANI testbed for a suite of full WAN tests due to the restriction in testbed management and administration in a remote data center. We had to leverage the existing end systems provided by the DOE’s ANI testbed. For our experiments now, we only can conduct memory-to-memory data transfers (between front-end hosts) to show that our RFTP affords a good solution to support end-to-end data transfers in the wide-area networks. We expect that if RFTP performs well over the RoCE link, then our full end-to-end data transfer system would perform equally well if it were deployed in the ANI testbed.

Figure 13 shows the bandwidth performance of RFTP when we use different numbers of parallel data streams. The x-axis shows the block size of data transferred, while the y-axis shows the actual payload data transfer bandwidth, excluding the protocol overhead. We verify that RFTP utilizes 97% of the raw bandwidth of the testbed link due to its efficient design (including the use of proactive feedbacks and asynchronous control message exchanges [23]). A small processing overhead is needed for control messages (for example creating RDMA channels and passing credit tokens), and this overhead decreases in a direct relation to increase in the message block size. The overhead reduction also is



(a) RFTP sending side CPU utilization



(b) RFTP receiving side CPU utilization

Figure 14: RFTP CPU utilization with various block sizes and numbers of streams.

reflected by the lower CPU consumption as shown in Figure 14(a) for the sender, and in Figure 14(b) for the receiver.

As we noted earlier that this wide area data transfer did not involve storage area networks due to the difficulty of deploying and managing them remotely at the NERSC. Our prior experiments show that we can get the maximum performance along each segment of end-to-end LAN and WAN data transfers, i.e., back-end data upload to the front-end system, data transfer over networks, and data offload to the back-end storage system again.

Furthermore, the performance of network data transfer is unaffected by long latency. Based on these observations, we conclude that our RFTP easily can scale up the performance that is commensurate with full-fledged end-to-end distributed testing or production infrastructures consisting of large-scale storage area networks with hundreds or thousands of target servers and long latency inter-data center network links. These infrastructures often are found at the DOE’s National Laboratories and cloud data centers with intensive data transfer requirements.

5. CONCLUSIONS

Modern data centers of scientific computing must transfer and synchronize a large amount of data continuously either locally within themselves or remotely with other data centers for visualization, analysis, and disaster recovery. Accordingly, end-to-end high performance data transfer software must combine efficient design and performance tuning, to eliminate any potential bottlenecks within storage systems, at front-end hosts, and along network communication paths. In this paper we have described a novel design of

such a system that integrates RDMA protocol implementation, multi-core NUMA tuning, and an optimized back-end storage area network.

To demonstrate the efficiency of our solution, we set up testbeds in both LANs and WANs. We studied the processing cost of TCP/IP stack and our RDMA based protocol. We demonstrated the performance benefits of an RDMA based solution adopted by both our RFTP and iSER. We also compared our solution with the high performance GridFTP software in various end-to-end configurations. Our performance evaluation demonstrated that our solution is three times faster than GridFTP. We also validated our protocol design in a 4000-mile network path provided by the Department of Energy’s ANI testbed. These evaluations and studies verified that our solution can achieve remarkable bandwidth utilization of 97% and fully utilize the available hardware capabilities.

Acknowledgments

The authors are grateful to the facility and hardware donation of Mellanox Technologies, Inc., LSI Corp., and Fusion-io, Inc. The authors benefited from numerous technical discussions with Gilad Shainer, Roi Dayan, Erin Filliater, Yaron Haviv, Bill Lee, Dudu Slama, and Todd Wilde, from Mellanox, Brian Tierney, Eric Pouyoul, and Scott Richmond from Lawrence Berkeley National Laboratory, Paul Grun and David McMillen from System Fabric Works, Inc., and Ezra Kissel and Martin Swany from Indiana University. This research is supported by United States Department of Energy, Grant No. DE-SC0003361. The ESnet Advanced Network Initiative (ANI) Testbed, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231. Both contracts are funded through the American Recovery and Reinvestment Act of 2009.

6. REFERENCES

- [1] NLANR/DAST : Iperf - the TCP/UDP bandwidth measurement tool, 2003-2008. <http://iperf.sourceforge.net/>.
- [2] NUMA memory allocation policy for tmpfs, 2006.
- [3] An introduction to the Intel QuickPath Interconnect, January 2009.
- [4] The Magellan report on cloud computing for science. Technical report, 2011.
- [5] New mellanox interconnect to break 100g throughput, June 2012.
- [6] perf : Linux profiling with performance counters, 2012.
- [7] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl’s Law through EPI throttling. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 298–309, May 2005.
- [8] J. Axboe. Flexible I/O Tester.
- [9] D. Cohen, T. Talpey, A. Kanevsky, U. Cummings, M. Krause, R. Recio, D. Crupnicoff, L. Dickman, and P. Grun. Remote Direct Memory Access over the Converged Enhanced Ethernet fabric: Evaluating the options. In *2009 17th IEEE Symposium on High Performance Interconnects (HOTI)*, pages 123–130, 2009.

- [10] P. Conway, N. Kalyanasundaram, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE Micro*, pages 16–29, 2010.
- [11] D. Dalessandro, A. Devulapalli, and P. Wyckoff. iSER storage target for object-based storage devices. In *Proceedings of Fourth International Workshop on Storage Network Architecture and Parallel I/Os*, September 2007.
- [12] U. Drepper. What every programmer should know about memory. September 2007.
- [13] IBTA. Infiniband Trade Association. <http://www.infinibandta.org/>, 2010.
- [14] InfiniBand Trade Association. InfiniBand Architecture Specification. *Release 1.2.1*, 2006.
- [15] A. Kleen. An NUMA API for linux, August 2004.
- [16] M. Ko, M. Chadalapaka, J. Hufford, U. Elzur, H. Shah, and P. Thaler. Internet Small Computer System Interface (iSCSI) Extensions for Remote Direct Memory Access (RDMA). RFC 5046 (Proposed Standard), Oct. 2007.
- [17] P. Lai, H. Subramoni, S. Narravula, A. Mamidala, and D. K. Panda. Designing efficient FTP mechanisms for high performance data-transfer over InfiniBand. In *Proceedings of International Conference on Parallel Processing (ICPP)*, September 2009.
- [18] J. D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. <http://www.cs.virginia.edu/stream/>.
- [19] D. Overbye. Physicists find elusive particle seen as key to universe, July 2012.
- [20] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. A remote direct memory access protocol specification. RFC 5040 (Proposed Standard), Oct. 2007.
- [21] Y. Ren, T. Li, D. Yu, S. Jin, and T. Robertazzi. Middleware support for RDMA-based data transfer in cloud computing. In *Proceedings of High-Performance Grid and Cloud Computing Workshop*, May 2012.
- [22] Y. Ren, T. Li, D. Yu, S. Jin, and T. Robertazzi. Design and testbed evaluation of rdma-based middleware for high-performance data transfer applications. *Journal of Systems and Software*, 86(7):1850 – 1863, 2013.
- [23] Y. Ren, T. Li, D. Yu, S. Jin, T. Robertazzi, B. L. Tierney, and E. Pouyoul. Protocols for wide-area data-intensive applications: design and performance issues. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12*, pages 34:1–34:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.
- [24] H. Subramoni, P. Lai, R. Kettimuthu, and D. K. Panda. High performance data transfer in grid environment using GridFTP over InfiniBand. In *Int'l Symposium on Cluster Computing and the Grid (CCGrid)*, May 2010.
- [25] A. Sweeney. Scalability in the XFS file system. In *Proceedings of USENIX Annual Technical Conference*, pages 1–14, 1996.
- [26] The HyperTransport Consortium. HyperTransport I/O technology overview, June 2004.
- [27] Y. Tian, W. Yu, and J. S. Vetter. RXIO: Design and implementation of high performance RDMA-capable gridftp. *Computers and Electrical Engineering*, 38(3):772 – 784, 2012.
- [28] B. Tierney, E. Kissel, M. Swany, and E. Pouyoul. Efficient data transfer protocols for big data. In *Proceedings of the 8th International Conference on eScience*, October 2012.
- [29] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, Mar. 1995.

The Stanford Dash Multiprocessor

**Daniel Lenoski, James Laudon, Kourosh Gharachorloo,
Wolf-Dietrich Weber, Anoop Gupta, John Hennessy,
Mark Horowitz, and Monica S. Lam
Stanford University**

The Computer Systems Laboratory at Stanford University is developing a shared-memory multiprocessor called Dash (an abbreviation for Directory Architecture for Shared Memory). The fundamental premise behind the architecture is that it is possible to build a scalable high-performance machine with a single address space and coherent caches.

The Dash architecture is scalable in that it achieves linear or near-linear performance growth as the number of processors increases from a few to a few thousand. This performance results from distributing the memory among processing nodes and using a network with scalable bandwidth to connect the nodes. The architecture allows shared data to be cached, thereby significantly reducing the latency of memory accesses and yielding higher processor utilization and higher overall performance. A distributed directory-based protocol provides cache coherence without compromising scalability.

The Dash prototype system is the first operational machine to include a scalable cache-coherence mechanism. The prototype incorporates up to 64 high-performance RISC microprocessors to yield performance up to 1.6 billion instructions per second and 600 million scalar floating point operations per second. The design of the prototype has provided deeper insight into the architectural and implementation challenges that arise in a large-scale machine with a single address space. The prototype will also serve as a platform for studying real applications and software on a large parallel system.

This article begins by describing the overall goals for Dash, the major features of the architecture, and the methods for achieving scalability. Next, we describe the directory-based coherence protocol in detail. We then provide an overview of the prototype machine and the corresponding software support, followed by some

**Directory-based
cache coherence gives
Dash the ease-of-use
of shared-memory
architectures while
maintaining the
scalability of
message-passing
machines.**

preliminary performance numbers. The article concludes with a discussion of related work and the current status of the Dash hardware and software.

Dash project overview

The overall goal of the Dash project is to investigate highly parallel architectures. For these architectures to achieve widespread use, they must run a variety of applications efficiently without imposing excessive programming difficulty. To achieve both high performance and wide applicability, we believe a parallel architecture must provide scalability to support hundreds to thousands of processors, high-performance individual processors, and a single shared address space.

The gap between the computing power of microprocessors and that of the largest supercomputers is shrinking, while the price/performance advantage of microprocessors is increasing. This clearly points to using microprocessors as the compute engines in a multiprocessor. The challenge lies in building a machine that can scale up its performance while maintaining the initial price/performance advantage of the individual processors. Scalability allows a parallel architecture to leverage commodity microprocessors and small-scale multiprocessors to build larger scale machines. These larger machines offer substantially higher performance, which provides the impetus for programmers to port their sequential applications to parallel architectures instead of waiting for the next higher performance uniprocessor.

High-performance processors are important to achieve both high total system performance and general applicability. Using the fastest microprocessors reduces the impact of limited or uneven parallelism inherent in some applications. It also allows a wider set of applications to exhibit acceptable performance with less effort from the programmer.

A single address space enhances the programmability of a parallel machine by reducing the problems of data partitioning and dynamic load distribution, two of the toughest problems in programming parallel machines. The shared address space also improves support for automatically parallelizing compilers, standard operating systems, multipro-

The Dash team

Many graduate students and faculty members contributed to the Dash project. The PhD students are Daniel Lenoski and James Laudon (Dash architecture and hardware design); Kourosh Gharachorloo (Dash architecture and consistency models); Wolf-Dietrich Weber (Dash simulator and scalable directories); Truman Joe (Dash hardware and protocol verification tools); Luis Stevens (operating system); Helen Davis and Stephen Goldschmidt (trace generation tools, synchronization patterns, locality studies); Todd Mowry (evaluation of prefetch operations); Aaron Goldberg and Margaret Martonosi (performance debugging tools); Tom Chanak (mesh routing chip design); Richard Simoni (synthetic load generator and directory studies); Josep Torrellas (sharing patterns in applications); Edward Rothberg, Jaswinder Pal Singh, and Larry Soule (applications and algorithm development). Staff research engineer David Nakahira contributed to the hardware design.

The faculty associated with the project are Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam.

gramming, and incremental tuning of parallel applications — features that make a single-address-space machine much easier to use than a message-passing machine.

Caching of memory, including shared writable data, allows multiprocessors with a single address space to achieve high performance through reduced memory latency. Unfortunately, caching shared data introduces the problem of cache coherence (see the sidebar and accompanying figure).

While hardware support for cache coherence has its costs, it also offers many benefits. Without hardware support, the responsibility for coherence falls to the user or the compiler. Exposing the issue of coherence to the user would lead to a complex programming model, where users might well avoid caching to ease the programming bur-

den. Handling the coherence problem in the compiler is attractive, but currently cannot be done in a way that is competitive with hardware. With hardware-supported cache coherence, the compiler can aggressively optimize programs to reduce latency without having to rely purely on a conservative static dependence analysis.

The major problem with existing cache-coherent shared-address machines is that they have not demonstrated the ability to scale effectively beyond a few high-performance processors. To date, only message-passing machines have shown this ability. We believe that using a directory-based coherence mechanism will permit single-address-space machines to scale as well as message-passing machines, while providing a more flexible and general programming model.

Dash system organization

Most existing multiprocessors with cache coherence rely on snooping to maintain coherence. Unfortunately, snooping schemes distribute the information about which processors are caching which data items among the caches. Thus, straightforward snooping schemes require that all caches see every memory request from every processor. This inherently limits the scalability of these machines because the common bus and the individual processor caches eventually saturate. With today's high-performance RISC processors this saturation can occur with just a few processors.

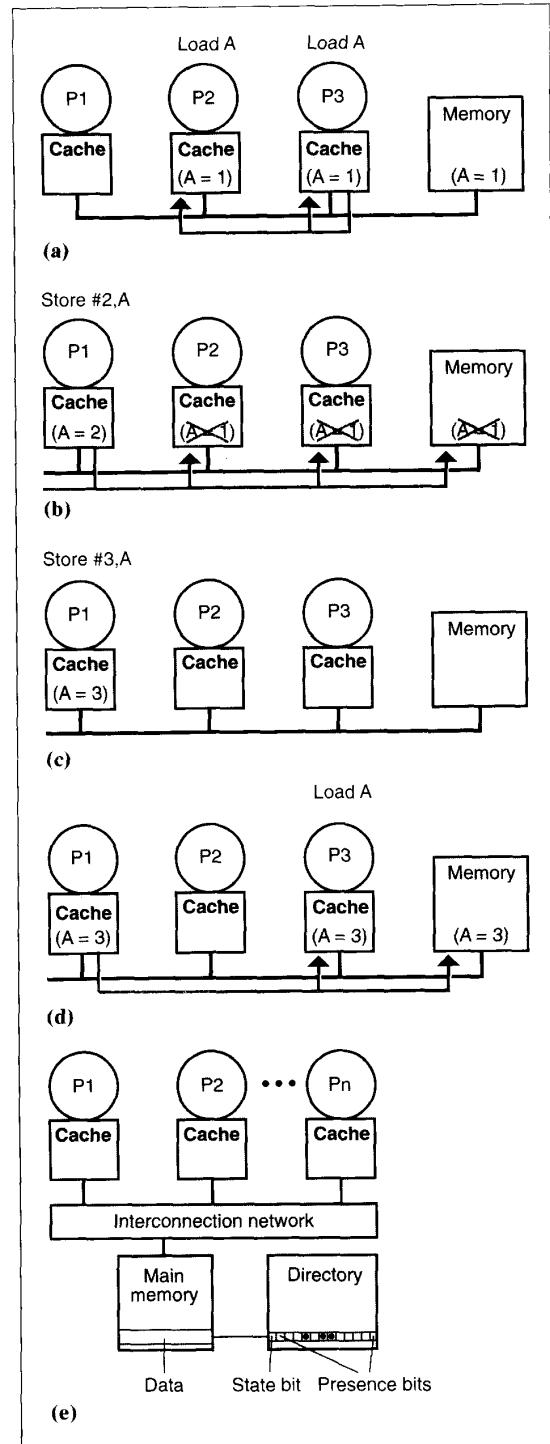
Directory structures avoid the scalability problems of snoopy schemes by removing the need to broadcast every memory request to all processor caches. The directory maintains pointers to the processor caches holding a copy of each memory block. Only the caches with copies can be affected by an access to the memory block, and only those caches need be notified of the access. Thus, the processor caches and interconnect will not saturate due to coherence requests. Furthermore, directory-based coherence is not dependent on any specific interconnection network like the bus used by most snooping schemes. The same scalable, low-latency networks such as Omega networks or k -nary n -cubes used by non-cache-coherent and

Cache coherence

Cache-coherence problems can arise in shared-memory multiprocessors when more than one processor cache holds a copy of a data item (a). Upon a write, these copies must be updated or invalidated (b). Most systems use invalidation since this allows the writing processor to gain exclusive access to the cache line and complete further writes into the cache line without generating external traffic (c). This further complicates coherence since this dirty cache must respond instead of memory on subsequent accesses by other processors (d).

Small-scale multiprocessors frequently use a snoopy cache-coherence protocol,¹ which relies on all caches monitoring the common bus that connects the processors to memory. This monitoring allows caches to independently determine when to invalidate cache lines (b), and when to intervene because they contain the most up-to-date copy of a given location (d). Snoopy schemes do not scale to a large number of processors because the common bus or individual processor caches eventually saturate, since they must process every memory request from every processor.

The directory relieves the processor caches from snooping on memory requests by keeping track of which caches hold each memory block. A simple directory structure first proposed by Censier and Feautrier² has one directory entry per block of memory (e). Each entry contains one presence bit per processor cache. In addition, a state bit indicates whether the block is uncached, shared in multiple caches, or held exclusively by one cache (that is, whether the block is dirty). Using the state and presence bits, the memory can tell which caches need to be invalidated when a location is written (b). Likewise, the directory indicates whether memory's copy of the block is up to date or which cache holds the most recent copy (d). If the memory and directory are partitioned into independent units and connected to the processors by a scalable interconnect, the memory system can provide scalable memory bandwidth.



References

1. J. Archibald and J.-L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. Computer Systems*, Vol. 4, No. 4, Nov. 1986, pp. 273-298.
2. L. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. Computers*, Vol. C-27, No. 12, Dec. 1978, pp. 1,112-1,118.

message-passing machines can be employed.

The concept of directory-based cache coherence is not new. It was first proposed in the late 1970s. However, the

original directory structures were not scalable because they used a centralized directory that quickly became a bottleneck. The Dash architecture overcomes this limitation by partitioning and

distributing the directory and main memory, and by using a new coherence protocol that can suitably exploit distributed directories. In addition, Dash provides several other mechanisms to

reduce and hide the latency of memory operations.

Figure 1 shows Dash's high-level organization. The architecture consists of a number of processing nodes connected through directory controllers to a low-latency interconnection network. Each processing node, or *cluster*, consists of a small number of high-performance processors and a portion of the shared memory interconnected by a bus. Multiprocessing within the cluster can be viewed either as increasing the power of each processing node or as reducing the cost of the directory and network interface by amortizing it over a larger number of processors.

Distributing memory with the processors is essential because it allows the system to exploit locality. All private data and code references, along with some of the shared references, can be made local to the cluster. These references avoid the longer latency of remote references and reduce the bandwidth demands on the global interconnect. Except for the directory memory, the resulting system

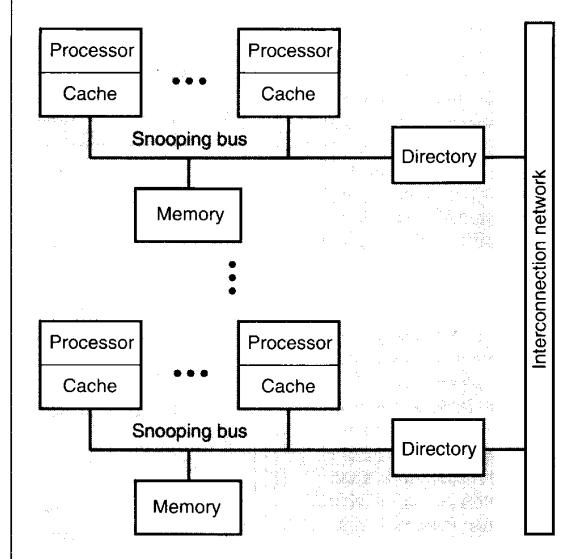


Figure 1. The Dash architecture consists of a set of clusters connected by a general interconnection network. Directory memory contains pointers to the clusters currently caching each memory line.

architecture is similar to many scalable message-passing machines. While not optimized to do so, Dash could emulate such machines with reasonable efficiency.

We have outlined why we believe a single-address-space machine with cache coherence holds the most promise for delivering scalable performance to a wide range of applications. Here, we address the more detailed issues in scaling such a directory-based system. The three primary issues are ensuring that the system provides scalable memory bandwidth, that the costs scale reasonably, and that mechanisms are provided to deal with large memory latencies.

Scalability in a multiprocessor requires the total memory bandwidth to scale linearly with the number of processors. Dash provides scalable bandwidth to data objects residing in local memory by distributing the physical memory among the clusters. For data accesses that must be serviced remotely, Dash uses a scalable interconnection network. Support

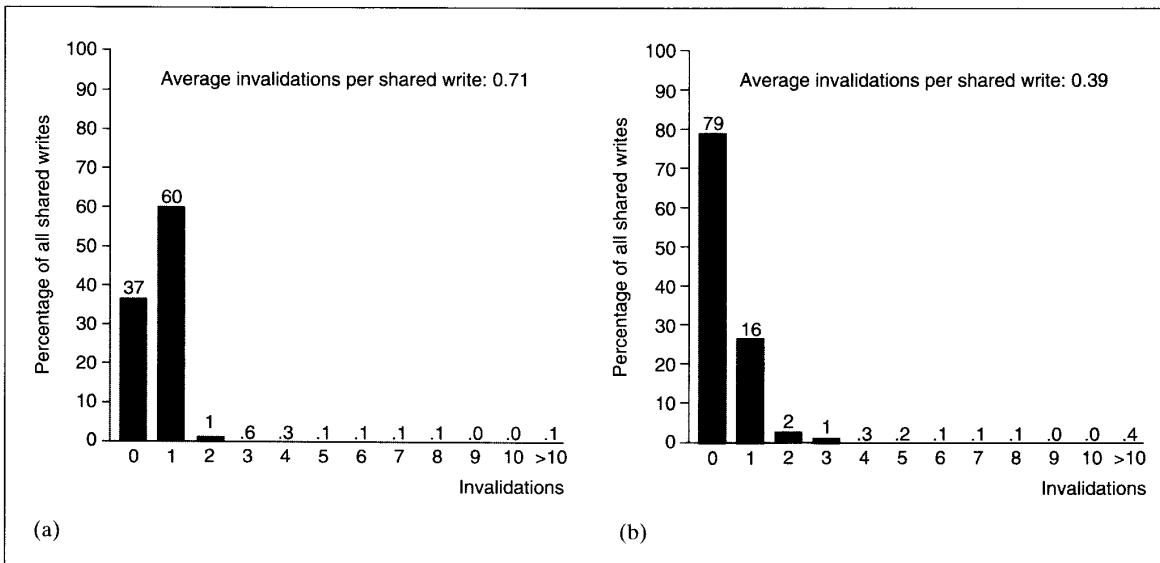


Figure 2. Cache invalidation patterns for MP3D (a) and PThor (b). MP3D uses a particle-based simulation technique to determine the structure of shock waves caused by objects flying at high speed in the upper atmosphere. PThor is a parallel logic simulator based on the Chandy-Misra algorithm.

of coherent caches could potentially compromise the scalability of the network by requiring frequent broadcast messages. The use of directories, however, removes the need for such broadcasts and the coherence traffic consists only of point-to-point messages to clusters that are caching that location. Since these clusters must have originally fetched the data, the coherence traffic will be within some small constant factor of the original data traffic. In fact, since each cached block is usually referenced several times before being invalidated, caching normally reduces overall global traffic significantly.

This discussion of scalability assumes that the accesses are uniformly distributed across the machine. Unfortunately, the uniform access assumption does not always hold for highly contended synchronization objects and for heavily shared data objects. The resulting *hot spots* — concentrated accesses to data from the memory of a single cluster over a short duration of time — can significantly reduce the memory and network throughput. The reduction occurs because the distribution of resources is not exploited as it is under uniform access patterns.

To address hot spots, Dash relies on a combination of hardware and software techniques. For example, Dash provides special extensions to the directory-based protocol to handle synchronization references such as queue-based locks (discussed further in the section, "Support for synchronization"). Furthermore, since Dash allows caching of shared writable data, it avoids many of the data hot spots that occur in other parallel machines that do not permit such caching. For hot spots that cannot be mitigated by caching, some can be removed by the coherence protocol extensions discussed in the section, "Update and deliver operations," while others can only be removed by restructuring at the software level. For example, when using a primitive such as a barrier, it is possible for software to avoid hot spots by gathering and releasing processors through a tree of memory locations.

Regarding system costs, a major scalability concern unique to Dash-like machines is the amount of directory memory required. If the physical memory in the machine grows proportionally with the number of processing nodes, then using a bit-vector to keep track of all

clusters caching a memory block does not scale well. The total amount of directory memory needed is $P^2 \times M/L$ megabits, where P is the number of clusters, M is the megabits of memory per cluster, and L is the cache-line size in bits. Thus, the fraction of memory devoted to keeping directory information grows as P/L . Depending on the machine size, this growth may or may not be tolerable. For example, consider a machine that contains up to 32 clusters of eight processors each and has a cache (memory) line size of 32 bytes. For this machine, the overhead for directory memory is only 12.5 percent of physical memory as the system scales from eight to 256 processors. This is comparable with the overhead of supporting an error-correcting code on memory.

For larger machines, where the overhead would become intolerable, several alternatives exist. First, we can take advantage of the fact that at any given time a memory block is usually cached by a very small number of processors. For example, Figure 2 shows the number of invalidations generated by two applications run on a simulated 32-processor machine. These graphs show that most writes cause invalidations to only a few caches. (We have obtained similar results for a large number of applications.) Consequently, it is possible to replace the complete directory bit-vector by a small number of pointers and to use a limited broadcast of invalidations in the unusual case when the number of pointers is too small. Second, we can take advantage of the fact that most main memory blocks will not be present in any processor's cache, and thus there is no need to provide a dedicated directory entry for every memory block. Studies^{1,2} have shown that a small directory cache performs almost as well as a full directory. These two techniques can be combined to support machines with thousands of processors without undue overhead from directory memory.

The issue of memory access latency also becomes more prominent as an architecture is scaled to a larger number of nodes. There are two complementary approaches for managing latency: methods that reduce latency and mechanisms that help tolerate it. Dash uses both approaches, though our main focus has been to reduce latency as much as possible. Although latency tolerating techniques are important, they often

require additional application parallelism to be effective.

Hardware-coherent caches provide the primary latency reduction mechanism in Dash. Caching shared data significantly reduces the average latency for remote accesses because of the spatial and temporal locality of memory accesses. For references not satisfied by the cache, the coherence protocol attempts to minimize latency, as shown in the next section. Furthermore, as previously mentioned, we can reduce latency by allocating data to memory close to the processors that use it. While average memory latency is reduced, references that correspond to interprocessor communication cannot avoid the inherent latencies of a large machine. In Dash, the latency for these accesses is addressed by a variety of latency hiding mechanisms. These mechanisms range from support of a relaxed memory consistency model to support of nonblocking prefetch operations. These operations are detailed in the sections on "Memory consistency" and "Prefetch operations."

We also expect software to play a critical role in achieving good performance on a highly parallel machine. Obviously, applications need to exhibit good parallelism to exploit the rich computational resources of a large machine. In addition, applications, compilers, and operating systems need to exploit cache and memory locality together with latency hiding techniques to achieve high processor utilization. Applications still benefit from the single address space, however, because only performance-critical code needs to be tuned to the system. Other code can assume a simple uniform memory model.

The Dash cache-coherence protocol

Within the Dash system organization, there is still a great deal of freedom in selecting the specific cache-coherence protocol. This section explains the basic coherence protocol that Dash uses for normal read and write operations, then outlines the resulting memory consistency model visible to the programmer and compiler. Finally, it details extensions to the protocol that support latency hiding and efficient synchronization.

Memory hierarchy. Dash implements an invalidation-based cache-coherence protocol. A memory location may be in one of three states:

- *uncached* — not cached by any cluster;
- *shared* — in an unmodified state in the caches of one or more clusters; or
- *dirty* — modified in a single cache of some cluster.

The directory keeps the summary information for each memory block, specifying its state and the clusters that are caching it.

The Dash memory system can be logically broken into four levels of hierarchy, as illustrated in Figure 3. The first level is the processor's cache. This cache is designed to match the processor speed and support snooping from the bus. A request that cannot be serviced by the processor's cache is sent to the second level in the hierarchy, the *local cluster*. This level includes the other processors' caches within the requesting processor's cluster. If the data is locally cached, the request can be serviced within the cluster. Otherwise, the request is sent to the *home cluster* level. The home level consists of the cluster that contains the directory and physical memory for a given memory address. For many accesses (for example, most private data references), the local and home cluster are the same, and the hierarchy collapses to three levels. In general, however, a request will travel through the interconnection network to the home cluster. The home cluster can usually satisfy the request immediately, but if the directory entry is in a dirty state, or in shared state when the requesting processor requests exclusive access, the fourth level must also be accessed. The *remote cluster* level for a memory block consists of the clusters marked by the directory as holding a copy of the block.

To illustrate the directory protocol, first consider how a processor read traverses the memory hierarchy:

- *Processor level* — If the requested location is present in the processor's cache, the cache simply supplies the data. Otherwise, the request goes to the local cluster level.

- *Local cluster level* — If the data resides within one of the other caches within the local cluster, the data is sup-

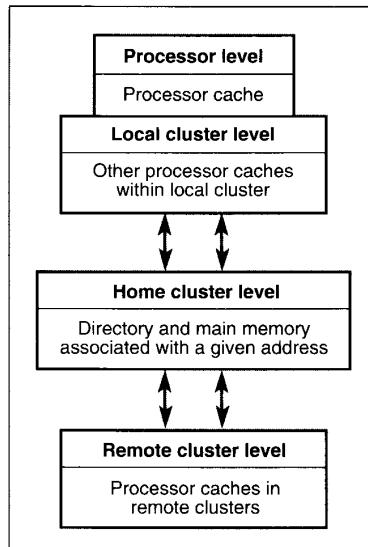


Figure 3. Memory hierarchy of Dash.

plied by that cache and no state change is required at the directory level. If the request must be sent beyond the local cluster level, it goes first to the home cluster corresponding to that address.

- *Home cluster level* — The home cluster examines the directory state of the memory location while simultaneously fetching the block from main memory. If the block is clean, the data is sent to the requester and the directory is updated to show sharing by the requester. If the location is dirty, the request is forwarded to the remote cluster indicated by the directory.

- *Remote cluster level* — The dirty cluster replies with a shared copy of the data, which is sent directly to the requester. In addition, a sharing write-back message is sent to the home level to update main memory and change the directory state to indicate that the requesting and remote cluster now have shared copies of the data. Having the dirty cluster respond directly to the requester, as opposed to routing it through the home, reduces the latency seen by the requesting processor.

Now consider the sequence of operations that occurs when a location is written:

- *Processor level* — If the location is dirty in the writing processor's cache, the write can complete immediately. Otherwise, a read-exclusive request is

issued on the local cluster's bus to obtain exclusive ownership of the line and retrieve the remaining portion of the cache line.

- *Local cluster level* — If one of the caches within the cluster already owns the cache line, then the read-exclusive request is serviced at the local level by a cache-to-cache transfer. This allows processors within a cluster to alternately modify the same memory block without any intercluster interaction. If no local cache owns the block, then a read-exclusive request is sent to the home cluster.

- *Home cluster level* — The home cluster can immediately satisfy an ownership request for a location that is in the uncached or shared state. In addition, if a block is in the shared state, then all cached copies must be invalidated. The directory indicates the clusters that have the block cached. Invalidations requests are sent to these clusters while the home concurrently sends an exclusive data reply to the requesting cluster. If the directory indicates that the block is dirty, then the read-exclusive request must be forwarded to the dirty cluster, as in the case of a read.

- *Remote cluster level* — If the directory had indicated that the memory block was shared, then the remote clusters receive an invalidation request to eliminate their shared copy. Upon receiving the invalidation, the remote clusters send an acknowledgment to the requesting cluster. If the directory had indicated a dirty state, then the dirty cluster receives a read-exclusive request. As in the case of the read, the remote cluster responds directly to the requesting cluster and sends a dirty-transfer message to the home indicating that the requesting cluster now holds the block exclusively.

When the writing cluster receives all the invalidation acknowledgments or the reply from the home or dirty cluster, it is guaranteed that all copies of the old data have been purged from the system. If the processor delays completing the write until all acknowledgments are received, then the new write value will become available to all other processors at the same time. However, invalidations involve round-trip messages to multiple clusters, resulting in potentially long delays. Higher processor utilization can be obtained by allowing the write to proceed immediately after the

ownership reply is received from the home. Unfortunately, this may lead to inconsistencies with the memory model assumed by the programmer. The next section describes how Dash relaxes the constraints on memory request ordering, while still providing a reasonable programming model to the user.

Memory consistency. The memory consistency model supported by an architecture directly affects the amount of buffering and pipelining that can take place among memory requests. In addition, it has a direct effect on the complexity of the programming model presented to the user. The goal in Dash is to provide substantial freedom in the ordering among memory requests, while still providing a reasonable programming model to the user.

At one end of the consistency spectrum is the *sequential consistency* model,³ which requires execution of the parallel program to appear as an interleaving of the execution of the parallel processes on a sequential machine. Sequential consistency can be guaranteed by requiring a processor to complete one memory request before it issues the next request.⁴ Sequential consistency, while conceptually appealing, imposes a large performance penalty on memory accesses. For many applications, such a model is too strict, and one can make do with a weaker notion of consistency.

As an example, consider the case of a processor updating a data structure within a critical section. If updating the structure requires several writes, each write in a sequentially consistent system will stall the processor until all other cached copies of that location have been invalidated. But these stalls are unnecessary as the programmer has already made sure that no other process can rely on the consistency of that data structure until the critical section is exited. If the synchronization points can be identified, then the memory need only be consistent at those points. In particular, Dash supports the use of the *release consistency* model,⁵ which only requires the operations to have completed before a critical section is released (that is, a lock is unlocked).

Such a scheme has two advantages. First, it provides the user with a reasonable programming model, since the programmer is assured that when the critical section is exited, all other processors will have a consistent view of the mod-

Release consistency provides a 10- to 40-percent increase in performance over sequential consistency.

ified data structure. Second, it permits reads to bypass writes and the invalidations of different write operations to overlap, resulting in lower latencies for accesses and higher overall performance. Detailed simulation studies for processors with blocking reads have shown that release consistency provides a 10- to 40-percent increase in performance over sequential consistency.⁵ The disadvantage of the model is that the programmer or compiler must identify all synchronization accesses.

The Dash prototype supports the release consistency model in hardware. Since we use commercial microprocessors, the processor stalls on read operations until the read data is returned from the cache or lower levels of the memory hierarchy. Write operations, however, are nonblocking. There is a write buffer between the first- and second-level caches. The write buffer queues up the write requests and issues them in order. Furthermore, the servicing of write requests is overlapped. As soon as the cache receives the ownership and data for the requested cache line, the write data is removed from the write buffer and written into the cache line. The next write request can be serviced while the invalidation acknowledgments for the previous write operations filter in. Thus, parallelism exists at two levels: the processor executes other instructions and accesses its first-level cache while write operations are pending, and invalidations of multiple write operations are overlapped.

The Dash prototype also provides fence operations that stall the processor or write-buffer until previous operations complete. These fence operations allow software to emulate more stringent consistency models.

Memory access optimizations. The use of release consistency helps hide the latency of write operations. However,

since the processor stalls on read operations, it sees the entire duration of all read accesses. For applications that exhibit poor cache behavior or extensive read/write sharing, this can lead to significant delays while the processor waits for remote cache misses to be filled. To help with these problems Dash provides a variety of prefetch and pipelining operations.

Prefetch operations. A prefetch operation is an explicit nonblocking request to fetch data before the actual memory operation is issued. Hopefully, by the time the process needs the data, its value has been brought closer to the processor, hiding the latency of the regular blocking read. In addition, nonblocking prefetch allows the pipelining of read misses when multiple cache blocks are prefetched. As a simple example of its use, a process wanting to access a row of a matrix stored in another cluster's memory can do so efficiently by first issuing prefetch reads for all cache blocks corresponding to that row.

Dash's prefetch operations are non-binding and software controlled. The processor issues explicit prefetch operations that bring a shared or exclusive copy of the memory block into the processor's cache. Not binding the value at the time of the prefetch is important in that issuing the prefetch does not affect the consistency model or force the compiler to do a conservative static dependency analysis. The coherence protocol keeps the prefetched cache line coherent. If another processor happens to write to the location before the prefetching processor accesses the data, the data will simply be invalidated. The prefetch will be rendered ineffective, but the program will execute correctly. Support for an exclusive prefetch operation aids cases where the block is first read and then updated. By first issuing the exclusive prefetch, the processor avoids first obtaining a shared copy and then having to rerequest an exclusive copy of the block. Studies have shown that, for certain applications, the addition of a small number of prefetch instructions can increase processor utilization by more than a factor of two.⁶

Update and deliver operations. In some applications, it may not be possible for the consumer process to issue a prefetch early enough to effectively hide the latency of memory. Likewise, if multiple

consumers need the same item of data, the communication traffic can be reduced if data is multicast to all the consumers simultaneously. Therefore, Dash provides operations that allow the producer to send data directly to consumers. There are two ways for the producing processor to specify the consuming processors. The *update-write* operation sends the new data directly to all processors that have cached the data, while the *deliver* operation sends the data to specified clusters.

The *update-write* primitive updates the value of all existing copies of a data word. Using this primitive, a processor does not need to first acquire an exclusive copy of the cache line, which would result in invalidating all other copies. Rather, data is directly written into the home memory and all other caches holding a copy of the line. These semantics are particularly useful for event synchronization, such as the release event for a barrier.

The *deliver* instruction explicitly specifies the destination clusters of the transfer. To use this primitive, the producer first writes into its cache using normal, invalidating write operations. The producer then issues a deliver instruction, giving the destination clusters as a bit vector. A copy of the cache line is then sent to the specified clusters, and the directory is updated to indicate that the various clusters now share the data. This operation is useful in cases when the producer makes multiple writes to a block before the consumers will want it or when the consumers are unlikely to be caching the item at the time of the write.

Support for synchronization. The access patterns to locations used for synchronization are often different from those for other shared data. For example, whenever a highly contended lock is released, waiting nodes rush to grab the lock. In the case of barriers, many processors must be synchronized and then released. Such activity often causes hot spots in the memory system. Consequently, synchronization variables often warrant special treatment. In addition to update writes, Dash provides two extensions to the coherence protocol that directly support synchronization objects. The first is queue-based locks, and the second is fetch-and-increment operations.

Most cache-coherent architectures handle locks by providing an atomic

test&set instruction and a cached test-and-test&set scheme for spin waiting. Ideally, these spin locks should meet the following criteria:

- minimum amount of traffic generated while waiting,
- low latency release of a waiting processor, and
- low latency acquisition of a free lock.

Cached test&set schemes are moderately successful in satisfying these criteria for low-contention locks, but fail for high-contention locks. For example, assume there are N processors spinning on a lock value in their caches. When the lock is released, all N cache values are invalidated, and N reads are generated to the memory system. Depending on the timing, it is possible that all N processors come back to do the test&set on the location once they realize the lock is free, resulting in further invalidations and rereads. Such a scenario produces unnecessary traffic and increases the latency in acquiring and releasing a lock.

The *queue-based locks* in Dash address this problem by using the directory to indicate which processors are spinning on the lock. When the lock is released, one of the waiting clusters is chosen at random and is granted the lock. The grant request invalidates only that cluster's caches and allows one processor within that cluster to acquire the lock with a local operation. This scheme lowers both the traffic and the latency involved in releasing a processor waiting on a lock. Informing only one cluster of the release also eliminates unnecessary traffic and latency that would be incurred if all waiting processors were allowed to contend. A time-out mechanism on the lock grant allows the grant to be sent to another cluster if the spinning process has been swapped out or migrated. The queued-on-lock-bit primitive described in Goodman et al.⁷ is similar to Dash's queue-based locks, but uses pointers in the processor caches to maintain the list of the waiting processors.

The *fetch-and-increment* and *fetch-and-decrement* primitives provide atomic increment and decrement operations on uncached memory locations. The value returned by the operations is the value before the increment or decrement. These operations have low serialization and are useful for implementing several

synchronization primitives such as barriers, distributed loops, and work queues. The serialization of these operations is small because they are done directly at the memory site. The low serialization provided by the fetch-and-increment operation is especially important when many processors want to increment a location, as happens when getting the next index in a distributed loop. The benefits of the proposed operations become apparent when contrasted with the alternative of using a normal variable protected by a lock to achieve the atomic increment and decrement. The alternative results in significantly more traffic, longer latency, and increased serialization.

The Dash implementation

A hardware prototype of the Dash architecture is currently under construction. While we have developed a detailed software simulator of the system, we feel that a hardware implementation is needed to fully understand the issues in the design of scalable cache-coherent machines, to verify the feasibility of such designs, and to provide a platform for studying real applications and software running on a large ensemble of processors.

To focus our effort on the novel aspects of the design and to speed the completion of a usable system, the base cluster hardware used in the prototype is a commercially available bus-based multiprocessor. While there are some constraints imposed by the given hardware, the prototype satisfies our primary goals of scalable memory bandwidth and high performance. The prototype includes most of Dash's architectural features since many of them can only be fully evaluated on the actual hardware. The system also includes dedicated performance monitoring logic to aid in the evaluation.

Dash prototype cluster. The prototype system uses a Silicon Graphics Power Station 4D/340 as the base cluster. The 4D/340 system consists of four Mips R3000 processors and R3010 floating-point coprocessors running at 33 megahertz. Each R3000/R3010 combination can reach execution rates up to 25 VAX MIPS and 10 Mflops. Each

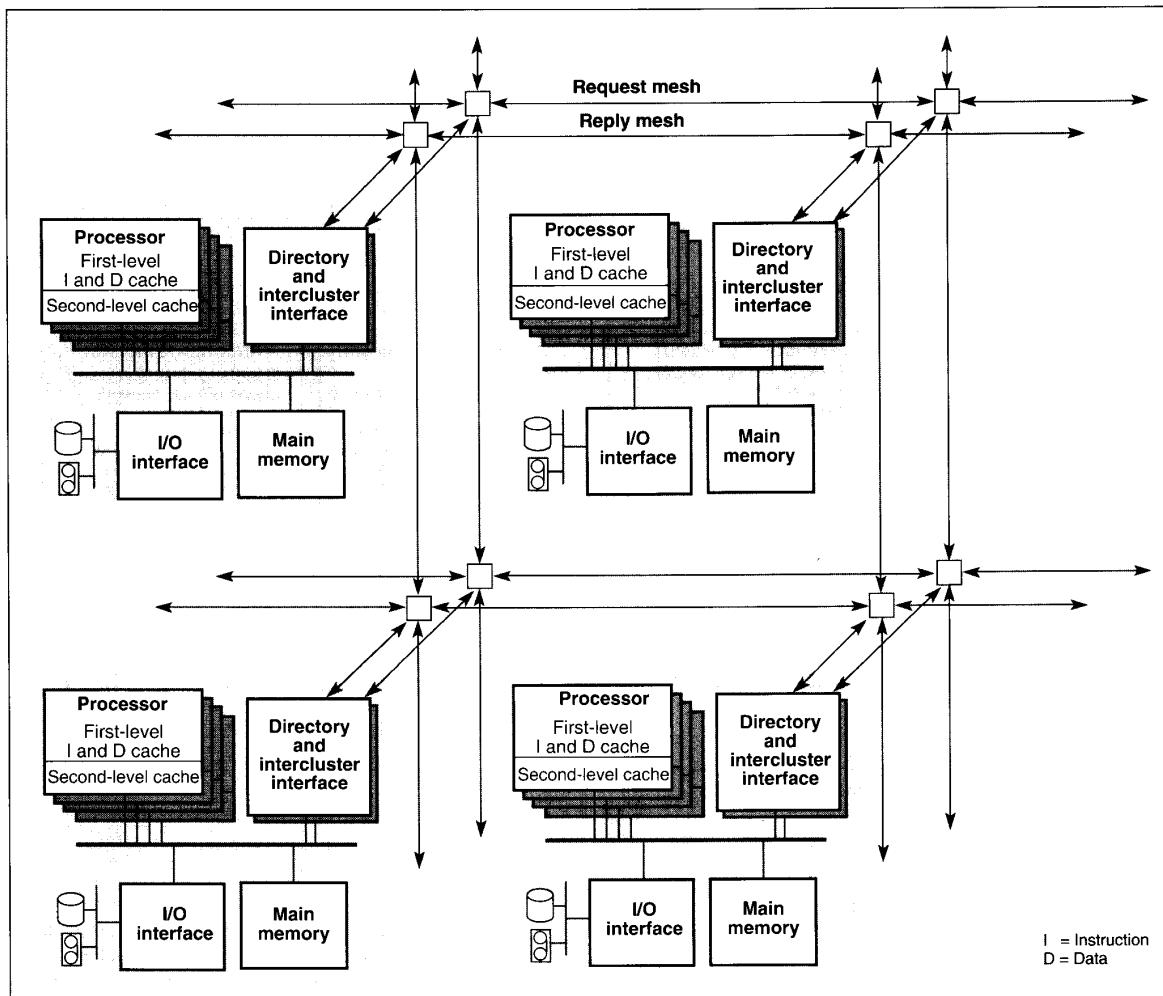


Figure 4. Block diagram of a 2×2 Dash system.

CPU contains a 64-kilobyte instruction cache and a 64-Kbyte write-through data cache. The 64-Kbyte data cache interfaces to a 256-Kbyte second-level write-back cache. The interface consists of a read buffer and a four-word-deep write buffer. Both the first- and second-level caches are direct-mapped and support 16-byte lines. The first level caches run synchronously to their associated 33-MHz processors while the second level caches run synchronous to the 16-MHz memory bus.

The second-level processor caches are responsible for bus snooping and maintaining coherence among the caches in the cluster. Coherence is maintained using an Illinois, or MESI (modified, exclusive, shared, invalid), protocol. The main advantage of using the Illinois protocol in Dash is the cache-to-cache transfers specified in it. While they do little

to reduce the latency for misses serviced by local memory, local cache-to-cache transfers can greatly reduce the penalty for remote memory misses. The set of processor caches acts as a cluster cache for remote memory. The memory bus (MPbus) of the 4D/340 is a synchronous bus and consists of separate 32-bit address and 64-bit data buses. The MPbus is pipelined and supports memory-to-cache and cache-to-cache transfers of 16 bytes every four bus clocks with a latency of six bus clocks. This results in a maximum bandwidth of 64 Mbytes per second. While the MPbus is pipelined, it is not a split-transaction bus.

To use the 4D/340 in Dash, we have had to make minor modifications to the existing system boards and design a pair of new boards to support the directory memory and intercluster interface. The main modification to the existing boards

is to add a bus retry signal that is used when a request requires service from a remote cluster. The central bus arbiter has also been modified to accept a mask from the directory. The mask holds off a processor's retry until the remote request has been serviced. This effectively creates a split-transaction bus protocol for requests requiring remote service. The new directory controller boards contain the directory memory, the intercluster coherence state machines and buffers, and a local section of the global interconnection network. The interconnection network consists of a pair of wormhole routed meshes, each with 16-bit wide channels. One mesh is dedicated to the request messages while the other handles replies. Figure 4 shows a block diagram of four clusters connected to form a 2×2 Dash system. Such a system could scale to support hundreds

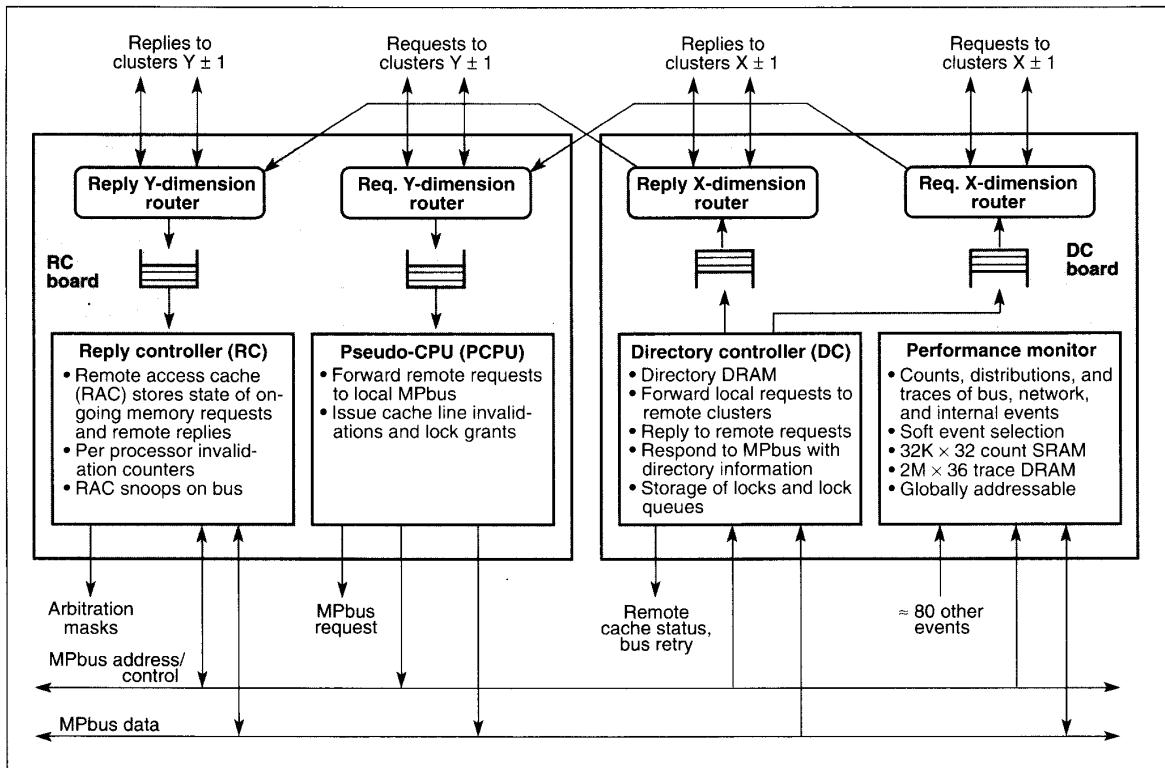


Figure 5. Block diagram of directory boards.

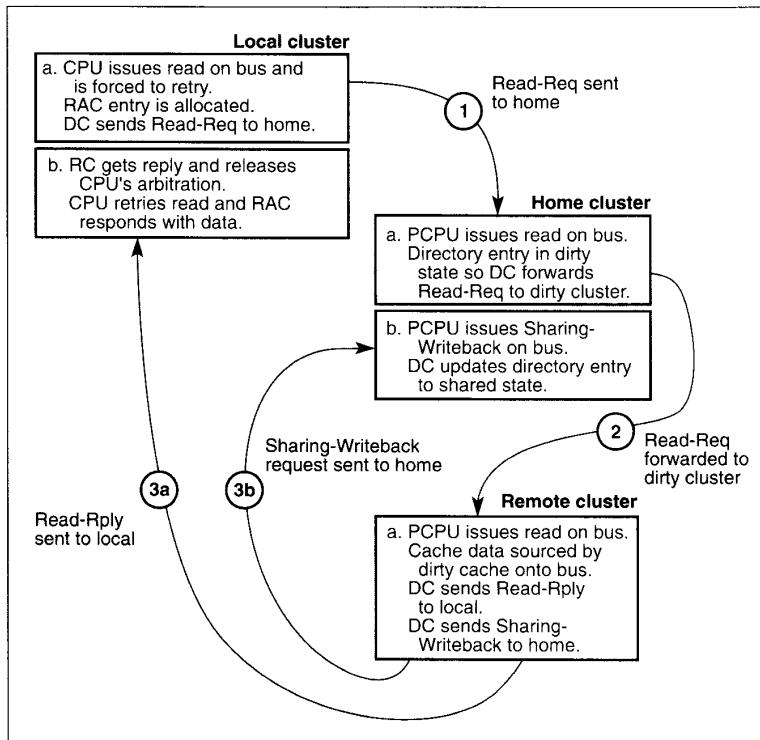


Figure 6. Flow of a read request to remote memory that is dirty in a remote cluster.

of processors, but the prototype will be limited to a maximum configuration of 16 clusters. This limit was dictated primarily by the physical memory addressability (256 Mbytes) of the 4D/340 system, but still allows for systems up to 64 processors that are capable of 1.6 GIPS and 600 scalar Mflops.

Dash directory logic. The directory logic implements the directory-based coherence protocol and connects the clusters within the system. Figure 5 shows a block diagram of the directory boards. The directory logic is split between the two logic boards along the lines of the logic used for outbound and inbound portions of intercluster transactions.

The directory controller (DC) board contains three major sections. The first is the directory controller itself, which includes the directory memory associated with the cachable main memory contained within the cluster. The DC logic initiates all outbound network requests and replies. The second section is the performance monitor, which can count and trace a variety of intra- and intercluster events. The third major section is the request and reply outbound

network logic together with the X -dimension of the network itself.

Each bus transaction accesses directory memory. The directory information is combined with the type of bus operation, the address, and the result of snooping on the caches to determine what network messages and bus controls the DC will generate. The directory memory itself is implemented as a bit vector with one bit for each of the 16 clusters. While a full-bit vector has limited scalability, it was chosen because it requires roughly the same amount of memory as a limited pointer directory given the size of the prototype, and it allows for more direct measurements of the machine's caching behavior. Each directory entry contains a single state bit that indicates whether the clusters have a shared or dirty copy of the data. The directory is implemented using dynamic RAM technology, but performs all necessary actions within a single bus transaction.

The second board is the reply controller (RC) board, which also contains three major sections. The first section is the reply controller, which tracks outstanding requests made by the local processors and receives and buffers replies from remote clusters using the remote access cache (RAC). The second section is the pseudo-CPU (PCPU), which buffers incoming requests and issues them to the cluster bus. The PCPU mimics a CPU on this bus on behalf of remote processors except that responses from the bus are sent out by the directory controller. The final section is the inbound network logic and the Y -dimension of the mesh routing networks.

The reply controller stores the state of ongoing requests in the remote access cache. The RAC's primary role is the coordination of replies to intercluster transactions. This ranges from the simple buffering of reply data between the network and bus to the accumulation of invalidation acknowledgments and the enforcement of release consistency. The RAC is organized as a 128-Kbyte direct-mapped snoopy cache with 16-byte cache lines.

One port of the RAC services the inbound reply network while the other snoops on bus transactions. The RAC is lockup-free in that it can handle several outstanding remote requests from each of the local processors. RAC entries are allocated when a local processor initiates a remote request, and they persist

until all intercluster transactions relative to that request have completed. The snoopy nature of the RAC naturally lends itself to merging requests made to the same cache block by different processors and takes advantage of the cache-to-cache transfer protocol supported between the local processors. The snoopy structure also allows the RAC to supplement the function of the processor caches. This includes support for a dirty-sharing state for a cluster (normally the Illinois protocol would force a write-back) and operations such as prefetch.

Interconnection network. As stated in the architecture section, the Dash coherence protocol does not rely on a particular interconnection network topology. However, for the architecture to be scalable, the network itself must provide scalable bandwidth. It should also provide low-latency communication. The prototype system uses a pair of *wormhole* routed meshes to implement the interconnection network. One mesh handles request messages while the other is dedicated to replies. The networks are based on variants of the mesh routing chips developed at the California Institute of Technology, where the data paths have been extended from 8 to 16 bits. Wormhole routing allows a cluster to forward a message after receiving only the first flit (flow unit) of the packet, greatly reducing the latency through each node. The average latency for each hop in the network is approximately 50 nanoseconds. The networks are asynchronous and self-timed. The bandwidth of each link is limited by the round-trip delay of the request-acknowledge signals. The prototype transfers flits at approximately 30 MHz, resulting in a total bandwidth of 120 Mbytes/second in and out of each cluster.

An important constraint on the network is that it must deliver request and reply messages without deadlocking. Most networks, including the meshes used in Dash, are guaranteed to be deadlock-free if messages are consumed at the receiving cluster. Unfortunately, the Dash prototype cannot guarantee this due, first, to the limited buffering on the directory boards and also to the fact that a cluster may need to generate an outgoing message before it can consume an incoming message. For example, to service a read request, the home

cluster must generate a reply message containing the data. Similarly, to process a request for a dirty location in a remote cluster, the home cluster needs to generate a forwarding request to that cluster. This requirement adds the potential for deadlocks that consist of a sequence of messages having circular dependencies through a node.

Dash avoids these deadlocks through three mechanisms. First, reply messages can always be consumed because they are allocated a dedicated reply buffer in the RAC. Second, the independent request and reply meshes eliminate request-reply deadlocks. Finally, a back-off mechanism breaks potential deadlocks due to request-request dependencies. If inbound requests cannot be forwarded because of blockages on the outbound request port, the requests are rejected by sending negative acknowledgment reply messages. Rejected requests are then retried by the issuing processor.

Coherence examples. The following examples illustrate how the various structures described in the previous sections interact to carry out the coherence protocol. For a more detailed discussion of the protocol, see Lenoski et al.⁸

Figure 6 shows a simple read of a memory location whose home is in a remote cluster and whose directory state is dirty in another cluster. The read request is not satisfied on the local cluster bus, so a Read-Req (message 1) is sent to the home. At this time the processor is told to retry, and its arbitration is masked. A RAC entry is allocated to track this message and assign ownership of the reply. The PCPU at the home receives the Read-Req and issues a cache read on the bus. The directory memory is accessed and indicates that the cache block is dirty in another cluster. The directory controller in the home forwards the Read-Req (message 2) to the dirty remote cluster. The PCPU in the dirty cluster issues the read on the dirty cluster's bus and the dirty processor's cache responds. The DC in the dirty cluster sends a Read-Rply (message 3a) to the local cluster and a Sharing-Writeback (message 3b) request to the home to update the directory and main memory. The RC in the local cluster receives the reply into the RAC, releases the requesting CPU for arbitration, and then sources the data onto the bus when the processor retries the read. In parallel,

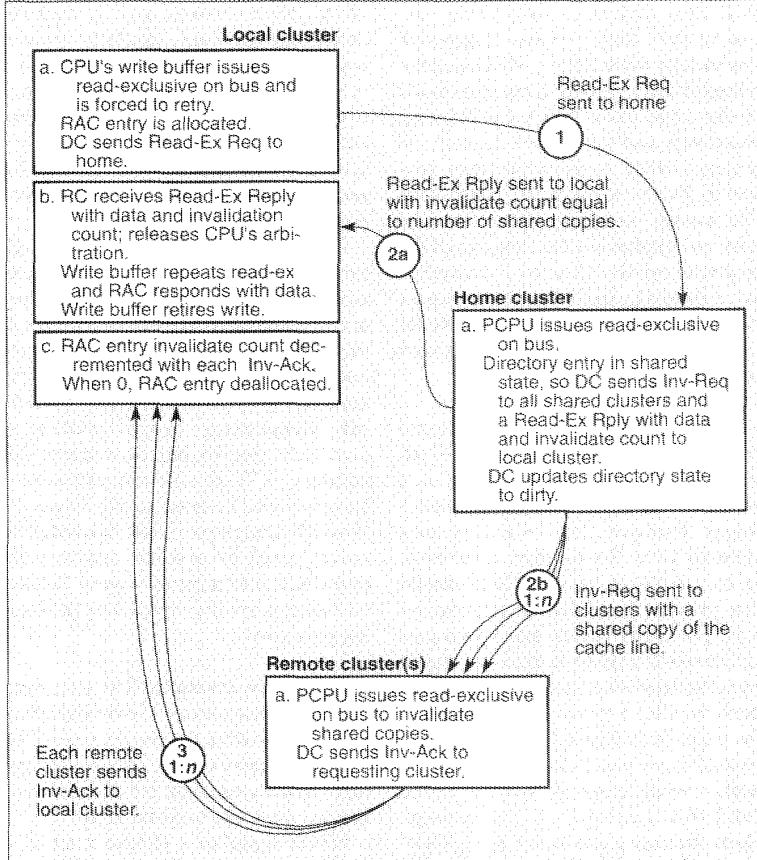


Figure 7. Flow of a read-exclusive request to remote memory that is shared in remote clusters.

the Sharing-Writeback request is received by the home PCPU, which issues it onto the bus. The sharing writeback updates the directory to a shared state indicating that the local and dirty clusters now have a read-only copy of the memory block.

Figure 7 shows the corresponding sequence for a store operation that requires remote service. The invalidation-based protocol requires the processor (actually the write buffer) to acquire exclusive ownership of the cache block before completing the store. Thus, if a store is made to a block that the processor does not have cached, or only has cached in a shared state, the processor issues a read-exclusive request on the local bus. In this case, no other cache holds the block dirty in the local cluster so a Read-Ex Req (message 1) is sent to the home cluster. As before, a RAC entry is allocated in the local cluster. At the home, the PCPU issues the read-exclusive request to the bus. The directory indicates that the line is in the shared state. This results in the DC sending a Read-Ex Rply (message 2a) to the local cluster and invalidation requests (Inv-Req, messages 2b) to the sharing clusters. The home cluster owns the block, so it can immediately update the directory to the dirty state indicating that the local cluster now holds an exclusive copy of the memory line. The

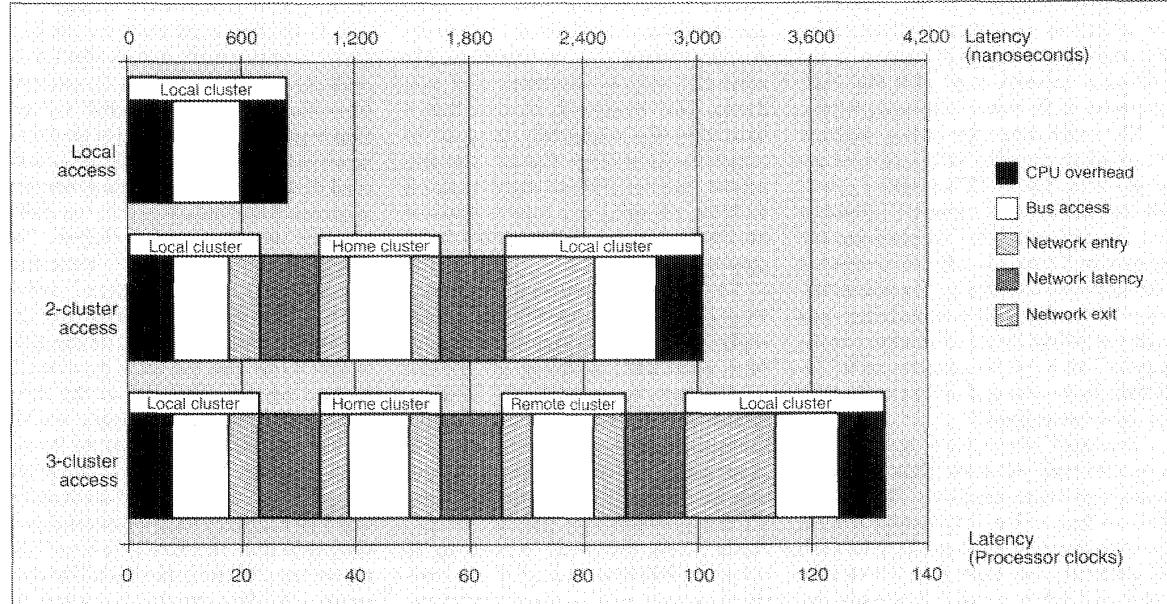


Figure 8. Latency of read requests on a 64-processor Dash prototype without contention.

Read-Ex Rply message is received in the local cluster by the RC, which can then satisfy the read-exclusive request. To assure consistency at release points, however, the RAC entry persists even after the write-buffer's request is satisfied. The RAC entry is only deallocated when it receives the number of invalidate acknowledgments (Inv-Ack, message 3) equal to an invalidation count sent in the original reply message. The RC maintains per-processor RAC allocation counters to allow the hardware to stall releasing synchronization operations until all earlier writes issued by the given processor have completed systemwide.

An important feature of the coherence protocol is its forwarding strategy. If a cluster cannot reply directly to a given request, it forwards responsibility for the request to a cluster that should be able to respond. This technique minimizes the latency for a request, as it always forwards the request to where the data is thought to be and allows a reply to be sent directly to the requesting cluster. This technique also minimizes the serialization of requests since no cluster resources are blocked while intercluster messages are being sent. Forwarding allows the directory controller to work on multiple requests concurrently (that is, makes it multithreaded) without having to retain any additional state about forwarded requests.

Software support

A comprehensive software development environment is essential to make effective use of large-scale multiprocessors. For Dash, our efforts have focused on four major areas: operating systems, compilers, programming languages, and performance debugging tools.

Dash supports a full-function Unix operating system. In contrast, many other highly parallel machines (for example, Intel iPSC2, Ncube, iWarp) support only a primitive kernel on the node processors and rely on a separate host system for program development. Dash avoids the complications and inefficiencies of a host system. Furthermore, the resident operating system can efficiently support multiprogramming and multiple users on the system. Developed in cooperation with Silicon Graphics, the Dash OS is a modified version of the

existing operating system on the 4D/340 (Irix, a variation of Unix System V.3). Since Irix was already multithreaded and worked with multiple processors, many of our changes have been made to accommodate the hierarchical nature of Dash, where processors, main memory, and I/O devices are all partitioned across the clusters. We have also adapted the Irix kernel to provide access to the special hardware features of Dash such as prefetch, update write, and queue-based locks. Currently, the modified OS is running on a four-cluster Dash system, and we are exploring several new algorithms for process scheduling and memory allocation that will exploit the Dash memory hierarchy.

At the user level, we are working on several tools to aid the development of parallel programs for Dash. At the most primitive level, a parallel macro library provides structured access to the underlying hardware and operating-system functions. This library permits the development and porting of parallel applications to Dash using standard languages and tools. We are also developing a parallelizing compiler that extracts parallelism from programs written for sequential machines and tries to improve data locality. Locality is enhanced by increasing cache utilization through *blocking* and by reducing remote accesses through *static partitioning* of computation and data. Finally, *prefetching* is used to hide latency for remote accesses that are unavoidable.

Because we are interested in using Dash for a wide variety of applications, we must also find parallelism beyond the loop level. To attack this problem we have developed a new parallel language called Jade, which allows a programmer to easily express dynamic coarse-grain parallelism. Starting with a sequential program, a programmer simply augments those sections of code to be parallelized with side-effect information. The compiler and runtime system use this information to execute the program concurrently while respecting the program's data dependence constraints. Using Jade can significantly reduce the time and effort required to develop a parallel version of a serial application. A prototype of Jade is operational, and applications developed with Jade include sparse-matrix Cholesky factorization, Locus Route (a printed-circuit-board routing algo-

rithm), and MDG (a water simulation code).

To complement our compiler and language efforts, we are developing a suite of performance monitoring and analysis tools. Our high-level tools can identify portions of code where the concurrency is smallest or where the most execution time is spent. The high-level tools also provide information about synchronization bottlenecks and load-balancing problems. Our low-level tools will couple with the built-in hardware monitors in Dash. As an example, they will be able to identify portions of code where most cache misses are occurring and will frequently provide the reasons for such misses. We expect such noninvasive monitoring and profiling tools to be invaluable in pinpointing critical regions for optimization to the programmer.

Dash performance

This section presents performance data from the Dash prototype system. First, we summarize the latency for memory accesses serviced by the three lower levels of the memory hierarchy. Second, we present speedup for three parallel applications running on a simulation of the prototype using one to 64 processors. Finally, we present the actual speedups for these applications measured on the initial 16-processor Dash system.

While caches reduce the effective access time of memory, the latency of main memory determines the sensitivity of processor utilization to cache and cluster locality and indicates the costs of interprocessor communication. Figure 8 shows the unloaded latencies for read misses that are satisfied within the local cluster, within the home cluster, and by a remote (that is, dirty) cluster. Latencies for read-exclusive requests issued by the write buffer are similar. A read miss to the local cluster takes 29 processor clocks (870 ns), while a remote miss takes roughly 3.5 times as long. The delays arise primarily from the relatively slow bus in the 3D/340 and from our implementation's conservative technology and packaging. Detailed simulation has shown that queuing delays can add 20 to 120 percent to these delays. While higher levels of integration could reduce the absolute time of the prototype latencies, we believe

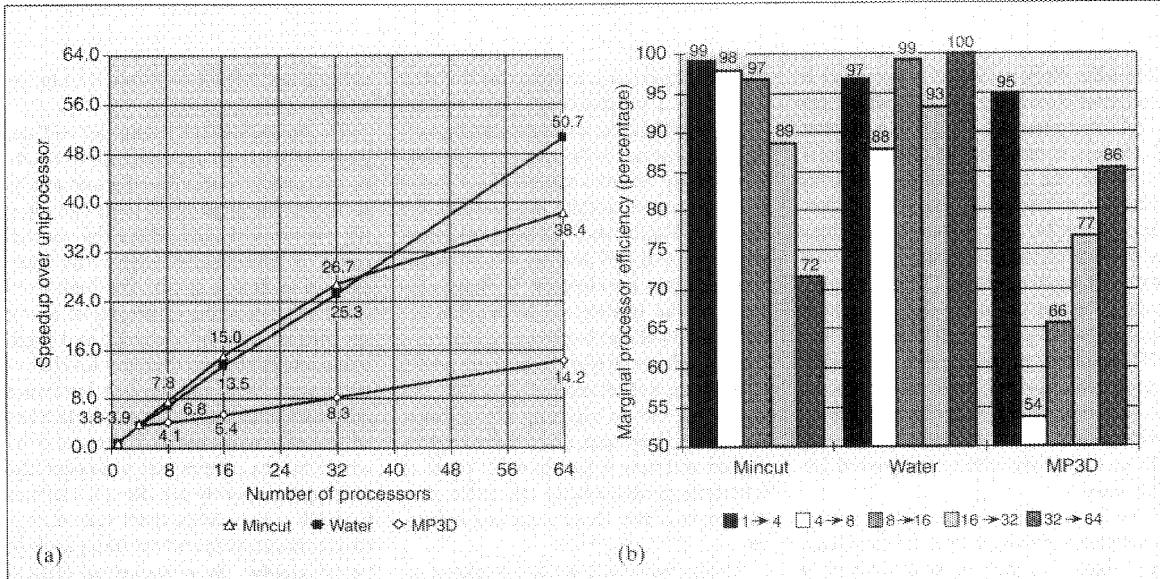


Figure 9. Speedup of three parallel applications on a simulation of the Dash prototype with one to 64 processors: (a) overall application speedup; (b) marginal efficiency of additional clusters.

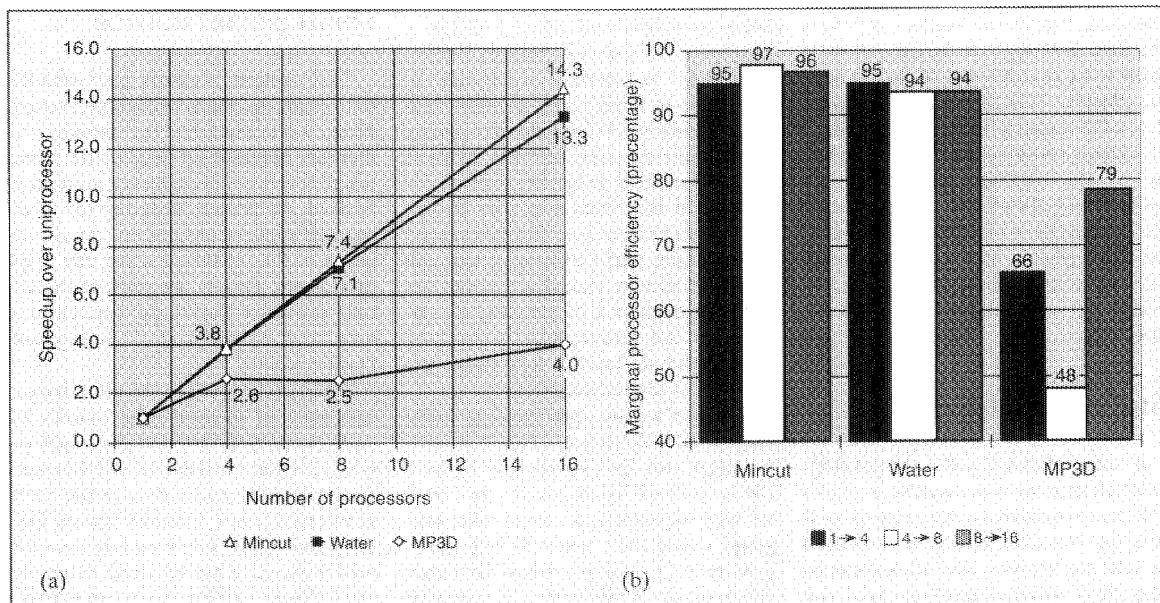


Figure 10. Speedup of three parallel applications on the actual Dash prototype hardware with one to 16 processors: (a) overall application speedup; (b) marginal efficiency of additional clusters.

the increasing clock rate of microprocessors implies that the latencies measured in processor clocks will remain similar.

Applications for large-scale multiprocessors must utilize locality to realize good cache hit rates, minimize remote accesses, and achieve high processor utilization. Figure 9 shows the speedup and processor efficiency for three appli-

cations on simulated Dash systems consisting of one to 64 processors (that is, one to 16 clusters). The line graph shows overall application speedup, while the bar chart shows the marginal efficiency of additional clusters. The marginal efficiency is defined as the average processor utilization, assuming processors were 100 percent utilized at the previous data point. The three applications

simulated are Water, Mincut, and MP3D. Water is a molecular-dynamics code that computes the energy of a system of water molecules. Mincut uses parallel simulated annealing to solve a graph-partitioning problem. MP3D models a wind tunnel in the upper atmosphere, using a discrete particle-based simulation.

The applications were simulated using a combination of the Tango multi-

processor simulator and a detailed memory simulator for the Dash prototype. Tango allows a parallel application to run on a uniprocessor and generates a parallel memory-reference stream. The detailed memory simulator is tightly coupled with Tango and provides feedback on the latency of individual memory operations.

On the Dash simulator, Water and Mincut achieve reasonable speedup through 64 processors. For Water, the reason is that the application exhibits good locality. As the number of clusters increases from two to 16, cache hit rates are relatively constant, and the percent of cache misses handled by the local cluster only decreases from 69 to 64 percent. Thus, miss penalties increase only slightly with system size and do not adversely affect processor utilizations. For Mincut, good speedup results from very good cache hit rates (98 percent for shared references). The speedup falls off for 64 processors due to lock contention in the application.

MP3D obviously does not exhibit good speedup on the Dash prototype. This particular encoding of the MP3D application requires frequent interprocessor communication, thus resulting in frequent cache misses. On average, about 4 percent of the instructions executed in MP3D generate a read miss for a shared data item. When only one cluster is being used, all these misses are serviced locally. However, when we go to two clusters, a large fraction of the cache misses are serviced remotely. This more than doubles the average miss latency, thus nullifying the potential gain from the added processors. Likewise, when four clusters are used, the full benefit is not realized because most misses are now serviced by a remote dirty cache, requiring a three-hop access.

Reasonable speedup is finally achieved when going from 16 to 32 and 64 processors (77 percent and 86 percent marginal efficiency, respectively), but overall speedup is limited to 14.2. Even on MP3D, however, caching is beneficial. A 64-processor system with the timing of Dash, but without the caching of shared data, achieves only a 4.1 speedup over the cached uniprocessor. For Water and Mincut the improvements from caching are even larger.

Figure 10 shows the speedup for the three applications on the real Dash hardware using one to 16 processors. The applications were run under an early

version of the Dash OS. The results for Water and Mincut correlate well with the simulation results, but the MP3D speedups are somewhat lower. The problem with MP3D appears to be that simulation results did not include private data references. Since MP3D puts a heavy load on the memory system, the extra load of private misses adds to the queuing delays and reduces the multi-processor speedups.

We have run several other applications on our 16-processor prototype. These include two hierarchical n -body applications (using Barnes-Hut and Greengard-Rokhlin algorithms), a radiosity application from computer graphics, a standard-cell routing application from very large scale integration computer-aided design, and several matrix-oriented applications, including one performing sparse Cholesky factorization. There is also an improved version of the MP3D application that exhibits better locality and achieves almost linear speedup on the prototype.

Over this initial set of 10 parallel applications, the harmonic mean of the speedup on 16 processors is 10.5. Furthermore, if old MP3D is left out, the harmonic mean rises to over 12.8. Overall, our experience with the 16-processor machine has been very promising and indicates that many applications should be able to achieve over 40 times speedup on the 64-processor system.

Related work

There are other proposed scalable architectures that support a single address space with coherent caches. A comprehensive comparison of these machines with Dash is not possible at this time, because of the limited experience with this class of machines and the lack of details on many of the critical machine parameters. Nevertheless, a general comparison illustrates some of the design trade-offs that are possible.

Encore GigaMax and Stanford Paradigm. The Encore GigaMax architecture⁹ and the Stanford Paradigm project¹⁰ both use a hierarchy-of-buses approach to achieve scalability. At the top level, the Encore GigaMax is composed of several clusters on a global bus. Each cluster consists of several processor modules, main memory, and a cluster cache. The cluster cache holds a copy of

all remote locations cached locally and also all local locations cached remotely. Each processing module consists of several processors with private caches and a large, shared, second-level cache. A hierarchical snoopy protocol keeps the processor and cluster caches coherent.

The Paradigm machine is similar to the GigaMax in its hierarchy of processors, caches, and buses. It is different, however, in that the physical memory is all located at the global level, and it uses a hierarchical directory-based coherence protocol. The clusters containing cached data are identified by a bit-vector directory at every level, instead of using snooping cluster caches. Paradigm also provides a lock bit per memory block that enhances performance for synchronization and explicit communication.

The hierarchical structure of these machines is appealing in that they can theoretically be extended indefinitely by increasing the depth of the hierarchy. Unfortunately, the higher levels of the tree cannot grow indefinitely in bandwidth. If a single global bus is used, it becomes a critical link. If multiple buses are used at the top, the protocols become significantly more complex. Unless an application's communication requirements match the bus hierarchy or its traffic-sharing requirements are small, the global bus will be a bottleneck. Both requirements are restrictive and limit the classes of applications that can be efficiently run on these machines.

IEEE Scalable Coherent Interface.

The IEEE P1596 Scalable Coherent Interface (SCI) is an interface standard that also strives to provide a scalable system model based on distributed directory-based cache coherence.¹¹ It differs from Dash in that it is an interface standard, not a complete system design. SCI only specifies the interfaces that each processing node should implement, leaving open the actual node design and exact interconnection network. SCI's role as an interface standard gives it somewhat different goals from those of Dash, but systems based on SCI are likely to have a system organization similar to Dash.

The major difference between SCI and Dash lies in how and where the directory information is maintained. In SCI, the directory is a distributed sharing list maintained by the processor caches

themselves. For example, if processors A, B, and C are caching some location, then the cache entries storing this location include pointers that form a doubly linked list. At main memory, only a pointer to the processor at the head of the linked list is maintained. In contrast, Dash places all the directory information with main memory.

The main advantage of the SCI scheme is that the amount of directory pointer storage grows naturally as new processing nodes are added to the system. Dash-type systems generally require more directory memory than SCI systems and must use a limited directory scheme to scale to a large configuration. On the other hand, SCI directories would typically use the same static RAM technology as the processor caches while the Dash directories are implemented in main memory DRAM technology. This difference tends to offset the potential storage efficiency gains of the SCI scheme.

The primary disadvantage of the SCI scheme is that the distribution of individual directory entries increases the latency and complexity of the memory references, since additional directory-update messages must be sent between processor caches. For example, on a write to a shared block cached by N processors (including the writing processor), the writer must perform the following actions:

- detach itself from the sharing list,
- interrogate memory to determine the head of the sharing list,
- acquire head status from the current head, and
- serially purge the other processor caches by issuing invalidation requests and receiving replies that indicate the next processor in the list.

Altogether, this amounts to $2N + 6$ messages and, more importantly, $N + 1$ serial directory lookups. In contrast, Dash can locate all sharing processors in a single directory lookup, and invalidation messages are serialized only by the network transmission rate.

The SCI working committee has proposed several extensions to the base protocol to reduce latency and support additional functions. In particular, the committee has proposed the addition of directory pointers that allow sharing lists to become sharing trees, support for request forwarding, use of a clean cached state, and support for queue-

based locks. While these extensions reduce the differences between the two protocols, they also significantly increase the complexity of SCI.

MIT Alewife. The Alewife machine¹² is similar to Dash in that it uses main memory directories and connects the processing nodes with mesh network. There are three main differences between the two machines:

- Alewife does not have a notion of clusters — each node is a single processor.
- Alewife uses software to handle directory pointer overflow.
- Alewife uses multicontext processors as its primary latency-hiding mechanism.

The size of clusters (one processor, four processors, or more) is dictated primarily by the engineering trade-offs between the overhead of hardware for each node (memory, network interface, and directory) and the bandwidth available within and between clusters. Techniques for scaling directories efficiently are a more critical issue. Whether hardware techniques, such as proposed in O'Kafka and Newton² and Gupta et al.,¹ or the software techniques of Alewife will be more effective remains an open question, though we expect the practical differences to be small. Multiple contexts constitute a mechanism that helps hide memory latency, but one that clearly requires additional application parallelism to be effective. Overall, while we believe that support for multiple contexts is useful and can complement other techniques, we do not feel that its role will be larger than other latency-hiding mechanisms such as release consistency and nonbinding prefetch.¹³

We have described the design and implementation decisions for Dash, a multiprocessor that combines the programmability of single-address-space machines with the scalability of message-passing machines. The key means to this scalability are a directory-based cache-coherence protocol, distributed memories and directories, and a scalable interconnection network. The design focuses on reducing memory latency to keep processor performance high, though it also provides latency-hiding techniques such as prefetch and release consistency to mitigate the effects of unavoidable system delays.

At the time of this writing, the 2×2 Dash prototype is stable. It is accessible on the Internet and used daily for research into parallel applications, tools, operating systems, and directory-based architectures. As indicated in the performance section, results from this initial configuration are very promising. Work on extending the 2×2 cluster system to the larger 4×4 (64-processor) system is ongoing. All major hardware components are on hand and being debugged. By the time this article is in print, we expect to have an initial version of the Unix kernel and parallel applications running on the larger machine. ■

Acknowledgments

This research was supported by DARPA contracts N00014-87-K-0828 and N00039-91-C-0138. In addition, Daniel Lenoski is supported by Tandem Computers, James Laudon and Wolf-Dietrich Weber are supported by IBM, and Kourosh Gharachorloo is supported by Texas Instruments. Anoop Gupta is partly supported by a National Science Foundation Presidential Young Investigator Award.

We also thank Silicon Graphics for their technical and logistical support and Valid Logic Systems for their grant of computer-aided engineering tools.

References

1. A. Gupta, W.-D. Weber, and T. Mowry, "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," *Proc. 1990 Int'l Conf. Parallel Processing*, IEEE Computer Society Press, Los Alamitos, Calif., Order No. 2101, pp. 312-321.
2. B.W. O'Kafka and A.R. Newton, "An Empirical Evaluation of Two Memory-Efficient Directory Methods," *Proc. 17th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 2047, 1990, pp. 138-147.
3. L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Trans. Computers*, Sept. 1979, Vol. C-28, No. 9, pp. 241-248.
4. C. Scheurich and M. Dubois, "Dependency and Hazard Resolution in Multiprocessors," *Proc. 14th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 776, 1987, pp. 234-243.

5. K. Gharachorloo, A. Gupta, and J. Hennessy, "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM, New York, 1991, pp. 245-257.
6. T. Mowry and A. Gupta, "Tolerating Latency Through Software in Shared-Memory Multiprocessors," *J. Parallel and Distributed Computing*, Vol. 12, No. 6, June 1991, pp. 87-106.
7. J.R. Goodman, M.K. Vernon, and P.J. Woest, "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors," *Proc. Third Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, IEEE CS Press, Los Alamitos, Calif., Order No. 1936, 1989, pp. 64-73.
8. D. Lenoski et al., "The Directory-Based Cache Coherence Protocol for the Dash Multiprocessor," *Proc. 17th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 2047, 1990, pp. 148-159.
9. A.W. Wilson, Jr., "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors," *Proc. 14th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 776, 1987, pp. 244-252.
10. D.R. Cheriton, H.A. Goosen, and P.D. Boyle, "Paradigm: A Highly Scalable Shared-Memory Multicomputer Architecture," *Computer*, Vol. 24, No. 2, Feb. 1991, pp. 33-46.
11. D.V. James et al., "Distributed-Directory Scheme: Scalable Coherent Interface," *Computer*, Vol. 23, No. 6, June 1990, pp. 74-77.
12. A. Agarwal et al., "Limitless Directories: A Scalable Cache Coherence Scheme," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM, New York, 1991, pp. 224-234.
13. A. Gupta et al., "Comparative Evaluation of Latency Reducing and Tolerating Techniques," *Proc. 18th Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 2146, 1991, pp. 254-263.



Daniel Lenoski is a research scientist in the Processor and Memory Group of Tandem Computers. He recently completed his PhD in electrical engineering in the Computer Systems Laboratory at Stanford University. His research ef-

orts concentrated on the design and implementation of Dash and other issues related to scalable multiprocessors. His prior work includes the architecture definition of Tandem's CLX 600, 700, and 800 series processors.

Lenoski received a BSEE from the California Institute of Technology in 1983 and an MSEE from Stanford in 1985.



James Laudon is a PhD candidate in the Department of Electrical Engineering at Stanford University. His research interests include multiprocessor architectures and algorithms.

Laudon received a BS in electrical engineering from the University of Wisconsin-Madison in 1987 and an MS in electrical engineering from Stanford University in 1988. He is a member of the IEEE Computer Society and ACM.



Kourosh Gharachorloo is a PhD candidate in the Computer Systems Laboratory at Stanford University. His research interests focus on techniques to reduce and tolerate memory latency in large-scale shared-memory multiprocessors.

Gharachorloo received the BS and BA degrees in electrical engineering and economics, respectively, in 1985 and the MS degree in electrical engineering in 1986, all from Stanford University.



Wolf-Dietrich Weber is a PhD candidate in the Computer Systems Laboratory at Stanford University. His research interests focus on directory-based cache coherence for scalable shared-memory multiprocessors.

Weber received the BA and BE degrees from Dartmouth College in 1986. He received an MS degree in electrical engineering from Stanford University in 1987.



Anoop Gupta is an assistant professor of computer science at Stanford University. His primary interests are in the design of hardware and software for large-scale multiprocessors.

Prior to joining Stanford, Gupta was on the

research faculty of Carnegie Mellon University, where he received his PhD in 1986. Gupta was the recipient of a DEC faculty development award from 1987-1989, and he received the NSF Presidential Young Investigator Award in 1990.



John Hennessy is a professor of electrical engineering and computer science at Stanford University. His research interests are in exploiting parallelism at all levels to build higher performance computer systems.

Hennessy is the recipient of a 1984 Presidential Young Investigator Award. In 1987, he was named the Willard and Inez K. Bell Professor of Electrical Engineering and Computer Science. In 1991, he was elected an IEEE fellow. During a leave from Stanford in 1984-85, he cofounded MIPS Computer Systems where he continues to participate in industrializing the RISC concept as chief scientist.



Mark Horowitz is an associate professor of electrical engineering at Stanford University. His research interests include high-speed digital integrated circuit designs, CAD tools for IC design, and processor architecture. He is a recipient of a 1985 Presidential Young Investigator Award.

During a leave from Stanford in 1989-90, he cofounded Rambus, a company that is working on improving memory bandwidth to and from DRAMs.

Horowitz received the BS and SM degrees in electrical engineering and computer science from the Massachusetts Institute of Technology and his PhD in electrical engineering from Stanford University.



Monica S. Lam has been an assistant professor in the Computer Science Department at Stanford University since 1988. Her current research project is to develop a compiler system that optimizes data locality and exploits parallelism at task, loop, and instruction granularities. She was one of the chief architects and compiler designers for the CMU Warp machine and the CMU-Intel's iWarp.

Lam received her BS from University of British Columbia in 1980 and a PhD in computer science from Carnegie Mellon University in 1987.

Readers may contact Daniel Lenoski at Tandem Computers, 19333 Valco Parkway, MS 3-03, Cupertino, CA 95014; e-mail lenoski_dan@tandem.com. Anoop Gupta can be reached at Stanford Univ., CIS-212, Stanford, CA 94305; e-mail ag@pepper.stanford.edu.

Exploring the Architecture of a Stream Register-Based Snoop Filter

Matthias Blumrich, Valentina Salapura, and Alan Gara

IBM Thomas J. Watson Research Center
Yorktown Heights, NY, USA

Abstract. Multi-core processors have become mainstream; they provide parallelism with relatively low complexity. As true on-chip symmetric multiprocessors evolve, coherence traffic between cores is becoming problematic, both in terms of performance and power. The negative effects of coherence (snoop) traffic can be significantly mitigated through the use of snoop filtering. The idea is to shield each cache with a device that can eliminate snoop requests for addresses that are known not to be in the cache. This improves performance significantly for caches that cannot perform normal load and snoop lookups simultaneously. In addition, the reduction of snoop lookups yields power savings. This paper describes Stream Register snoop filtering, which captures the spatial locality of multiple memory reference streams in a small number of registers. We propose a snoop filter that combines Stream Registers with "snoop caching", a mechanism that captures the temporal locality of frequently-accessed addresses. Simulations of SPLASH-2 benchmarks on a 4-core multiprocessor illustrate tradeoffs and strengths of these two techniques. We show that their combination is most effective, eliminating 94% - 99% of all snoop requests using only a small number of stream registers and snoop cache lines.

1 Introduction

As single-core performance becomes increasingly hard to improve, and marginal costs are growing, both in terms of complexity and power/performance inefficiency [1], the use of multi-core solutions to improve throughput in multi-threaded workloads has become increasingly attractive [2].

Unlike designs which target single-thread solutions with degrading power/performance efficiency, suitably scalable parallel workloads show little or no degradation in efficiency while delivering significant increases in performance through the use of multithreaded workloads. Using parallelism at the processor level also aligns with the limits of future technologies. Although performance growth has been driven by technologically-enabled increases in processor operating frequency for the past 20 years, it is increasingly hard to obtain with new technologies. One of the main reasons is the impact of wire delays as feature sizes are shrunk [3], requiring increasingly more sophisticated microarchitectures.

While faster transistors and wires are increasingly hard to obtain, the application of Dennard's CMOS scaling theory [4] is continuing to deliver improvements in density. Thus, multi-core solutions are based on a commercially viable exploitation of modern CMOS fabrication processes.

Several multi-core solutions have been introduced over the past few years, such as the IBM POWER4 and POWER5 servers, the IBM Blue Gene/P system [5], the Intel Quad Core processors [6], and the Cell Broadband Architecture [7]. Indeed, multi-core is now a well-established trend. A major challenge in the implementation of chip multiprocessors is providing a suitable memory subsystem and on-chip interconnect that combines low average access latency with high bandwidth.

As the number of processors per chip rises, the coherence traffic per processor consequently increases. One solution to reducing the cost of coherence is to manage it in software; a solution adopted by both the Blue Gene/L [8] and Cell system architectures. In Blue Gene/L, software managed coherence is achieved by using one of two software abstraction models: virtual node mode, wherein each processor is a separate node in the Blue Gene/L-optimized MPI implementation, or coprocessor mode, where one processor is a dedicated computational node and a second processor provides I/O and system management functions. In the Cell Broadband Architecture, coherent DMA and the SPU-local store provide the necessary memory abstractions for building high-performance systems. Although software-managed coherence offers an attractive solution to achieving low-complexity memory architectures, it requires advanced compilation technologies and careful application tuning. While this is acceptable for high-end application-specific systems, providing low-complexity, coherent memory is an attractive solution for a wider range of systems.

To reduce the complexity of implementing coherence in chip multiprocessors, two component costs must be addressed:

- The bottleneck of a bus-based snoop implementation, which must be arbitrated between a high number of nodes.
- The cost of providing snoop ports to each processor's cache, or the cost of maintaining a central directory.

In this paper we have investigated the use of coherence request filtering (or snoop filtering) in a point-to-point coherence network to address these costs. The basic idea is to provide a mechanism which will significantly reduce the interference of coherence requests with processor operations without incurring costs of chip area, memory latency, or complexity inherent in existing hardware coherency support.

The contributions of this work include (1) a novel, highly efficient, point-to-point snoop filter architecture filtering in excess of 98% of all snoop requests, (2) significant reduction of area over a solution that duplicates cache directories to provide separate snoop directories, (3) an architecture to exploit the cache replacement policy to periodically re-train the snoop filters, increasing filtering effectiveness from an average of 30% to an average of 90% for the workloads studied, and (4) evaluation of the proposed architecture, including variations of several key parameters.

This paper is organized as follows. We begin with related work in Section 2. Then Section 3 describes the snoop filter architecture, and Section 4 gives a detailed description of stream registers. The simulation environment and methodology are presented in Section 5, followed by experimental results and analysis in Section 6. Finally, Section 7 concludes.

2 Related Work

In prior art, JETTY [9] is a snoop filter that combines two complementary filtering methods. The JETTY paper defines a characterization of filters as “include” or “exclude”. An include filter tracks what *is* contained in a cache (or caches) while an exclude filter tracks what *is not*. The exclude filter consists of a cache of recently invalidated lines. A snoop that hits in the exclude filter is guaranteed not to be in the cache, so it can be filtered. The JETTY include filter captures a superset of a cache’s contents. A snoop that hits in the include filter may be in the cache and should not be filtered. The include filter is like a simple bloom filter with direct-map hash functions applied to sub-fields of the address.

The JETTY paper makes an argument for snoop filtering as a means for power savings. However, our work was primarily motivated by the need to filter useless snoops that reduce performance. We also considered chip area and power consumption as significant constraints, causing us to look beyond the simple and accurate method of duplicating the cache tags as a filter. Because our simulation methodology and system organization differ considerably from the JETTY study, it is difficult to compare performance results. However, we simulated many of the same applications using the same problem sizes.

Several coherent network switches contain snoop filters that block unnecessary coherence requests from ever leaving a node. One such example is the Scalability Port Switch of the Intel E8870 chipset [10]. In this case, the snoop filter tracks the state of all cache lines within a 4-processor node for a system with up to four such nodes. Kant [11] modeled a similar system architecture with such a snoop filter. This architecture is also described in the Azusa system [12], which is based on Intel Itanium processors and may use an Intel chipset.

In [13], a HyperTransport network switch for use with AMD Opteron processors is described. The snoop filtering technique is basically the same as that of the E8870, including the fact that 4-processor nodes are supported.

A similar but more tightly-coupled architecture is evaluated in [14], where a single memory controller switch connects multiple multi-processor nodes and contains a snoop filter. The filter prevents unnecessary snoop requests between the nodes, and several variants are studied.

Snoop filters in tightly-coupled multiprocessors, such as chip multiprocessors (CMPs), can be located at each processor in order to squash unnecessary snoops without changing the overall coherence scheme. Ekman et. al. [15] describe a CMP architecture with Page Sharing Tables, which are exclude filters at the granularity of memory pages rather than cache lines. This architecture is a bit more involved in that the Page Sharing Tables coordinate to track page sharing rather than just presence.

The idea of preventing remote snoop requests from being broadcast can also be applied at the chip level as described in [16]. In this work, snoop filters keep track of memory regions, which can be quite large, and block remote snoops for memory that is known not to be shared.

3 Snoop Filter Architecture

In symmetric multiprocessor (SMP) architectures, coherency snoop requests represent a significant fraction of all cache accesses, but only a small fraction of snoop requests are actually found in any of the remote caches [9,17]. This is particularly true of supercomputing applications where data partitioning and data blocking have been performed to increase locality of reference and optimize overall compute performance. As a result, embedded cores with single-ported caches suffer significant performance loss due to unnecessary snooping because their caches are unavailable during snooping.

While data reference locality may make a coherence implementation seem unnecessary, there is a fine line between being able to prove that there is no data sharing at all, and the statistical observation that almost all data references are local. The former is a program correctness statement, the latter is a performance statement. Thus, providing an efficient snoop filtering implementation that can filter the vast majority of non-shared data traffic without burdening the cache bandwidth of every processor in the system provides a significant performance improvement over traditional cache-coherent systems. Snoop filtering and cache coherence also offer advantages over programs operating on fully disjoint data sets without hardware coherence by simplifying application porting and tuning – it no longer becomes necessary to eliminate all remote references under all circumstances, but most remote references for most situations, leading to increased programmer productivity by letting programmers focus on the common case.

This motivated us to introduce a simple hardware device that filters out incoming snoop requests, reducing the number of actual snoop requests presented to the cache, thus increasing performance and reducing power consumption. A snoop filter is associated with each of the four processors and is located outside the L1 cache. To make a snoop filter a viable implementation choice, it has to meet several requirements:

- Functional correctness – the filter cannot filter requests for data which are locally cached.
- Effectiveness – the filter should filter out a large fraction of received snoop requests.
- Design efficiency – the filter should be small and power-efficient.

In theory, a perfect snoop filter can be created by duplicating the cache tag directory and using it to determine exactly which snoops should be forwarded to the cache. However, this approach generally does not meet the design efficiency requirement because of practical limitations. In particular, the cache tag store is a highly optimized and integrated component that cannot easily be extracted, and a custom-designed equivalent with the same performance requires a large investment of time and expertise to complete. Furthermore, this paper will show that a highly-effective filter can be implemented in a fraction of the area needed for duplicate cache tags.

Ideally, a snoop filter will operate at the (typically lower) memory nest frequency to reduce power dissipation and design complexity. Lower frequency and reduced latch count reduces the number of state transitions and load on clock nets, which are the major contributors to power dissipation. Operating the snoop filter at a lower frequency simplifies the design, as transactions have to be less heavily pipelined, eliminating a variety of bypass conditions which have to be validated and tested. It also simplifies timing closure.

In some sense, a snoop filter trades off power consumed by cache lookups with power consumed by the filter. However, there is an additional overall energy reduction because the processor performance benefit resulting from reduced snoop interference causes applications to complete sooner. Therefore, energy, measured as power consumed in time, is reduced.

In this work, we only consider a write-through L1 cache, where data integrity between the processors is maintained with a cache coherence protocol based on snoop invalidates. Thus, every time a processor issues a store, snoop invalidate messages are generated at all other processors, and no other coherence messages are required. This approach could be extended to write-back coherence protocols, where both read and write snoops generally require responses from remote caches. At each remote cache, the snoop filter would quickly determine whether the snooped address was not present, and if so, immediately send the response without the need to snoop the cache.

3.1 Point-to-Point Snoop Filter Interconnection

Previous work on snoop filters has only considered bus based systems. In such systems, all cache controllers snoop a shared bus to determine whether they have a copy of every requested data block. A simple and common-place coherence protocol that utilizes snooping is “write-invalidate”.

In a write-invalidate protocol, each write causes all copies of the written line to be invalidated in all other caches. If two or more processors attempt to write the same data simultaneously, only one of them wins the race, causing the other processors’ copies to be invalidated. The use of the shared bus enforces write serialization.

For every write bus transaction, all cache controllers have to check their cache address tags (a.k.a. snoop) independently to see if they are caching the written line. With the increasing number of processors on a bus, snooping activity increases as well. Unnecessary coherency requests degrade performance of the system, especially impacting the supercomputing applications where only a small fraction of snoop requests are actually found in any of the remote caches.

In [9], several proposals for reducing snoop requests using snoop filters are described. While reducing the number of snoop requests presented to a cache up to about 70%, the performance of the systems are still limited because of the interconnect architecture and lack of support for multi-porting. The architecture described is based on a shared system bus, which establishes a common event ordering across the system. While such global time ordering is desirable to simplify the filter architecture, it limits the possible system configurations to those with a single, shared bus. Alas, such systems are known to be limited in scalability due to contention for the single global resource. In addition, global buses tend to be slow, due to the high load of multiple components attached to them, and inefficient to place in CMPs.

Thus, in a highly-optimized, high-bandwidth system, it is desirable to provide alternate interconnect architectures, such as star or point-to-point. These are advantageous, as they only have a single sender and transmitter, thereby reducing the load, allowing the use of high speed protocols, and simplifying floor planning in CMPs. Using point-to-point interconnects also allows several transmissions to be in-progress simultaneously, thereby increasing the data transfer parallelism and overall data throughput.

Another limitation of the bus-based architecture is the inability to perform snoop filtering on several requests simultaneously, because simultaneous snoop requests from several processors have to be serialized by the bus. Allowing the processing of several snoop requests concurrently provides a significant increase in the number of requests that can be handled at any one time, and thus increases overall system performance. However, this increased data throughput means that snoop filters for such interconnects must be designed to accommodate multiple requests simultaneously.

In this paper, we opt for a system incorporating snoop filters to increase overall performance and power efficiency without limiting the system design options to a common bus. We have designed a snoop filter architecture supporting systems using point-to-point connections that allows each processor's snoop filter to filter requests from multiple memory writers concurrently. Our high-performance snoop filter is implemented in a pipelined fashion to enable high system clock speeds. Figure 1 illustrates our approach.

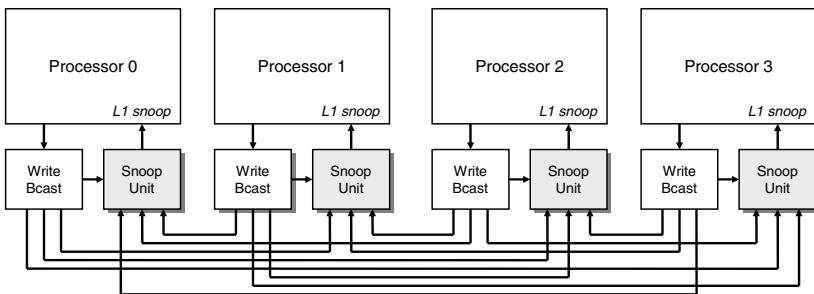


Fig. 1. Chip multiprocessor using snoop filters and a point-to-point interconnection architecture. All writes are broadcast by each processor to all remote caches for invalidation.

To take advantage of the point-to-point architecture and allow for concurrent filtering of multiple snoop requests, we implement a separate snoop filter block, or “port filter”, for each interconnect port. Thus, coherency requests of all ports are processed concurrently, and a small fraction of all requests are forwarded to the processor. For example, each snoop filter in Figure 1 would have three separate port filters, each of which handles requests from one remote processor.

As the number of processors in a CMP scales up, the interconnect naturally transitions from point-to-point to a network-on-chip (NoC). At those scales, coherence protocols are likely to be based on directories, where filtering effectively takes place at the source. Therefore, our system architecture is most obviously applicable to CMPs of modest scale [18].

It should be noted that the snoop filters are basically transparent with respect to the behavior of the point-to-point interconnect. This interconnect poses particular design challenges, such as ordering between snoop invalidates, that exist regardless of the presence of snoop filters. In this study, we assumed a consistency protocol that is not sensitive to the order of invalidations coming from different processors. When necessary, a global synchronization would be used to enforce completion of all snoops.

3.2 Snoop Filter Variants

Early on, we decided to include multiple filter units which implement various filtering algorithms in each port filter block. The motivation for this decision was to capture various characteristics of the memory references because some filtering units best capture time locality of memory references, whereas others capture reference streams. We will show in this paper that the combination of filtering algorithms achieves the highest snoop filtering rate, reducing the number of snoop requests up to 99%.

We have explored a number of snoop filter variants, but have selected the combination of a stream register based filter and a snoop cache. The snoop cache is essentially a Vector-Exclusive-JETTY [9]. It filters snoops using an algorithm which is based on the temporal locality property of snoop requests, meaning that if a single snoop request for a particular location was made, it is probable that another request to the same location will be made soon. This filter unit records a subset of memory blocks that are *not* cached.

The stream registers use an orthogonal filtering technique, exploiting the regularity of memory accesses. They record a superset of blocks that *are* cached. Results of both filter units are considered in a combined filtering decision. If either one of the filtering units decides that a snoop request should be discarded, then it is discarded.

The snoop cache filter unit keeps a record of memory references which are guaranteed not to be in the cache. These blocks have been snooped recently (thus invalidated in the cache) and are still not cached (i.e. they were not loaded into the cache since invalidation). The snoop cache filter unit contains a small array of address tags. An entry is created for each snoop request. A subsequent request for the same block will hit in the snoop cache, and be filtered. If the block is loaded in the processor cache, the corresponding entry is removed from the snoop cache, and any new coherency request to the same block will miss in the snoop cache and be forwarded to the processor cache. There is one dedicated snoop cache filter unit for each remote memory writer (processor, DMA, etc.) to allow for concurrent filtering of multiple coherency requests, thus increasing system performance.

A single snoop cache contains M snoop cache lines, each consisting of an address tag field, and a valid line vector. The address tag field is typically not the same as the address tag of the L1 data cache, but is reduced by the number of bits used for encoding a valid line vector. The valid line vector is a bit-vector that records the presence of individual, consecutive lines within an aligned block. Thus, N least significant bits from the address are decoded to a valid line vector with 2^N bits, where each bit of the vector effectively utilizes the remainder of the stored address, significantly increasing the snoop cache capacity efficiently. In the extreme case when N is zero, the whole entry in the snoop cache represents only one L1 data cache line, and the valid line vector has only one bit, corresponding to a “valid” bit.

4 Stream Registers

The stream register filter unit was introduced in [19]. Unlike the snoop cache that keeps track of what *is not* in the cache, the stream register filter keeps track of what *is* in the cache (i.e. it is an include filter). More precisely, the stream registers keep track of at

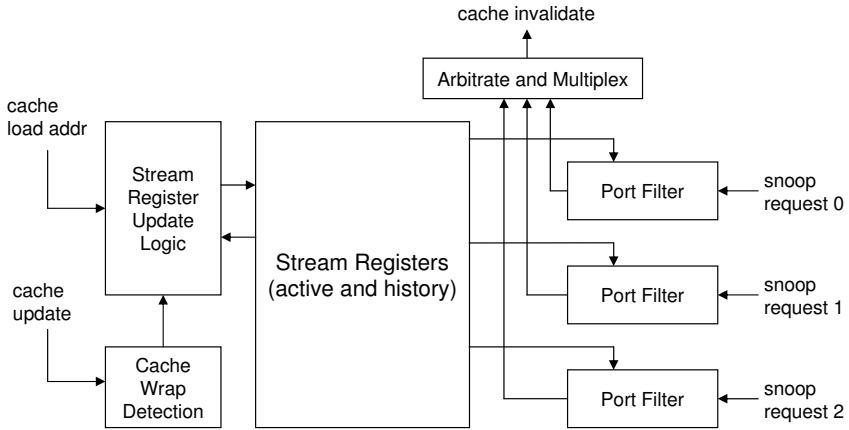


Fig. 2. Architecture of a snoop filter using stream registers. There are three port filters in order to handle snoop requests from all remote processors simultaneously.

least the lines that are in the cache, but may assume that some lines are cached which are not actually there. However, forwarding some unnecessary snoop requests to the cache does not affect correctness. The stream registers capture address streams, so they are advantageous for applications where too many spatially-distributed references overflow the snoop cache filter units.

The stream register filter unit consists of two sets of stream registers and masks (active and history sets), port filter logic, cache wrap detection logic, and stream register update logic, as illustrated in Figure 2. The stream registers and masks keep track of data which were recently loaded into the cache of the processor. One active stream register is updated every time the cache loads a new line, which is presented to the Stream Register Update Logic with appropriate control signals. A particular register is chosen for update based upon the current stream register state and the address of the new line being loaded into the cache.

Every remote snoop (snoop requests 0-2) is checked against the stream registers to see if it might be in the cache or not. This check can be performed in parallel because stream register lookups never change the state of the registers. Therefore, our architecture includes a port filter for each remote processor (or other snoop source such as DMA). Figure 2 shows three Port Filters. Snoop requests coming from one of the remote processors are checked in one of the Port Filters. Each arriving snoop requests address is compared with the state of the stream registers to determine if the snoop request could possibly be in the cache. In parallel, it is checked against a snoop cache, one of which exists in each Port Filter. If either the stream register or snoop cache lookup determine that the address is *not* in the L1 cache, then it is filtered out. Otherwise the request is forwarded to the Arbitration and Multiplexing interface and on to the cache. The Arbitrate and Multiplex logic shares the snoop interface of the cache between the Port Filters and queues unfiltered snoop requests, allowing for the maximum snoop request rate.

A stream register actually consists of a pair of registers (the base and the mask) and a valid bit. The base register keeps track of address bits that are common to all of the cache lines represented by the stream register, while the corresponding mask register keeps track of which bits these are. For example, if considering an address space of 2^{32} bytes with a cache line size of 32 bytes, a cache line load address is 27 bits in length, and the base and mask registers of the stream registers are also 27 bits in length.

Initially, the valid bit is set to zero, indicating that the stream register is not in use, and the contents of the base and mask registers are irrelevant. When the first cache line load address is added to this stream register, the valid bit is set to one, the base register is set to the line address, and all the bits of the mask register are set to one, indicating that all of the bits in the base register are significant. That is, an address that matches the address stored in the base register exactly is considered to be in the cache, while an address differing in any bit or bits is not.

At some point, another cache line load address will be added to this stream register. The new address is compared to the base register AND-ed with the mask register to determine which significant bits are different, and the mask register is then updated so that the differing bit positions become zeros in the mask. These zeros indicate that the corresponding bits of the base register are “don’t-care”, or can be assumed to take any value (zero or one). Therefore, these bits are no longer significant during comparisons to the stream register.

As an example, suppose the first two cache line load addresses are 0x1708fb1 and 0x1708fb2 (hexadecimal values). Then the contents of the stream register after these loads is:

Step 0: Base = 0x1708fb1, Mask = 0xffffffff

Step 1: Base = 0x1708fb2, Mask = 0xffffffffc

As the second address and the base register differed in the two least significant bits, those bits are cleared in the mask register. At this point, the stream register indicates that the addresses 0x1708fb0, 0x1708fb1, 0x1708fb2, and 0x1708fb3 can all be in the cache because it can no longer distinguish the two least significant bits.

Every cache line load address is added to exactly one of the multiple stream registers. Therefore, the collection of stream registers represents the complete cache state. The decision of which active register to update is made by the register update logic. In order to capture streams, we want addresses separated by the same stride to be added to the same stream register. We say that such addresses have an affinity for one another. We considered two policies for determining affinity and selecting which stream register to update:

- choose the stream register with minimal Hamming distance from the line load address (i.e. the stream register which will result in the minimum number of mask register bits changing to zero).
- choose the stream register where the most upper bits of the base register match those of the line load address.

Either mechanism guarantees that all addresses presented to the stream registers will be included within them.

Another issue is when to choose a new register instead of one that already contains a stream. We do this by assigning an “empty affinity” to unused registers and then including them in the update selection process. So when using the Hamming distance policy, for example, an empty register is chosen if the empty affinity is less than the affinity calculated for all used registers.

Over time, as cache line load addresses are added to the stream registers, they become less and less accurate in terms of their knowledge of what is actually in the cache. Every mask bit that becomes zero increases the number of cache lines that the corresponding stream register specifies as being in the cache, reducing the effectiveness of the stream register filtering. In the limit, the mask register becomes all zeros and every possible address is included in the register and considered to be in the cache.

Loss of accuracy is a disadvantage common to every snoop filtering technique that uses much less storage than the cache tag array. Snoop registers are specifically intended to remain accurate for strided streams, but they will not fare well with random addresses. To overcome the progressive loss of accuracy, the stream register snoop filter includes a mechanism for resetting the registers back to their initial condition. As there is no efficient way to remove an address from the stream registers and guarantee correctness, the filter clears the registers whenever the cache has been completely replaced, and they begin accumulating addresses anew. We call this complete replacement (relative to some initial state) a “cache wrap”. The cache wrap detection logic monitors cache updates and determines when all of the cache lines present in the initial state have been overwritten. To do this, information must be provided by the L1 cache, either in the form of individual replacement notifications, or as a single event indicating that a wrap has occurred.

At that point, all of the stream registers are copied to a second “history” set of registers and masks and the “active” stream registers are all cleared to begin accumulating cache line load addresses anew. In addition, the state of the cache at the time of the wrap becomes the new initial state for the purpose of detecting the next cache wrap. The stream registers in the history set are never updated. However, they are treated the same as the active set by the Port Filters when deciding whether a snoop address could be in the cache. Their purpose is to make up for the fact that individual cache sets wrap at different times, but never survive two cache wraps.

The stream registers exploit periodic cache wrapping in order to refresh, and they rely upon knowing when the wraps occur. The second requirement is not difficult to implement, but does require a modified cache that either indicates all replacements, or tracks them from some point in time and indicates when all lines have been replaced.

Periodic cache wrapping is not guaranteed, but occurs frequently in practice. Caches with a round-robin replacement policy, such as those of the IBM PowerPC cores used in the Blue Gene/L and Blue Gene/P supercomputers, wrap relatively frequently. Specific wrapping behavior is a function of replacement policy and workload behavior. In either case, it is statistically possible that wrapping occurs infrequently, or never (i.e. a particular cache set is seldom or never used), but this pathological situation is not a threat to correctness. If this is a serious concern, then the stream register architecture could be extended to include a full cache invalidation and stream register reset when no wrap occurs for a long period of time.

5 Experimental Methodology

The experiments in this paper represent our top-down approach of validating the stream register ideas and showing that stream registers are effective for a broad range of applications. Our methodology trades detail for the ability to process huge traces of several applications. We feel that this is the best approach because the order of memory accesses is the critical factor in determining snoop filter effectiveness. That is, the order of a snoop request for some address relative to a snoop filter acquiring that address is what determines whether the filter rejects the address or not. Other papers (such as JETTY [9]) relate filter effectiveness to performance gain and power reduction. For example, the JETTY paper estimates that L2 cache snooping alone accounts for 33% of the total power of a typical 4-way SMP.

For our experiments, we used several applications from the publicly-available SPLASH-2 [20] benchmark suite. These are well known benchmarks containing shared-memory applications that have driven much research on memory system architectures and cache-coherence protocols. We have chosen to use these publicly-available codes because they are good representatives for a wide range of scientific applications, which is where we expect to see the most significant impact of CMPs. We have run the kernels (LU, FFT, Cholesky, and Radix), and some of the applications (Barnes, Ocean, Raytrace, and Fmm).

For each of the benchmarks chosen, we have simulated a full, four-processor application run and collected the entire L1 data cache miss sequence to determine coherence snoop requests. Table 1 shows the benchmarks used, the total number of accesses to memory, and the average percentage of misses in the L1 caches. Whereas the hit rate in the local cache of a processor is high, the percentage of hits in the caches of all other processors (we refer to such processors as “remote”) is very low.

Table 1 shows that across all benchmarks virtually all coherency snoop requests will miss in the remote caches, this representing the total coherency snoop filter opportunity. Such small hit rates are due to the relatively small (32KB) first-level caches and highlight the importance of snoop filtering for CMP cache coherence, particularly when maintained between L1 caches. Although few snoops hit, the ones that do are essential and the ones that do not should be filtered. We also list the total number of snoop requests generated by all four processors collectively (i.e. the total number of writes).

We used a custom simulator written with Augmint [21] to collect the memory access traces. Augmint is a public-domain, execution-driven, multiprocessor simulation environment for Intel x86 architectures, running UNIX or Windows. Augmint does not include a memory backend, thus requiring users to develop one from scratch. We modeled the L1 data caches of four PowerPC 440 processors [22] and an ideal memory system below that (since we were only concerned with the order of accesses and their effect on snoop filtering rates). Then we collected a trace of all memory references, including the source processor and address.

We developed a custom back-end simulator to process the traces and produce the results in this paper. Because we wanted to measure the relative effectiveness of snoop filters over very long traces, we were not concerned with cycle accuracy. We were only concerned with the order of accesses and their effect upon the snoop filters and caches. Therefore, the trace entries are processed in order, and they have an instant, atomic

Table 1. SPLASH-2 benchmark characteristics. The low remote cache hit rate shows that almost all invalidation snoops are useless and can be eliminated.

Benchmark	Input parameters	Accesses to memory	Local cache hit rate	Remote cache hit rate	Total coherency accesses
Barnes	16K particles	1,602,120,476	99.73%	0.00047%	1,968,916,971
FFT	256K points	58,481,113	97.12%	0.000057%	52,627,671
LU	512 matrix	202,643,933	99.24%	0.000088%	204,434,958
Ocean	258 x 258 ocean	310,234,016	93.36%	0.03%	143,647,839
Cholesky	tk15.O	678,266,460	99.43%	0.00043%	614,572,560
FMM	16K particles	2,084,764,684	99.76%	0.00016%	2,976,937,884
Radix	10M keys	2,716,061,135	99.48%	0.00068%	3,491,931,132
Raytrace	car	404,977,091	98.43%	0.018%	358,731,051

effect upon the simulated caches and snoop filters. This simplification allowed us to compare many different alternative architectures, while exposing the significant trends. As a result, however, we could not measure execution times.

The back-end simulator models the private L1 data caches and snoop filters of four processors. We model the cache of the PowerPC 440 embedded processor, which is the building block for the Blue Gene/L supercomputer [8]. Each cache is 32KB in size and has a line size of 32 bytes. The caches are organized as 16 sets, each of which has 64 ways and utilizes a round-robin replacement policy.

While such high associativity may seem extreme, it is important to note that associativity is a characteristic that improves cache accuracy at the potential cost of cycle time. Overall, the gain in accuracy from high associativity can compensate for ever-increasing memory latencies, especially in a CMP where the overall performance is not as closely coupled to the cycle time as it is in a uniprocessor. Round-robin replacement is a viable, perhaps necessary, choice for a highly-associative cache because of its implementation simplicity (compared to LRU). Because we were only concerned with the order of accesses and their effect on snoop filtering rates, we did not model the memory system below the L1 caches.

The back-end simulator responds to loads and stores as follows:

- LOAD: Update the local cache. Then update the stream registers (insert the load address) and the snoop caches (delete the load address) of the local snoop filter.
- STORE: Update the local cache. Then update the three remote snoop filters, which includes a lookup to see if the snoop resulting from the store should be filtered, and a snoop cache update if it is not. Finally, propagate snoop invalidations to all remote caches for which the snoop request was not filtered.

6 Experiments and Simulation Results

We are interested in exploring the snoop filter design space to find the best compromise that yields a good filtering rate. We studied the impact of several design parameters. For stream registers, we considered their number, the replacement policy, and the empty affinity. For the snoop cache, we considered the number of entries and the size of the

valid line vector. The goal is to eliminate as many unnecessary coherence requests as possible, thus improving performance and reducing power dissipation. We also analyze the impact of using both snoop filter units together to exploit their combined strength, and explore several configurations to identify the optimal design point.

6.1 Stream Register Size

In order to determine the optimal number of stream registers, we have varied their number exponentially from 4 to 32, as shown in Figure 3. Not surprisingly, our experiments show that more stream registers filter a higher percentage of coherence snoop requests. But even when using only eight stream registers, we filter more than 90% of all snoop requests for three benchmark applications.

We observed that the effect of increasing the number of stream registers is not linear with respect to the snoop filter rate. Choosing only four stream registers is clearly a bad policy. For the SPLASH-2 benchmarks, selecting 8 or 16 stream registers seems to be the best compromise, whereas 32 stream registers (which doubles the area compared to 16 stream registers) only increases the snoop filter rate significantly for one benchmark.

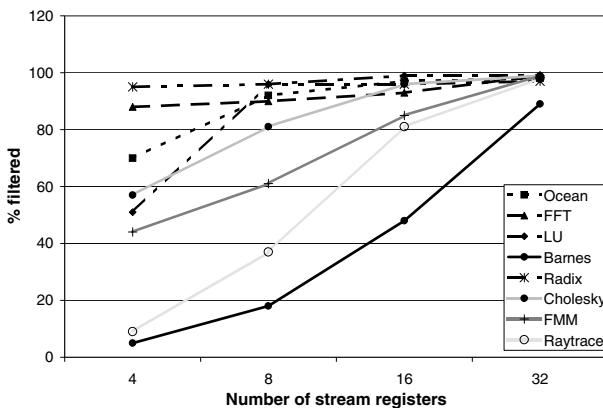


Fig. 3. Stream register filter behavior. Percentage of snoops filtered as the number of stream registers is increased.

6.2 Stream Register Update Policy

As previously described, every cache line load address is added to exactly one stream register. The register selected for update depends on the stream register update policy and the empty affinity value. We have evaluated two different selection policies:

- minimal Hamming distance, and
- most matching upper bits (MMUB).

For the minimal Hamming distance selection policy, we calculate the Hamming distance between each new load address and all stored values in the stream registers combined with their paired masks so that only relevant bits (i.e. bits that are 1 in the mask)

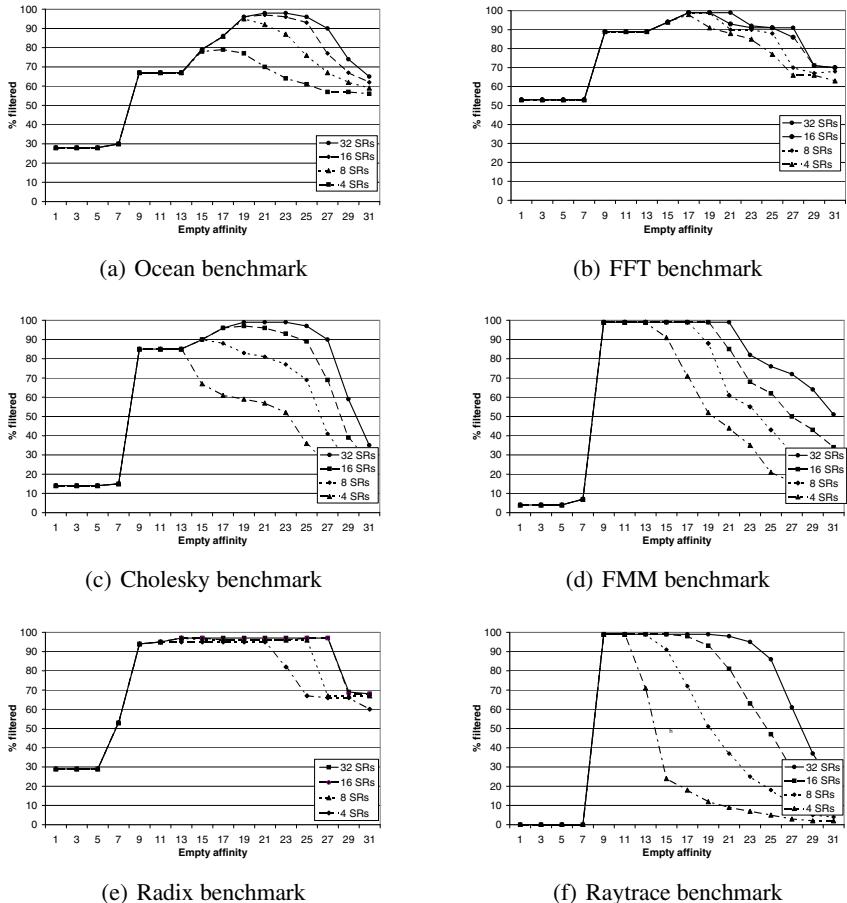


Fig. 4. Percentage of snoops filtered as the empty affinity and number of stream registers is varied using the MMUB update policy

are considered. The affinity is the number of mask register bits that will be changed to 0, and the stream register with the minimum affinity is selected for update. In the MMUB update policy, we choose the stream register where the largest number of relevant upper bits match those of the line load address.

In our evaluation of stream register update policies, we have also varied the empty affinity value. As discussed in Section 4, empty affinity is the default threshold value assigned to an empty register. Figures 4(a) to 4(f) show the effect of varying the empty affinity for various stream register sizes using the MMUB update policy. If the empty affinity is set too low, empty stream registers are used to establish new streams even for memory accesses belonging to the same stream. As a result, the filter rate of the stream registers will be very low because few streams are captured. Similarly, setting the empty affinity value too high causes streams to share registers and obliterate each others mask bits, resulting in a low filter rate. When the empty affinity is increased

to more than 13, it starts to play a role in the filter rate, depending on the number of stream registers. For filters having a higher number of stream registers, a higher affinity value is advantageous because it allows for more sensitive stream determination. For configurations with a smaller number of stream registers, a lower affinity allows for the most effective stream discrimination. Overall, the optimal empty affinity value is about 19 for eight stream registers, and about 23 for 32 stream registers.

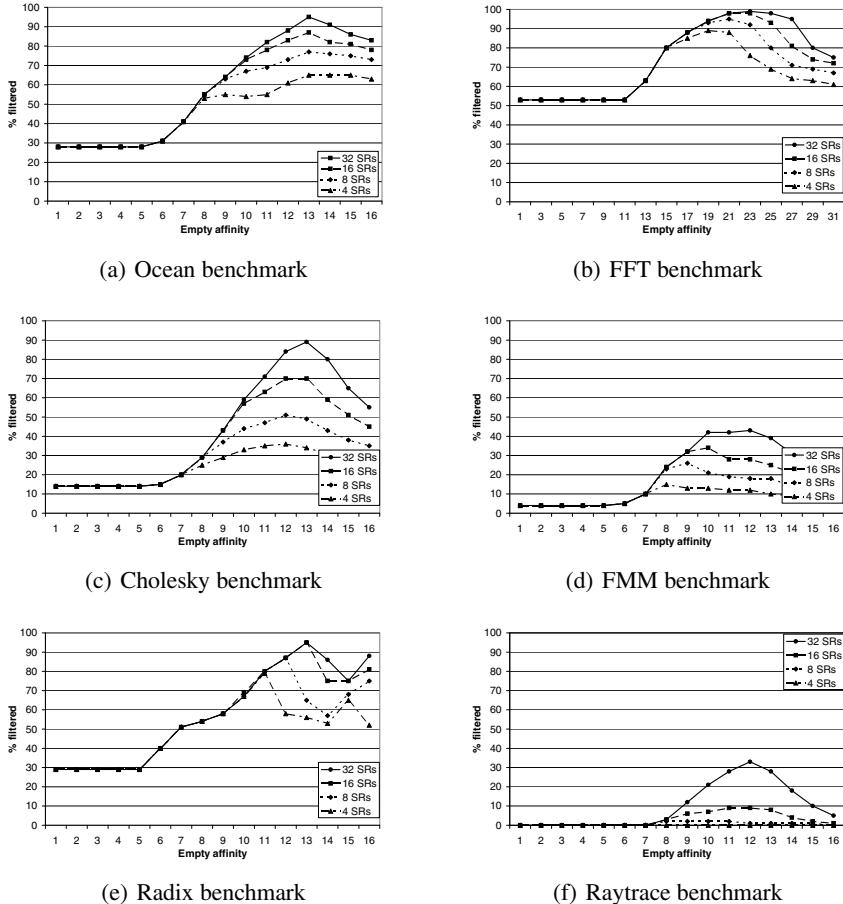


Fig. 5. Percentage of snoops filtered as the empty affinity and number of stream registers is varied using the minimum Hamming distance update policy

Figure 5 shows the effect of varying the empty affinity for various stream register sizes using the minimum Hamming distance update policy. Similar to the MMUB update policy, setting the empty affinity too low or too high causes the filter rate of the stream registers to be low. The optimal empty affinity number is between 19 and 25, depending on the number of stream registers in the filter and the application's memory access pattern.

Across all applications, the sensitivity of the filter rate to the empty affinity value seem to be less for the MMUB update policy when compared with the minimum Hamming distance policy. In addition, for some applications - like Raytrace and FMM - the MMUB update policy significantly outperforms the Hamming distance policy. While MMUB achieves almost 100% filtering for these applications, the Hamming distance policy reaches less than 40%, even for the largest configurations.

The MMUB policy has the advantage of ignoring low-order address bits when establishing streams. The minimum Hamming distance policy results in well-correlated addresses that differ in their low-order address bits being mapped to different stream registers, thereby causing a kind of pollution which limits effectiveness.

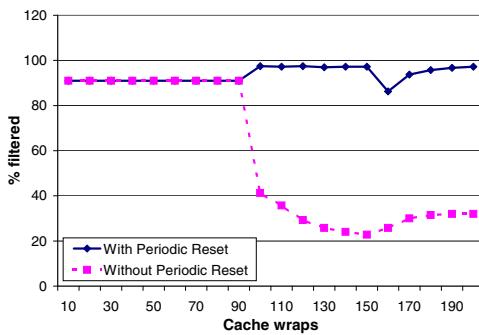


Fig. 6. Effectiveness of stream registers when implementing cache wrap for the FFT benchmark

6.3 Stream Register Clearing

As described in Section 4, the stream registers need to be cleared periodically in order to track changes in programs' L1 cache contents. The need for this is illustrated in Figure 6, which shows average, cumulative stream register effectiveness for a portion of the FFT benchmark running on a 4-processor CMP. During the course of the run, the caches wrapped 200 times.

Initially, the L1 working set is captured effectively by the stream registers and resetting provides no benefit. At around the 100th wrap, the working set changes, and the stream registers with reset can track the change. The stream registers without reset, however, forward many unnecessary snoops, causing their effectiveness to plummet. Although the stream registers without reset only become less accurate over time, the modest recovery in their effectiveness only indicates that the cache contents has changed in their favor.

6.4 Snoop Cache Size

In order to determine the optimal sizing for a snoop cache-based filter, we have varied two parameters: the number of lines, ranging from 4 to 64, and the number of bits used in the valid line vector, ranging from 1 to 64 (encoding 0 to 6 consecutive lines respectively). The results are illustrated in Figure 7 for several SPLASH-2 benchmarks.

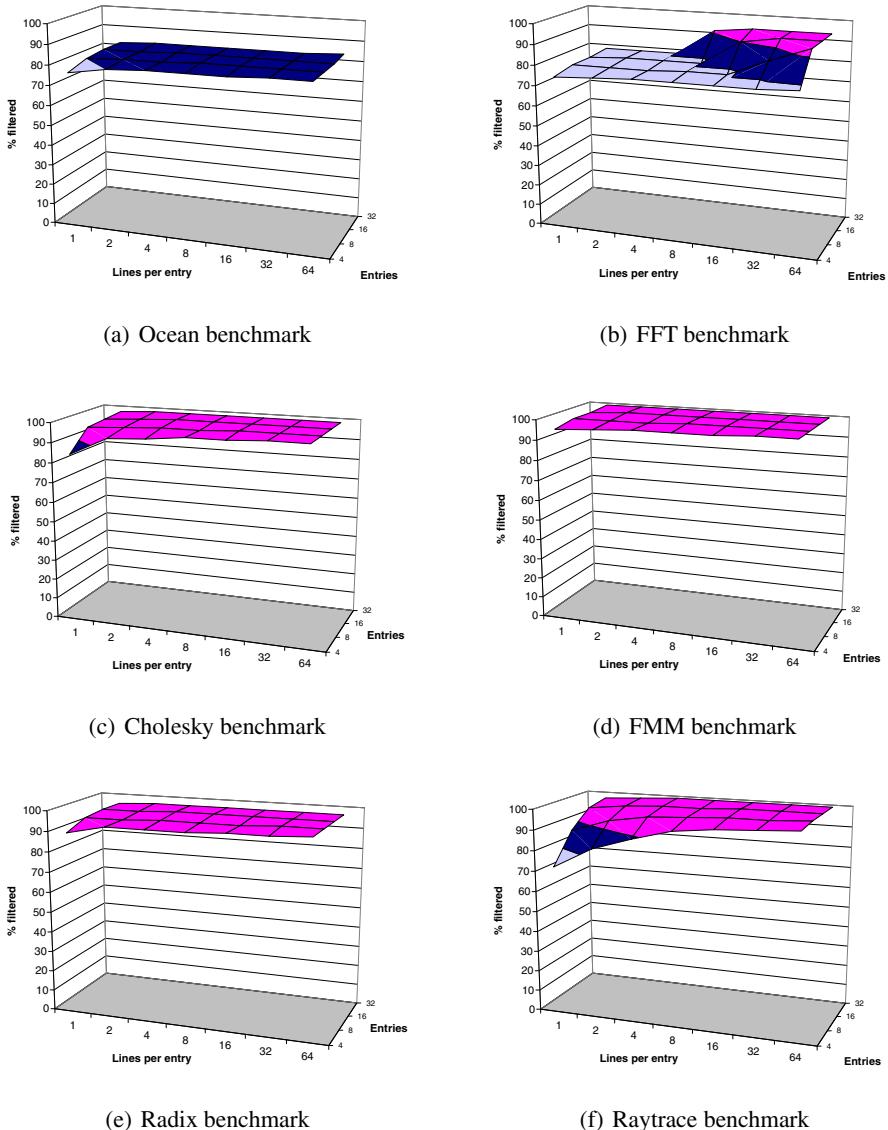


Fig. 7. Snoop cache filter behavior. Percentage of snoops filtered as the number of entries and the number of lines per entry is varied.

Our experiments show that using filters with a greater number of snoop cache lines and/or a longer valid line vector are more effective at filtering snoop requests. But even for relatively small snoop caches having only 4 cache lines each, and with valid line vectors of 32 bits, we reach the snoop cache filter limit across the benchmark applications. The filter limit varies for various applications from 82% for Ocean to 99% for FMM.

We observe that for each application, depending of its memory access pattern, the shape of the cache size/valid vector line length surface differs. By increasing only the number of cache lines or only the length of the valid line vector, the maximum filtering possibility for a particular application can be reached. For example, Ocean (Figure 7(a)) reaches its maximum filtering rate of 83% for snoop cache filter units with almost minimal sizing, having only 4 cache lines and as little as 2-bit-long valid line vectors. Similarly, FMM (Figure 7(d)) reaches its filtering maximum of 99% with a small configurations of only 8 cache lines and 8-bit long valid line vectors.

On the contrary, the FFT benchmark reaches maximum filtering only for bigger configurations. As illustrated in Figure 7(b), at least 8 cache lines and 64-bit long valid line vectors are needed to reach the maximum filtering rate of 93%. The memory access pattern of FFT requires a higher number of cache lines because it has a high number of streams, and a higher number of bits in the valid line vector because these streams are longer.

The optimal snoop filter configuration achieves near maximum filtering across all benchmarks, requiring a minimal number of latches for its implementation. This translates directly to a minimal number of aggregated bits. The dependency on the number of bits required by each of the snoop cache sizes explored is illustrated in Figure 8.

We observe that the effect of increasing the number of cache lines and valid line vector length is not linear with respect to the area requirements. Whereas the snoop filtering rate varied symmetrically with the number of cache lines and the valid line vector length, the area requirement increases significantly when increasing the number of cache lines. Thus, snoop cache configurations with lower numbers of cache lines and longer valid line vectors are better design points. For the SPLASH-2 benchmarks, selecting 8 cache lines with 32- or 16-bit valid line vectors seems to be the best compromise.

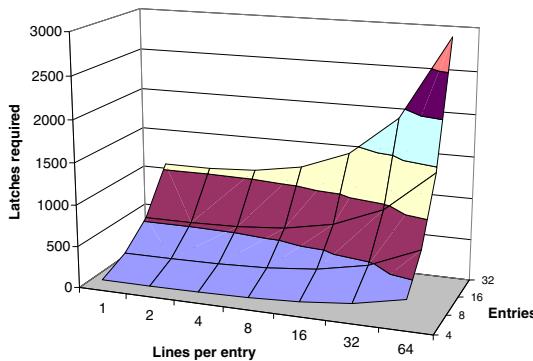


Fig. 8. Approximate number of latches required to implement a snoop filter unit with snoop caches, depending on the number of entries and the lines per entry

6.5 The Most Effective Combination

We have discussed and analyzed two snoop filters separately. As both filters cover different memory access patterns, the most effective filtering is achieved when putting the

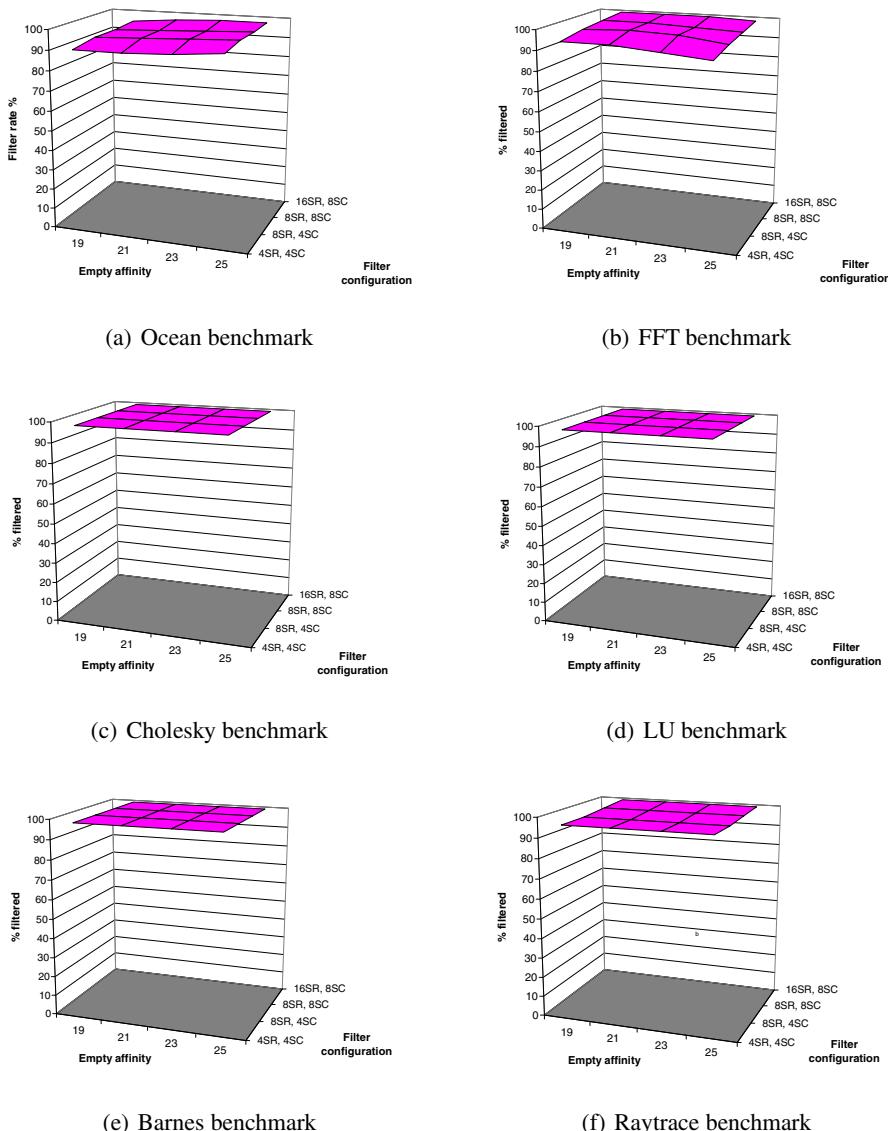


Fig. 9. Combined filter behavior. Percentage of snoops filtered for several stream register and snoop cache configurations as the empty affinity is varied in its most effective range.

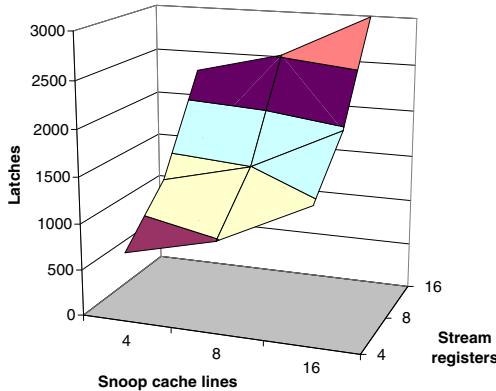


Fig. 10. Approximate number of latches required to implement a snoop filter unit with both snoop caches and stream registers, for several effective configurations

two filters together. We will show that using the combination of two filters, we can achieve high filtering rates even though each filter unit is quite small.

In order to determine the optimal sizing for our snoop filter, we have varied three parameters: the number of stream registers (4, 8, or 16), the number of snoop cache lines (4 or 8), and the empty affinity (in the most effective range from 19 to 25). We keep the snoop cache valid line vector at 32 bits in length. The results for several SPLASH-2 benchmarks are illustrated in Figures 9(a) to 9(f). Clearly, combining the two filtering techniques results in a highly effective combination across all of the benchmarks we studied.

Once again, we must consider the size of the combined filter in terms of latch count in order to determine the optimal configuration. Figure 10 shows how latch count varies with the number of stream registers and snoop cache lines for a valid line vector of 32 bits. The stream registers grow faster than the snoop caches, so the optimal configuration prefers larger snoop caches over more stream registers. The knee in the latch count and performance curves appears to be at 8 stream registers and 8 entries per snoop cache. By way of comparison, the L1 cache tags of the PowerPC 440 processor require more than an order of magnitude more latches.

7 Conclusion

With the emergence of commodity CMPs, we have entered the era of the SMP-on-a-chip. These high-performance systems will generate an enormous amount of shared memory traffic, so it will be important to eliminate as much of the useless inter-processor snooping as possible. In addition, power dissipation has become a major factor in chip density, so mechanisms to eliminate useless coherence actions will be important.

In this paper, we have described and evaluated a snoop filtering architecture that is appropriate for high-performance CMPs. Our architecture uses multiple,

complementary filtering techniques and parallelizes the filters so that they can handle snoop requests from all remote processors simultaneously.

We have described stream register snoop filtering, which uses a small number of registers to capture strided access streams. We explored the stream register and snoop cache design spaces using the SPLASH-2 benchmarks together with a custom trace generator and simulator.

Finally, we developed a highly-effective snoop filter that combines stream registers with snoop caches, and experimentally evaluated this design. We show that the combined filter can be very small in size, yet effective over all of the benchmarks we studied.

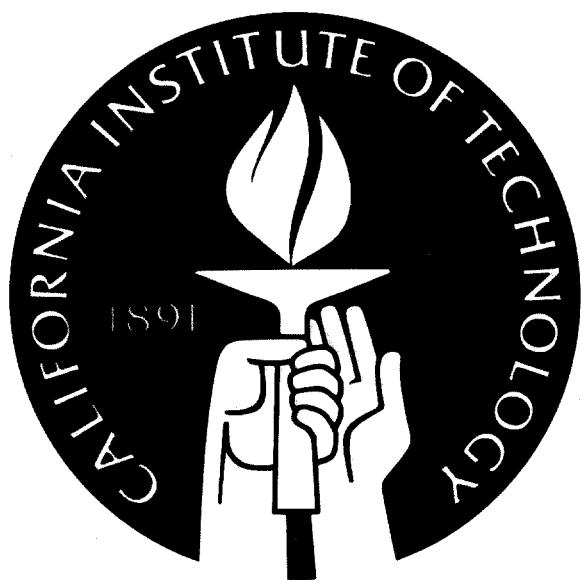
Acknowledgements

This work has been supported and partially funded by Argonne National Laboratory and Lawrence Livermore National Laboratory on behalf of the United States Department of Energy under Subcontract No. B554331.

References

1. Srinivasan, V., Brooks, D., Gschwind, M., Bose, P., Zyuban, V., Strenski, P., Emma, P.: Optimizing pipelines for power and performance. In: Proceedings of the 35th Annual International Symposium on Microarchitecture, pp. 333–344. ACM/IEEE, Istanbul/Turkey (2002)
2. Salapura, V., et al.: Power and performance optimization at the system level. In: Proceedings of the 2nd International Conference on Computing Frontiers, pp. 125–132. ACM, Ischia (2005)
3. Gonzalez, R., Horowitz, M.: Energy dissipation in general purpose microprocessors. IEEE Journal of Solid State Circuits 31(9), 1277–1284 (1996)
4. Dennard, R., Gaenslen, F., Yu, H.-N., Rideout, V., Bassous, E., LeBlanc, A.: Design of ion-implanted MOSFETs with very small physical dimensions. IEEE Journal of Solid-State Circuits, 256–268 (1974)
5. IBM Blue Gene Team: Overview of the IBM Blue Gene/P project. IBM Journal of Research and Development 52(1/2) (January 2008)
6. Intel, Intel quad-core technology, <http://www.intel.com/technology/quadcore>
7. Gschwind, M., Hofstee, H.P., Flachs, B., Hopkins, M., Watanabe, Y., Yamazaki, T.: Synergistic processing in Cell's multicore architecture. IEEE Micro 26(2), 10–24 (2006)
8. Bright, A.A., Ellavsky, M.R., Gara, A., Haring, R.A., Kopsay, G.V., Lembach, R.F., Marcella, J.A., Ohmacht, M., Salapura, V.: Creating the Blue Gene/L supercomputer from low power SoC ASICs. In: 2005 IEEE International Solid-State Circuits Conference on Digest of Technical Papers, pp. 188–189 (2005)
9. Moshovos, A., Memik, G., Falsafi, B., Choudhary, A.N.: JETTY: Filtering snoops for reduced energy consumption in SMP servers. In: Proceedings of the 7th International Symposium on High-Performance Computer Architecture, pp. 85–96 (2001)
10. Briggs, F., Chittor, S., Cheng, K.: Micro-architecture techniques in the Intel E8870 scalable memory controller. In: Proceedings of the 3rd Workshop on Memory Performance Issues, pp. 30–36 (June 2004)
11. Kant, K.: Estimation of invalidation and writeback rates in multiple processor systems, <http://kkant.gamerspace.net/papers/inval.pdf>
12. Aono, F., Kimura, M.: The Azusa 16-way Itanium server. IEEE Micro 20(5), 54–60 (2000)

13. Keltcher, C.N., McGrath, K.J., Ahmed, A., Conway, P.: The AMD opteron processor for multiprocessor servers. *IEEE Micro* 23(2), 66–76 (2003)
14. Chinthamani, S., Iyer, R.: Design and evaluation of snoop filters for web servers. In: Proceedings of the 2004 Symposium on Performance Evaluation of Computer Telecommunication Systems (July 2004)
15. Ekman, S., Dahlgren, F., Stenstrom, P.: TLB and snoop energy-reduction using virtual caches in low-power chip-multiprocessors. In: Proceedings of the 2002 International Symposium on Low Power Electronics and Design, pp. 243–246 (August 2002)
16. Moshovos, A.: Regionscout: Exploiting coarse grain sharing in snoop-based coherence. In: Proceedings of the 32nd Annual International Symposium on Computer Architecture, pp. 234–245 (June 2005)
17. Saldanha, C., Lipasti, M.: Power efficient cache coherence. In: Proceedings of the Workshop on Memory Performance Issues (June 2001)
18. Salapura, V., Blumrich, M., Gara, A.: Design and implementation of the Blue Gene/P snoop filter. In: Proceedings of the 14th International Symposium on High-Performance Computer Architecture, pp. 5–14 (February 2008)
19. Salapura, V., Blumrich, M., Gara, A.: Improving the accuracy of snoop filtering using stream registers. In: Proceedings of the 8th MEDEA Workshop, pp. 25–32 (September 2007)
20. Woo, S., Ohara, M., Torrie, E., Singh, J., Gupta, A.: The SPLASH-2 programs: Characterization and methodological considerations. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 24–36. ACM, New York (1995)
21. Nguyen, A.-T., Michael, M., Sharma, A., Torrellas, J.: The Augmint multiprocessor simulation toolkit for Intel x86 architectures. In: Proceedings of 1996 International Conference on Computer Design, pp. 486–490 (October 1996)
22. IBM, IBM PowerPC 440 product brief (July 2006), http://www-306.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_440_EMBEDDED_Core



The Torus Routing Chip

William J. Dally
and
Charles L. Seitz

Computer Science Department
California Institute of Technology

5208:TR:86

The Torus Routing Chip

William J. Dally
Charles L. Seitz

Computer Science Department
California Institute of Technology

5208:TR:86

January 24, 1986

To be published in *Journal of Distributed Computing* vol. 1, no. 3, 1986.

Abstract

The torus routing chip (TRC) is a self-timed chip that performs deadlock-free *cut-through* routing in k -ary n -cube multiprocessor interconnection networks using a new method of deadlock avoidance called *virtual channels*. A prototype TRC with byte wide self-timed communication channels achieved on first silicon a throughput of 64Mbits/s in each dimension, about an order of magnitude better performance than the communication networks used by machines such as the Caltech Cosmic Cube or Intel iPSC. The latency of the cut-through routing of only 150ns per routing step largely eliminates message locality considerations in the concurrent programs for such machines. The design and testing of the TRC as a self-timed chip was no more difficult than it would have been for a synchronous chip.

The research described in this paper was sponsored in part by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597, in part by Intel Corporation, and in part by an AT&T Ph.D. fellowship.

Copyright © California Institute of Technology, 1986.

1 Introduction

Message-passing concurrent computers such as the Caltech Cosmic Cube [13] and Intel iPSC [6] consist of many processing *nodes* that interact by sending messages over communication channels between the nodes. We designed the torus routing chip (TRC) as a building block to construct high-throughput, low-latency k -ary n -cube interconnection networks for message-passing concurrent computers.

The TRC is a self-timed VLSI circuit that provides deadlock-free packet communications in k -ary n -cube (torus) networks [12] with up to $k = 256$ processors in each dimension. While intended primarily for $n = 2$ -dimensional networks, the chips can be cascaded to handle n -dimensional networks using $\lceil \frac{n}{2} \rceil$ TRC chips at each processing node. A prototype TRC has been laid out, fabricated, and tested.

Even if only two dimensions are used, the TRC can be used to construct concurrent computers with up to 2^{16} nodes. It would be very difficult to distribute a global clock over an array of this size [4]. To avoid this problem, the TRC is entirely self-timed [11], thus permitting each processing node to operate at its own rate with no need for global synchronization. Synchronization, when required, is performed by arbiters in the TRC.

To reduce the latency of communications that traverse more than one channel, the TRC uses *cut-through* [7] routing rather than *store-and-forward* routing. Instead of reading an entire packet into a processing node before starting transmission to the next node, the TRC forwards each byte of the packet to the next node as soon as it arrives. Cut-through routing thus results in a message latency that is the *sum* of two terms, one of which depends on the message length, L , and other of which depends on the number of communication channels traversed, D . Store-and-forward routing gives a latency that depends on the product of L and D . Another advantage of cut-through routing is that communications do not use up the memory bandwidth of intermediate nodes. A packet does not interact with the processor or memory of intermediate nodes along its route. Packets remain strictly within the TRC network until they reach their destination.

The TRC uses *virtual channels* to perform deadlock-free routing in torus networks. By splitting each physical channel into two virtual channels and making routing dependent on the virtual channel on which a message arrives, the TRC converts the cycle of channel dependencies in each dimension into a spiral.

This paper describes the considerations that went into the design of the TRC in a “top-down” order that starts with a formal discussion of the deadlock problem in Section 2. We develop a model of communications in multiprocessor interconnection networks and prove a strong theorem about deadlock. Based on this model, the concept of virtual channels is presented in Section 3. Sections 4 and 5 present the design of the TRC at the system and logical levels. Experimental results are reviewed in Section 6.

2 Deadlock-Free Routing

Deadlock in the interconnection network of a concurrent computer occurs when no message

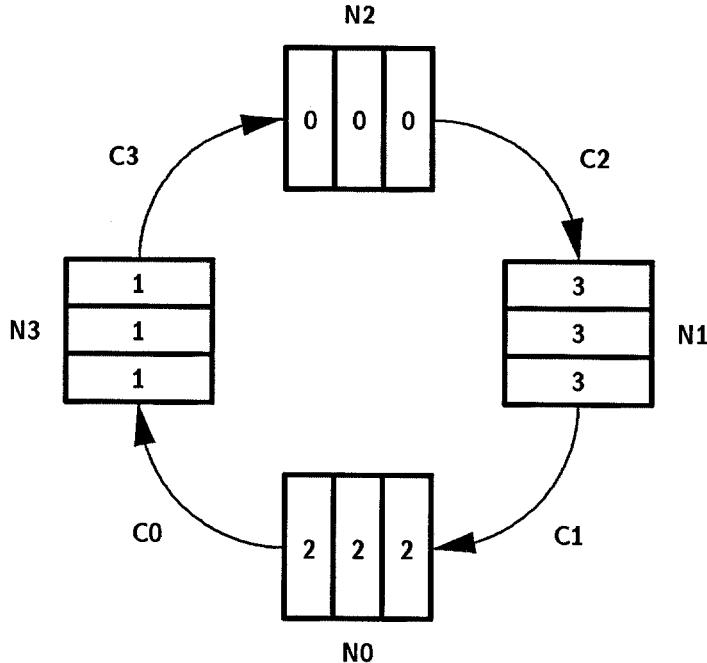


Figure 1: Deadlock in a 4-Cycle

can advance toward its destination because the queues of the message system are full [8]. Consider the example shown in Figure 1. The queues of each node in the 4-cycle are filled with messages destined for the opposite node. No message can advance toward its destination; thus the cycle is deadlocked. In this locked state, no communication can occur over the deadlocked channels until exceptional action is taken to break the deadlock.

The technique of virtual channels allows deadlock-free routing to be performed in any strongly-connected interconnection network [2]. This technique involves splitting physical channels on cycles into multiple virtual channels and then restricting the routing so the dependence between the virtual channels is acyclic.

Definition 1 A flow control digit or *flit* is the smallest unit of information that a queue or channel can accept or refuse. Generally a *packet* consists of many flits. Each packet carries its own routing information.

We have adopted this complication of standard terminology to distinguish between those flow control units that always include routing information - viz. packets - and those lower level flow control units that do not - viz. flits. The literature on computer networks [16] has been able to avoid this distinction between packets and flits because most networks include routing information with every flow control unit; thus the flow control units are packets. That is not the case in the interconnection networks used by message-passing concurrent computers such as the Caltech Cosmic Cube [13].

We assume the following:

1. Every packet arriving at its destination node is eventually consumed.

2. A node can generate packets destined for any other node.
3. The route taken by a packet is determined only by its destination, and not by other traffic in the network.
4. A node can generate packets of arbitrary length. Packets will generally be longer than a single flit.
5. Once a queue accepts the first flit of a packet, it must accept the remainder of the packet before accepting any flits from another packet.
6. An available queue may arbitrate between packets that request that queue space, but may not choose amongst waiting packets.
7. Nodes can produce packets at any rate subject to the constraint of available queue space (source queued).

The following definitions develop a notation for describing networks, routing functions, and configurations.

Definition 2 An *interconnection network*, I , is a strongly connected *directed graph*, $I = G(N, C)$. The vertices of the graph, N , represent the set of processing nodes. The edges of the graph, C , represent the set of communication channels. Associated with each channel, c_i , is a queue with capacity $\text{cap}(c_i)$. The source node of channel c_i is denoted s_i and the destination node d_i .

Definition 3 A *routing function* $\mathbf{R} : C \times N \mapsto C$ maps the current channel, c_c , and destination node, n_d , to the next channel, c_n , on the route from c_c to n_d , $\mathbf{R}(c_c, n_d) = c_n$. A channel is not allowed to route to itself, $c_c \neq c_n$. Note that this definition restricts the routing to be memoryless in the sense that a packet arriving on channel c_c destined for n_d has no memory of the route that brought it to c_c . However, this formulation of routing as a function from $C \times N$ to C has more memory than the conventional definition of routing as a function from $N \times N$ to C . Making routing dependent on the current channel rather than the current node allows us to develop the idea of channel dependence. Observe also that the definition of \mathbf{R} precludes the route from being dependent on the presence or absence of other traffic in the network. \mathbf{R} describes strictly deterministic and non-adaptive routing functions.

Definition 4 A *channel dependency graph*, D , for a given interconnection network, I , and routing function, \mathbf{R} , is a directed graph, $D = G(C, E)$. The vertices of D are the channels of I . The edges of D are the pairs of channels connected by \mathbf{R} :

$$E = \{(c_i, c_j) | \mathbf{R}(c_i, n) = c_j \text{ for some } n \in N\}. \quad (1)$$

Since channels are not allowed to route to themselves, there are no 1-cycles in D .

Definition 5 A *configuration* is an assignment of a subset of N to each queue. The number of flits in the queue for channel c_i will be denoted $\text{size}(c_i)$. If the queue for channel c_i contains a flit destined for node n_d , then $\text{member}(n_d, c_i)$ is true. A configuration is legal if

$$\forall c_i \in C, \text{size}(c_i) \leq \text{cap}(c_i). \quad (2)$$

Definition 6 A *deadlocked configuration* for a routing function, \mathbf{R} , is a non-empty legal configuration of channel queues such that

$$\forall c_i \in C, (\forall n \ni \text{member}(n, c_i), n \neq d_i \text{ and } c_j = \mathbf{R}(c_i, n) \Rightarrow \text{size}(c_j) = \text{cap}(c_j)) \quad (3)$$

In this configuration no flit is one step from its destination, and no flit can advance because the queue for the next channel is full. A routing function, \mathbf{R} , is *deadlock-free* on an interconnection network, I , if no deadlock configuration exists for that function on that network.

Theorem 1 A routing function, \mathbf{R} , for an interconnection network, I , is deadlock-free iff there are no cycles in the channel dependency graph, D .

Proof:

\Rightarrow Suppose a network has a cycle in D . Since there are no 1-cycles in D , this cycle must be of length two or more. Thus one can construct a deadlocked configuration by filling the queues of each channel in the cycle with flits destined for a node two channels away, where the first channel of the route is along the cycle.

\Leftarrow Suppose a network has no cycles in D . Since D is acyclic one can assign a total order to the channels of C so that if $(c_i, c_j) \in E$ then $c_i > c_j$. Consider the least channel in this order with a full queue, c_l . Every channel, c_n , that c_l feeds is less than c_l , and thus does not have a full queue. Thus, no flit in the queue for c_l is blocked, and one does not have deadlock. ■

3 Virtual Channels

Now that we have established this if-and-only-if relationship between deadlock and the cycles in the channel dependency graph, we can approach the problem of making a network deadlock-free by breaking the cycles. We can break such cycles by splitting each physical channel along a cycle into a group of *virtual channels*. Each group of virtual channels shares a physical communication channel; however, each virtual channel requires its own queue.

Consider for example the case of a unidirectional four-cycle shown in Figure 2A, $N = \{n_0, \dots, n_3\}$, $C = \{c_0, \dots, c_3\}$. The interconnection graph I is shown on the left and the dependency graph D is shown on the right. We pick channel c_0 to be the dividing channel of the cycle and split each channel into high virtual channels, c_{10}, \dots, c_{13} , and low virtual channels, c_{00}, \dots, c_{03} , as shown in Figure 2B.

Packets at a node numbered less than their destination node are routed on the high channels, and packets at a node numbered greater than their destination node are routed on the low channels. Channel c_{00} is not used. We now have a total ordering of the virtual channels

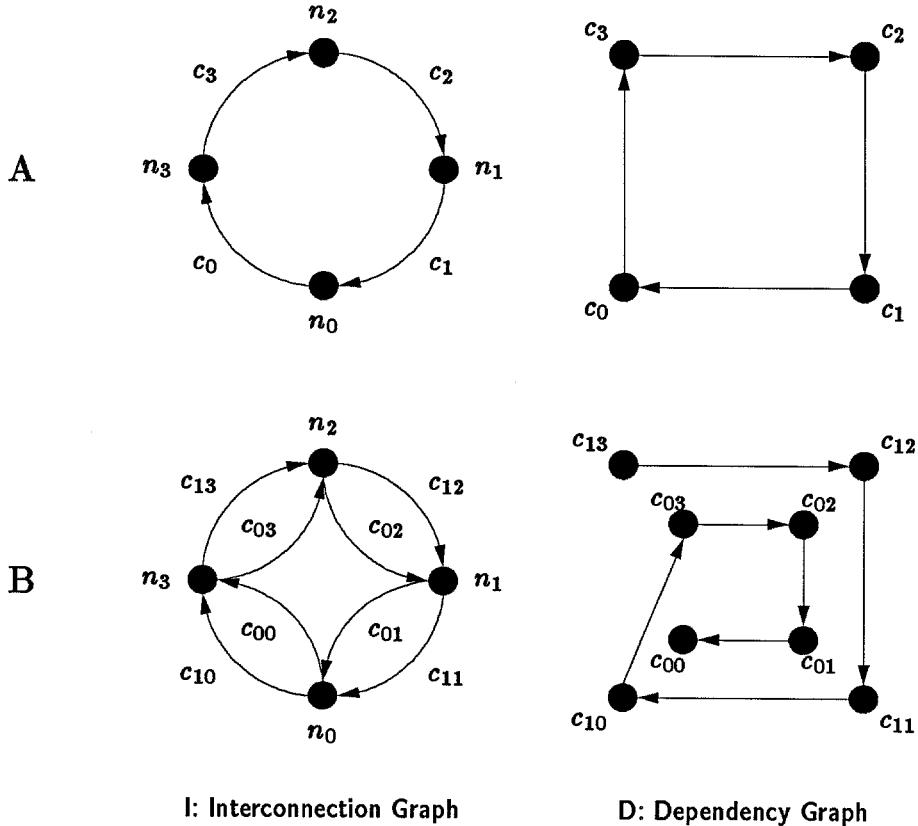


Figure 2: Breaking Deadlock with Virtual Channels

according to their subscripts: $c_{13} > c_{12} > c_{11} > c_{10} > c_{03} > c_{02} > c_{01}$. Thus, there is no cycle in D , and the routing function is deadlock-free. In [2] this technique is applied to construct deadlock-free routing functions for k -ary n -cubes, cube-connected cycles, and shuffle-exchange networks. In each case virtual channels are added to the network and the routing is restricted to route packets in order of decreasing channel subscripts. In the next two sections, the routing function for k -ary n -cubes is developed into a chip.

Many deadlock-free routing algorithms have been developed for store-and-forward computer communications networks [5]. These algorithms are all based on the concept of a *structured buffer pool*. The packet buffers in each node of the network are partitioned into classes, and the assignment of buffers to packets is restricted to define a partial order on buffer classes. The structured buffer pool method has in common with the virtual channel method that both prevent deadlock by assigning a partial order to resources. The two methods differ in that the structured buffer pool approach restricts the assignment of buffers to packets while the virtual channel approach restricts the routing of messages. Either method can be applied to store-and-forward networks, but the structured buffer pool approach is not directly applicable to cut-through networks, since the flits of a packet cannot be interleaved.

4 System Design

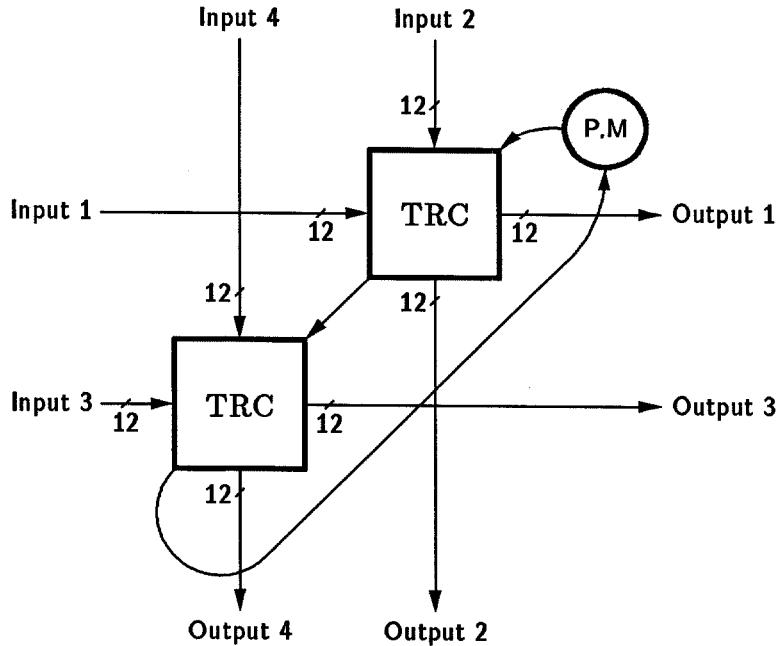


Figure 3: A Dimension 4 Node

The torus routing chip (TRC) can be used to construct arbitrary k -ary n -cube interconnection networks. Each TRC routes packets in two dimensions, and the chips are cascadable as shown in Figure 3 to construct networks of dimension greater than two. The first TRC in each node routes packets in the first two dimensions and strips off their address bytes before passing them to the second TRC. This next chip then treats the next two bytes as addresses in the next two dimensions and routes packets accordingly. The network can be extended to any number of dimensions.

A block diagram of a two-dimensional message-passing concurrent computer constructed around the TRC is shown in Figure 4. Each node consists of a processor, its local memory, and a TRC. Each TRC in the torus is connected to its processor by a processor input channel and a processor output channel. Connections on the edges of the torus wrap around to the opposite edge. One can avoid the long end-around connection by folding the torus, as shown in Figure 5.

A *flit* in the TRC is a byte whose 8 bits are transmitted in parallel. The X and Y channels each consist of 8 data lines and 4 control lines. The 4 control lines are used for separate request/acknowledge signal pairs for each of two virtual channels. The processor channels are also 8 bits wide, but have only two control lines each.

The packet format is shown in Figure 6. A packet begins with two address bytes. The bytes contain the relative X and Y addresses of the destination node. The relative address in a given direction, say X, is a count of the number of channels that must be traversed in the X direction to reach a node with the same X address as the destination. After the address comes the data field of the packet. This field may contain any number of *non-zero* data bytes. The packet is terminated by a zero tail byte. Later versions of the TRC may use an extra bit to tag the tail of a packet, and might also include error checking.

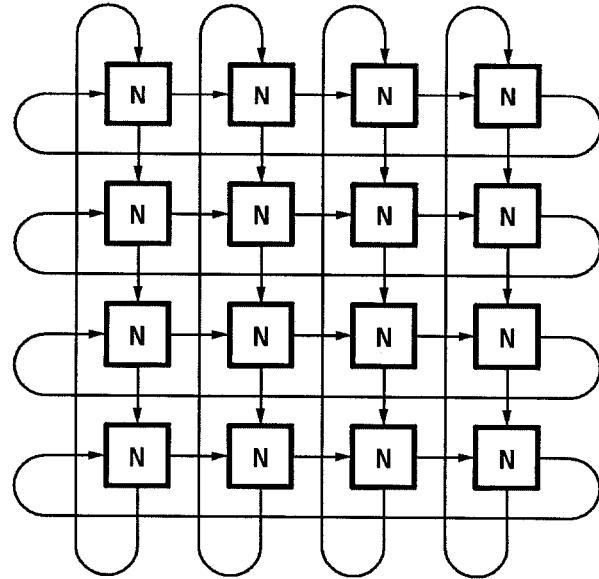


Figure 4: A Torus System

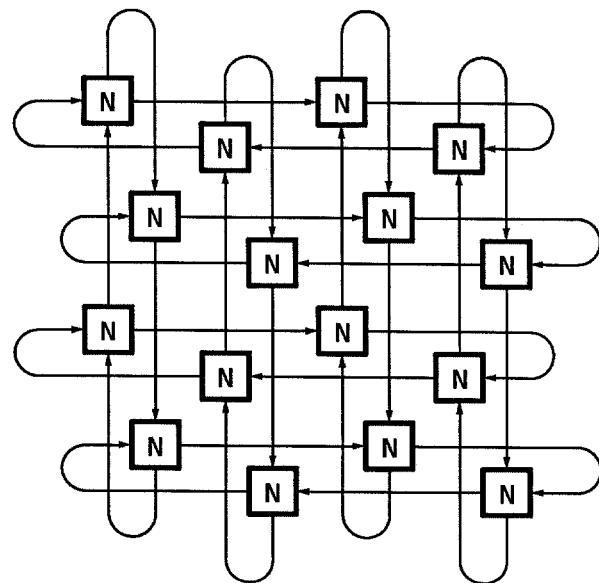


Figure 5: A Folded Torus System

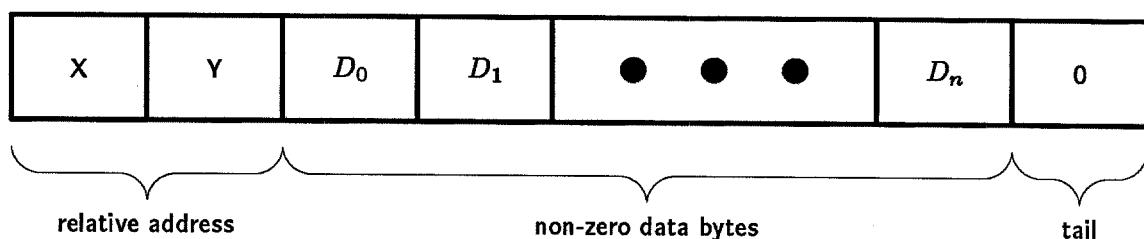


Figure 6: Packet Format

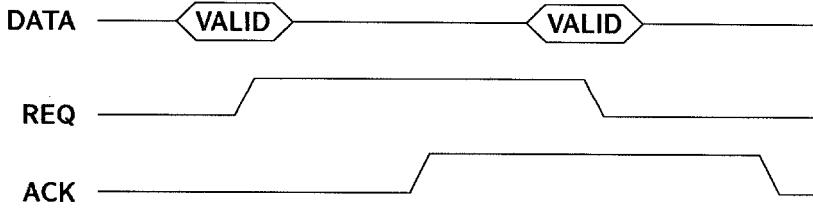


Figure 7: Virtual Channel Protocol

The TRC network routes packets first in the X direction, then in the Y direction. Packets are routed in the direction of decreasing address, decrementing the relative address at each step. When the relative X address is decremented to zero, the packet has reached the correct X coordinate. The X address is then stripped from the packet, and routing is initiated in the Y dimension. When the Y address is decremented to zero, the packet has reached the destination node. The Y address is then stripped from the packet, and the data and tail bytes are delivered to the node.

Each of the X and Y physical channels is multiplexed into two virtual channels. In each dimension packets begin on virtual channel 1. A packet remains on virtual channel 1 until it reaches its destination or address zero in the direction of routing. After a packet crosses address zero it is routed on virtual channel 0. The address 0 origin of the torus network in X and Y is determined by two input pins on the TRC. The effect of this routing algorithm is to break the channel dependency cycle in each dimension into a two-turn spiral similar to that shown in Figure 2. Packets enter the spiral on the outside turn and reach the inside turn only after passing through address zero.

Each virtual channel in the TRC uses the 2-cycle signaling convention shown in Figure 7. Each virtual channel has its own request (R) and acknowledge (A) lines. When $R = A$, the receiver is ready for the next flit (byte). To transfer information, the sender waits for $R = A$, takes control of the data lines, places data on the data lines, toggles the R line, and releases the data lines. The receiver samples data on each transition of R line. When the receiver is ready for the next byte, it toggles the A line.

The protocol allows both virtual channels to have requests pending. The sending end does not wait for any action from the receiver before releasing the channel. Thus, the other virtual channel will never wait longer than the data transmission time to gain access to the channel. Since a virtual channel always releases the physical channel after transmitting each byte, the arbitration is fair. If both channels are always ready, they will alternate bytes on the physical channel.

Consider the example shown in Figure 8. Virtual channel X1 gains control of the physical channel, transmits one byte of information, and releases the channel. Before this information is acknowledged, channel X0 takes control of the channel and transmits two bytes of information. Then X1, having been acknowledged, takes the channel again.

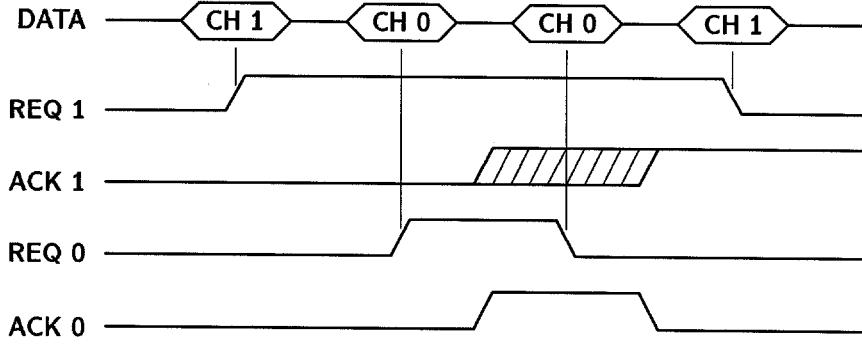


Figure 8: Channel Protocol Example

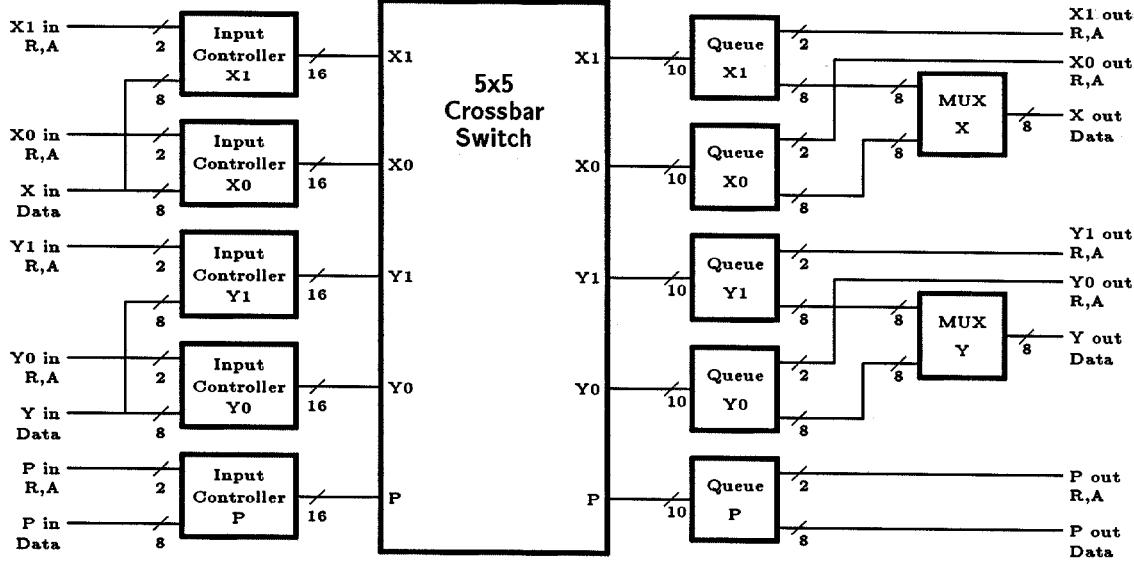


Figure 9: TRC Block Diagram

5 Logic Design

As shown in Figure 9, the TRC consists of five input controllers, a five by five crossbar switch, five output queues, and two output multiplexers. There is one input controller and one output controller for each virtual channel. The output multiplexers serve to multiplex two virtual channels onto a single physical channel.

The input controller is responsible for packet routing. When a packet header arrives, the input controller selects the output channel, adjusts the header by decrementing and sometimes stripping the byte, and then passes all bytes to the crossbar switch until the tail byte is detected.

The input controller, shown in Figure 10, consists of a datapath and a self-timed state machine. The datapath contains a latch, a zero checker, and a decrementer. A state

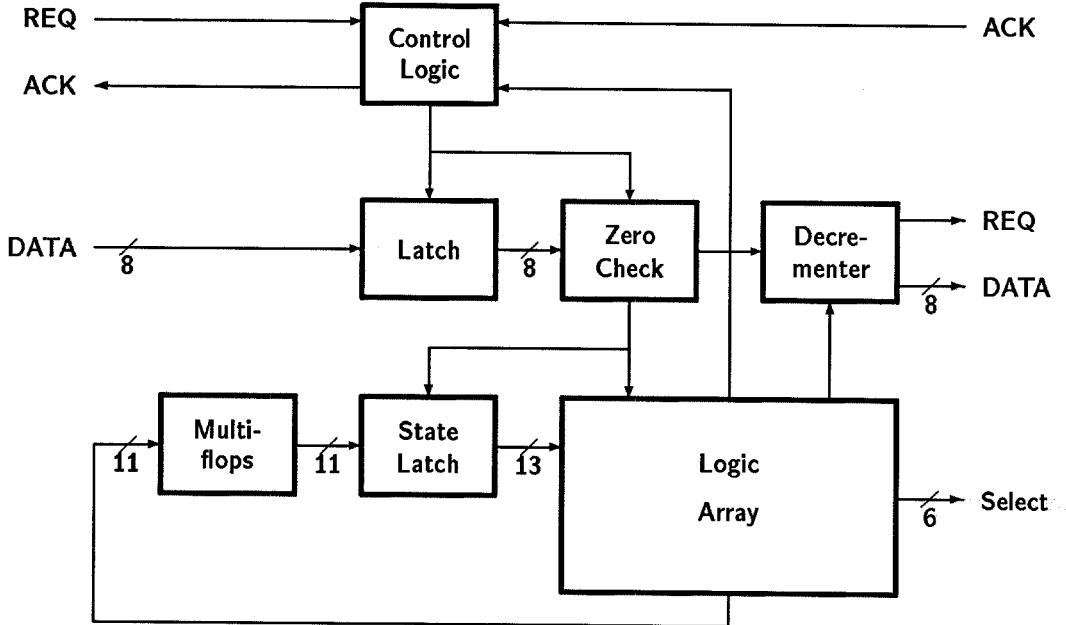


Figure 10: Input Controller Block Diagram

latch, logic array, and control logic comprise the state machine. When the request line for the channel is toggled, data is latched, and the zero checker is enabled. When the zero checker makes a decision, the logic array is enabled to determine the next state, the selected crossbar channel, and whether to strip, decrement, or pass the current byte. When the required operation has been completed, possibly requiring a round trip through the crossbar, the state and selected channel are saved in cross-coupled multi-flops and the logic array is precharged.

The input controller and all other internal logic operates using a 4-cycle self-timed signaling convention [11]. One function of the state machine control logic is to convert the external 2-cycle signaling convention into the on-chip 4-cycle signaling convention. The signaling convention is converted back to 2-cycle at the output pads.

The crossbar switch performs the switching and arbitration required to connect the five input controllers to the five output queues. A single crosspoint of the switch is shown in Figure 11. A two-input interlock (mutual-exclusion) element in each crosspoint arbitrates requests from the current input channel (row) with requests from all lower channels (rows). The interlock elements are connected in a priority chain so that an input channel must *win* the arbitration in the current row and all higher rows before gaining access to the output channel (column).

The output queues buffer data from the crossbar switch for output. The queues are each of length four. While a shorter queue would suffice to decouple input and output timing, the longer queue also serves to smooth out the variation in delays due to channel conflicts.

Each output multiplexer performs arbitration and switching for the virtual channels that share a common physical channel. As shown in Figure 12, a small self-timed state machine sequences the events of placing the data on the output pads, asserting request, and removing

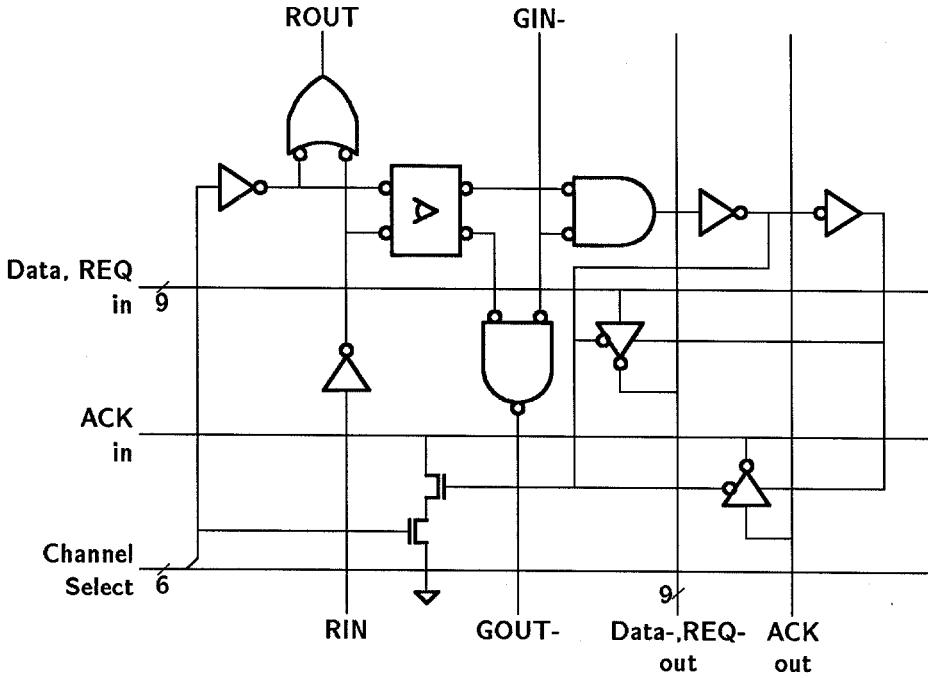


Figure 11: Crosspoint of the Crossbar Switch

the output data. An interlock element is used to resolve conflicts between channels for the data pads.

To interface the on-chip equipotential region to the off-chip equipotential region that connects adjacent chips, self-timed output pads (Figure 7.22 in [11]) are used. A Schmidt Trigger and exclusive-OR gate in each of these pads signals the state machine when the pad is finished driving the output. These completion signals are used to assure that the data pads are valid before the request is asserted and that the request is valid before the data is removed from the pads and the channel released.

6 Experimental Results

The design of the TRC began in August 1985. The chip was completely designed and simulated at the transistor level before any layout was performed. The circuit design was described using CNTK, a language embedded in C [3], and was simulated using MOSSIM [1]. A subtle error in the self-timed controllers was discovered at the circuit level before any time-consuming layout was performed. Once the circuit design was verified, the TRC was laid out in the new MOSIS scalable CMOS technology [17] using the Magic system [10]. A second circuit description was generated from the artwork and six layout errors were discovered by simulation of the extracted circuit. The verified layout was submitted to MOSIS for fabrication in September 1985.

The first batch of chips was completed the first week of December but failed to function because of fabrication errors. A second run of chips (same design), returned the second week of December, contained some fully functional chips.

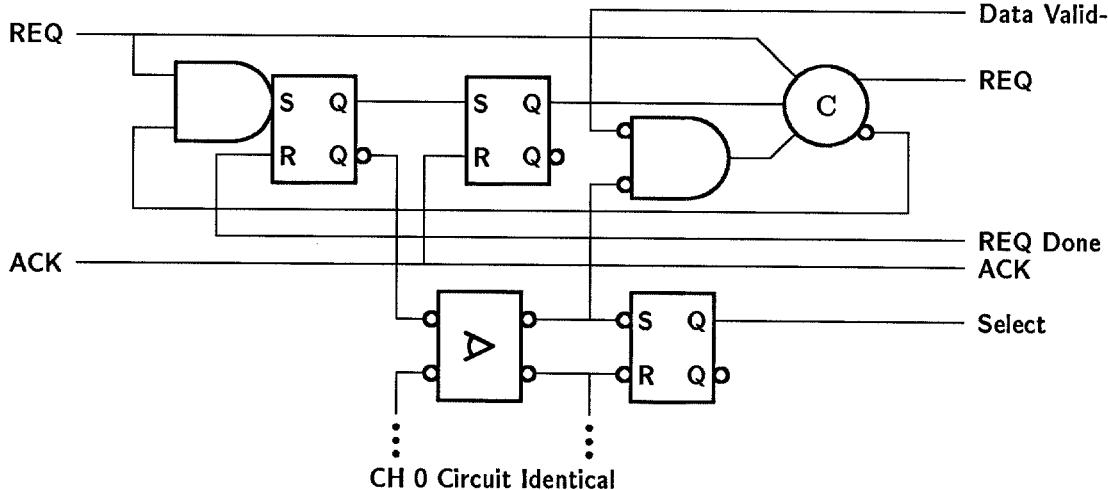


Figure 12: Output Multiplexer Control

Performance measurements on the chips are shown in Figure 14. To measure the maximum channel rate, the output request and acknowledge lines were tied together, and the input acknowledge was inverted and fed back into input request. In this configuration the chip runs at a maximum speed, shown in Figure 14A, of $\approx 4\text{MHz}$. This sluggish performance, about one fifth of what we expected, was traced to an overlooked critical path in the input controller. The chip still functioned correctly thanks to the self-timing.

The delays from input request to output request and input acknowledge, shown in Figure 14B, are 150ns and 250ns respectively. Data propagation time from input to output (not shown) was measured to be 60ns for both rising and falling edges. Thus data is set up 90ns ahead of the output request. Data hold time, shown in Figure 14C, is 20ns.

Tau model calculations suggest that a redesigned TRC should operate at 20MHz and have an input to output delay of 50ns. The redesign will involve decoupling the timing of the input controller by placing single stage queues between the input pads and input controller and between the input controller and the crossbar switch. The input controller will be modified to speed up critical paths.

7 Conclusion

This work was motivated by the ongoing design and implementation of experimental concurrent computers at Caltech and the investigation [15] of interconnection networks for these machines. A strong argument for a binary n -cube interconnection was the existence of the e-cube algorithm [9] for deadlock-free packet routing. Until the development of virtual channels, we knew of no comparable algorithm for cubes of higher arity. The TRC demonstrates the use of virtual channels to provide deadlock-free packet routing in k -ary n -cube multiprocessor communication networks.

Communication between nodes of a message-passing concurrent computer need not be slower than the communication between the processor and memory of a conventional sequen-

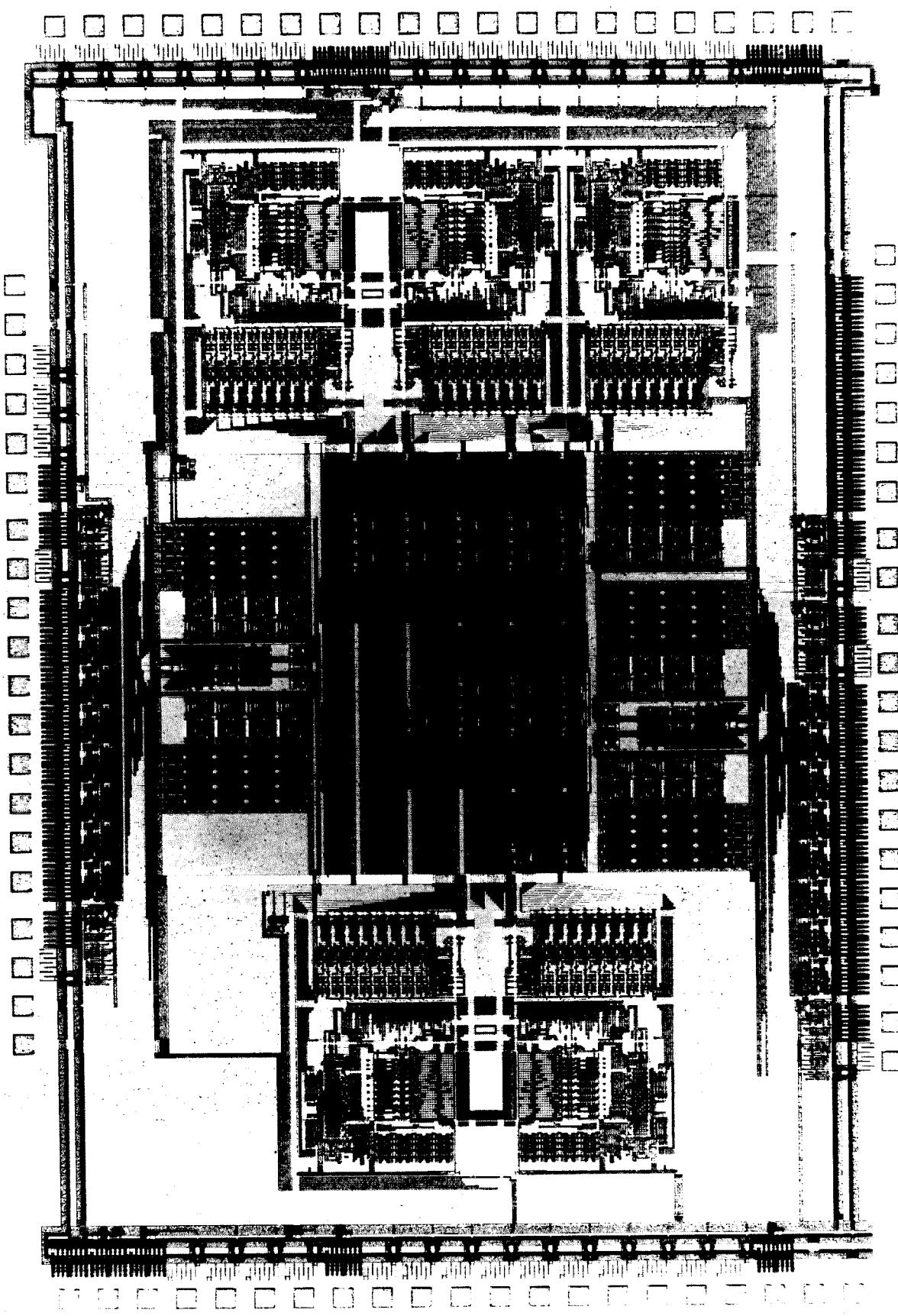


Figure 13: The Torus Routing Chip

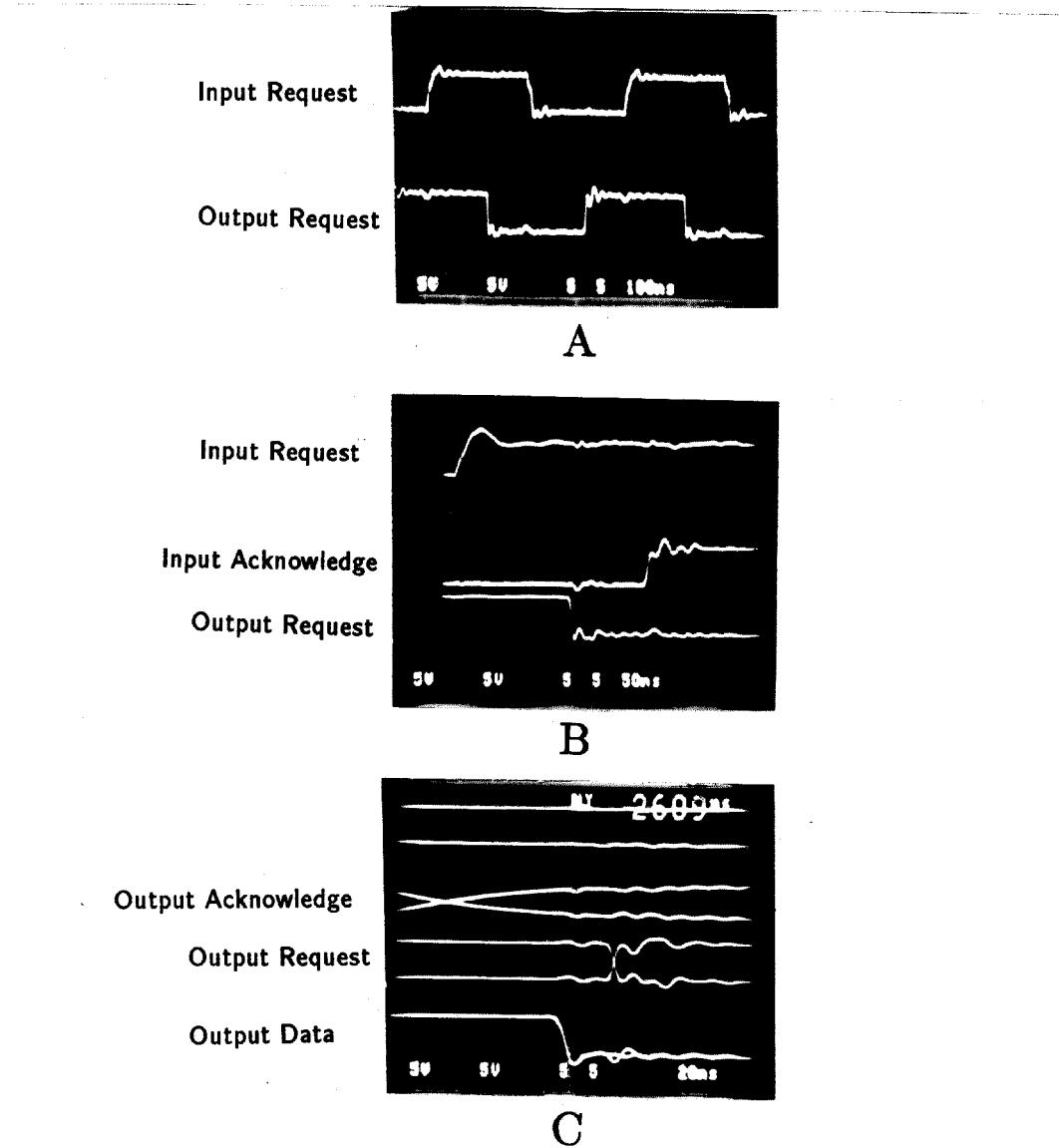


Figure 14: TRC Performance Measurements

tional computer. By using byte-wide datapaths and cut-through routing, the TRC provides node-to-node communication times that approach main memory access times of sequential computers. Communications across the diameter of a network, however, require substantially longer than a memory access time.

In spite of our past success in building machines using binary n -cube interconnection networks, there are some compelling reasons to experiment with machines using a torus network. First, the torus is easier to wire. Any network topology must be embedded in the plane for implementation. The torus maps naturally into the plane with all wires the same length; the cube maps into the plane in a less uniform way. Second, the torus more evenly distributes load to communication channels. When a cube is embedded in the plane, a saturated communication channel may run parallel to an idle channel. In the torus, by grouping these channels together to make fewer but higher bandwidth channels, the saturated channel can use all of the idle channel's capacity.

Compare, for example, a 256-node binary 8-cube with a 256-node 16-ary 2-cube (16×16 torus) constructed with the same bisection width. If the 8-cube uses single bit communication channels, 256 wires will pass through a bisection of the cube, 128 in each direction. Thus, with the same amount of wire we can construct a torus with 8-bit wide communication channels. Assuming the channels operate at the same rate ¹, by choosing the torus network we trade a 4-fold increase in diameter (from 8 to 32) for a 8-fold increase in channel throughput. In general, for a $N = 2^n$ node computer we trade a $\frac{2\sqrt{N}}{n}$ increase in diameter for a $\frac{\sqrt{N}}{2}$ increase in channel throughput.

We plan to use the TRC and its successors in future experimental concurrent computers. Our first machine will use the TRC along with commercial microprocessors and memory parts to construct a 2-dimensional torus of several hundred processors. In $3\mu\text{m}$ scalable CMOS technology the TRC measures $4.5\text{mm} \times 6.5\text{mm}$ with pads. After scaling to $1.6\mu\text{m}$ technology there will room on a single die to combine both the TRC and a simple processor. With further scaling some of the processor's local memory may be moved on-chip.

The TRC serves as still another counterexample to the myth that self-timed systems are more complex than synchronous systems. The design of the TRC is not significantly more complex than a synchronous design that performs the same function. As for speed, the TRC will certainly be faster than a synchronous chip since each chip can operate at its full speed with no danger of timing errors. A synchronous chip is generally operated at a slower speed that reflects the timing of a worst-case chip and adds a timing margin.

The real challenge in concurrent computing is software. The development of concurrent software is strongly influenced by available concurrent hardware. We hope that by providing machines with higher performance internode communication we will encourage concurrency to be exploited at a finer grain size in both system and application software.

¹This assumption favors the cube since some of its channels are quite long while the torus channels are uniformly short.

Acknowledgements

The authors thank Craig Steele, Don Speck and Bill Athas for their many helpful suggestions, Fritz Nordby for designing the pads used on the TRC, and Keith Solomon for his work on the layout of the input controller datapath.

References

- [1] Bryant, Randy, Schuster, Mike and Whiting, Doug, *MOSSIM II: A Switch-Level Simulator for MOS LSI User's Manual*, Caltech Technical Report 5033:TR:82, January 1983.
- [2] Dally, William J. and Seitz, Charles L., *Deadlock-Free Message Routing in Multiprocessor Interconnection Networks*, Dept. of Computer Science, California Institute of Technology, Technical Report 5206:TR:86, 1986.
- [3] Dally, William J. *CNTK: An Embedded Language for Circuit Description*, Dept. of Computer Science, California Institute of Technology, Display File, in preparation.
- [4] Fisher, A.L. and Kung, H.T., "Synchronizing Large VLSI Processor Arrays," *IEEE Transactions on Computers*, C-34(8), August 1985, pp. 734-740.
- [5] Gunther, Klaus D., "Prevention of Deadlocks in Packet-Switched Data Transport Systems," *IEEE Transactions on Communications*, Vol. COM-29, No. 4, April 1981, pp. 512-524.
- [6] Intel iPSC User's Guide, Intel Document No. 175455-001, August 1985.
- [7] Kermani, Parviz and Kleinrock, Leonard, "Virtual Cut-Through: A New Computer Communication Switching Technique," *Computer Networks*, Vol 3., 1979, pp. 267-286.
- [8] Kleinrock, Leonard, *Queueing Systems*, Wiley, 1976, Vol. 2, pp. 438-440.
- [9] Lang, Charles R., *The Extension of Object-Oriented Languages to a Homogeneous Concurrent Architecture*, Dept. of Computer Science, California Institute of Technology, Technical Report, 5014:TR:82, 1982, pp. 118-124.
- [10] Ousterhout, John K. et. al., "The Magic VLSI Layout System," *IEEE Design and Test of Computers*, 2(1), February 1985, pp. 19-30.
- [11] Seitz, Charles L., "System Timing", Chapter 7 in *Introduction to VLSI Systems*, C. A. Mead and L. A. Conway, Addison Wesley, 1980.
- [12] Seitz, Charles L., "Concurrent VLSI Architectures," *IEEE Transactions on Computers*, C-33(12), December 1984, pp. 1247-1265.
- [13] Seitz, Charles L., "The Cosmic Cube," *CACM*, 28(1), January 1985, pp. 22-33.
- [14] Seitz, Charles L. et. al., *The Hypercube Communications Chip*, Dept. of Computer Science, California Institute of Technology, Display File 5182:DF:85, March 1985.

[15] Steele, Craig S., *Placement of Communicating Processes on Multiprocessor Networks*, Dept. of Computer Science, California Institute of Technology, Technical Report 5184:TR:85, 1985.

[16] Tanenbaum, A. S., *Computer Networks*, Prentice Hall, 1981, pp. 15-21.

[17] Trotter, D., *Miss MOSIS Scalable CMOS Rules*, Version 1.2, 1985.

Very Large Instruction Word Architectures (VLIW Processors and Trace Scheduling)

Binu Mathew
{mbinu} @ {cs.utah.edu}

1 What is a VLIW Processor?

Recent high performance processors have depended on Instruction Level Parallelism (ILP) to achieve high execution speed. ILP processors achieve their high performance by causing multiple operations to execute in parallel, using a combination of compiler and hardware techniques. Very Long Instruction Word (VLIW) is one particular style of processor design that tries to achieve high levels of instruction level parallelism by executing long instruction words composed of multiple operations. The long instruction word called a MultiOp consists of multiple arithmetic, logic and control operations each of which would probably be an individual operation on a simple RISC processor. The VLIW processor concurrently executes the set of operations within a MultiOp thereby achieving instruction level parallelism. The remainder of this article discusses the technology, history, uses and the future of such processors.

2 Different Flavors of Parallelism

Improvements in processor performance come from two main sources: faster semiconductor technology and parallel processing. Parallel processing on multiprocessors, multicomputers and processor clusters has traditionally involved a high degree of programming effort in mapping an algorithm to a form that can better exploit multiple processors and threads of execution. Such reorganization has often been productively applied, especially for scientific programs. The general-purpose microprocessor industry on the other hand has pursued methods of automatically speeding up existing programs without major restructuring effort. This lead to the development of Instruction Level Parallel (ILP) processors that try to speed up program execution by overlapping the execution of multiple instructions from an otherwise sequential program.

A simple processor that fetches and executes one instruction at a time is called a simple scalar processor. A processor with multiple function units has the potential to execute several operations in parallel. If the decision about which operations to execute in an overlapped manner is made at run time by the hardware, it is called a super scalar processor. To a simple scalar processor, a binary program represents a plan of execution. The processor acts as an interpreter that executes the instructions in the program one at a time. From the point of view of a modern super scalar processor, an input program is more like a representation of an algorithm for which several different

plans of execution are possible. Each plan of execution specifies when and on which function unit each instruction from the instruction stream is to be executed.

Different types of ILP processors vary in the manner in which the plan of execution is derived, but it typically involves both the compiler and the hardware. In the current breed of high performance processors like the Intel Pentium and the MIPS R18000, the compiler tries to expose parallelism to the processor by means of several optimizations. The net result of these optimizations is to place as many independent operations as possible close to each other in the instruction stream. At run time, the processor examines several instructions at a time, analyses the dependences between instructions and keeps track of the availability of data and hardware resources for each instruction. It tries to schedule each instruction as soon as the data and function units it needs are available. The processor's decisions are complicated by the fact that memory accesses often have variable latencies that depend on whether a memory access hits in the cache or not. Since such processors decide which function unit should be allocated to which instruction as execution progresses, they are said to be dynamically scheduled. Often, as a further performance improvement, such processors allow later instructions that are independent to execute ahead of an earlier instruction which is waiting for data or resources. In that case the processor is said to be out of order.

Branches are common operations in general-purpose code. On encountering a branch, a processor must decide whether or not to take the branch. If the branch is to be taken, the processor must start fetching instructions from the branch target. To avoid delays due to branches, modern processors try to predict the outcome of branches and execute instructions from beyond the branch. If the processor predicted the branch incorrectly, it may need to undo the effects of any instructions it has already executed beyond the branch. If a super scalar processor uses resources that may otherwise go idle to execute operations the result of which may or may not be used, it is said to be speculative.

Out of order speculative execution comes at a significant hardware expense. The complexity and non-scalability of the hardware structures used to implement these features could significantly hinder the performance of future processors. An alternative solution to this problem is to simplify processor hardware and transfer some of the complexity of extracting ILP to the compiler and run time system—the solution explored by VLIW processors.

Joseph Fisher, who coined the acronym VLIW, characterized such machines as architectures which issue one long instruction per cycle, where each long instruction called a MultiOp consists of many tightly coupled independent operations each of which execute in a small and statically predictable number of cycles. In such a system, the task of grouping independent operations into a MultiOp is done by a compiler or binary translator. The processor freed from the cumbersome task of dependence analysis has to merely execute in parallel the operations contained within a MultiOp. This leads to simpler and faster processor implementations. In later sections, we will see how VLIW processors try to deal with the problems of branch and memory latencies and implement their own variant of speculative execution. But, first, we present a brief history of VLIW processors.

3 A Brief History of VLIW Processors

For various reasons which were appropriate at that time, early computers were designed to have extremely complicated instructions. These instructions made designing the control circuits for such computers difficult. A solution to this problem was microprogramming, a technique proposed by Maurice Wilkes in 1951. In a micro programmed CPU, each program instruction is considered a macroinstruction to be executed by a simpler processor inside the CPU. Corresponding to each macroinstruction, there will be a sequence of microinstructions stored in a microcode ROM in the CPU.

Horizontal microprogramming is a particular style of microprogramming where bits in a wide microinstruction are directly used as control signals within the processor. In contrast, vertical microprogramming uses a shorter microinstruction or series of microinstructions in combination with some decoding logic to generate control signals. Microprogramming became a popular technique for implementing the control unit of processors after IBM adopted it for the System/360 series of computers.

Even before the days of the first VLIW machines, there were several processors and custom computing devices that used a single wide instruction word to control several function units working in parallel. However these machines were typically hand-coded and the code for such machines could not be generalized to other architectures. The basic problem was that compilers at that time looked only within basic blocks to extract ILP. Basic blocks are often short and contain many dependences and therefore the amount of ILP that can be obtained inside a basic block is quite limited.

Joseph Fisher, a pioneer of VLIW, while working on PUMA, a CDC-6600 emulator was frustrated by the difficulty of writing and maintaining 64 bit horizontal microcode for that processor. He started investigating a technique for global microcode compaction—a method to generate long horizontal microcode instructions from short sequential ones. Fisher soon realized that the technique he developed in 1979, called trace scheduling, could be used in a compiler to generate code for VLIW like architectures from a sequential source since the style of parallelism available in VLIW is very similar to that of horizontal microcode. His discovery lead to the design of the ELI-512 processor and the Bulldog trace-scheduling compiler.

Two companies were founded in 1984 to build VLIW based mini supercomputers. One was Multiflow, started by Fisher and colleagues from Yale University. The other was Cydrome founded by VLIW pioneer Bob Rau and his colleagues. In 1987, Cydrome delivered its first machine, the 256 bit Cydra 5, which included hardware support for software pipelining. In the same year, Multiflow delivered the Trace/200 machine, which was followed by the Trace/300 in 1988 and Trace/500 in 1990. The 200 and 300 series used a 256 bit instruction for 7 wide issue, 512 bits for 14 wide issue and 1024 bits for 28 wide issue. The 500 series only supported 14 and 28 wide issue. Unfortunately, the early VLIW machines were commercial failures. Cydrome closed in 1988 and Multiflow in 1990.

Since then, VLIW processors have seen a revival and some degree of commercial success. Some of the notable VLIW processors of recent years are the IA-64 or Itanium from Intel, the Crusoe processor from Transmeta, the Trimedia media-processor from Philips and the TMS320C62x series of DSPs from Texas Instruments. Some important research machines designed during this time include the Playdoh from HP labs, Tinker from North Carolina State University, and the Imagine stream and image processor currently developed at Stanford University.

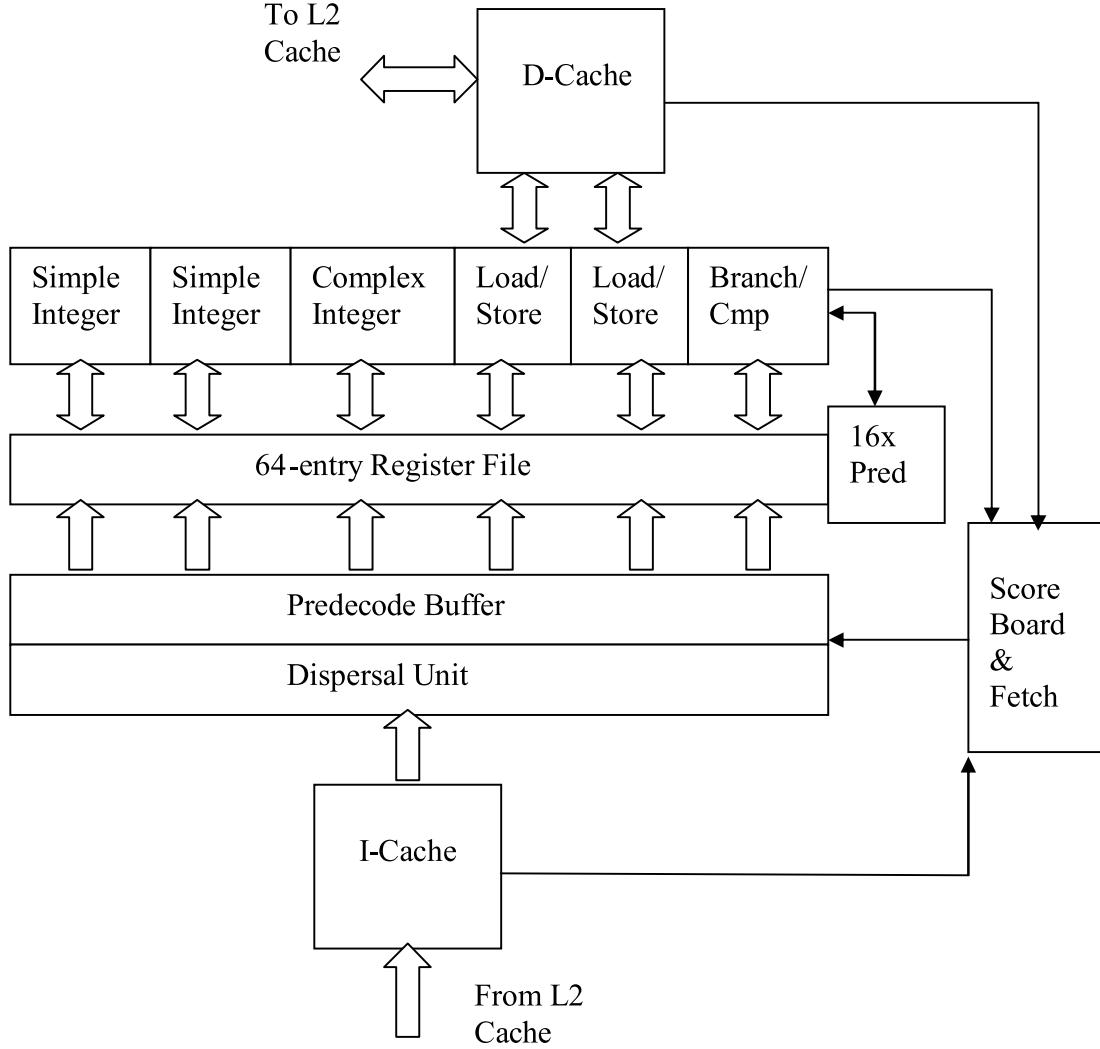


Figure 1: **Defoe Architecture**

4 Defoe: An Example VLIW Architecture

We now describe Defoe, an example processor used in this section to give the reader a feel for VLIW architecture and programming. Though it does not exist in reality, its features are derived from those of several existing VLIW processors. Later sections that describe the IA-64 and the Crusoe, will contrast those architectures with Defoe. Figure 1 shows the architecture of the Defoe processor.

4.1 Function units

Defoe is a 64-bit architecture with the following function units.

- Two load/store units.

Stop bit (1 bit)	Predicate (4 bits)	Opcode (9 bits)	Rdest (6)	Rsrc1 (6)	Rsrc2 (6)
---------------------	------------------------	--------------------	----------------	----------------	----------------

Figure 2: Instruction Encoding

- Two simple ALUs that perform add, subtract, shift and logical operations on 64-bit numbers and packed 32, 16 and 8-bit numbers. In addition, these units also support multiplication of packed 16 and 8-bit numbers.
- One complex ALU that can perform multiply and divide on 64-bit integers and packed 32, 16 and 8-bit integers.
- One branch unit that performs branch, call and comparison operations.

There is no support for floating point. Figure 1 shows a simplified diagram of the Defoe architecture.

4.2 Registers and Predication

Defoe has a set of 64 programmer visible general purpose registers which are 64 bits wide. As in the MIPS architecture, register R0 always contains zero. Predicate registers are special 1-bit registers that specify a true or false value. There are 16 programmer visible predicate registers in the Defoe named PR0 to PR15. All operations in Defoe are predicated, i.e., each instruction contains a predicate register field (PR) that contains a predicate register number. At run-time, if control reaches the instruction, the instruction is always executed. However, at execution time, if the predicate is false, the results are not written to the register file and side effects such as TLB miss exceptions are suppressed. In practice, for reasons of efficiency, this may be implemented by computing a result, but writing back the old value of the target register. Predicate register 0 always contains the value 1 and cannot be altered. Specifying PR0 as the predicate performs unconditional operations. Comparison operations use a predicate register as their target register.

4.3 Instruction Encoding

Defoe is a 64-bit compressed VLIW architecture. In an uncompressed VLIW system, MultiOps have a fixed length. When suitable operations are not available to fill the issue slots within a MultiOp, NOPs are inserted into those slots. A compressed VLIW architecture uses variable length MultiOps to get rid of those NOPs and achieve high code density. In the Defoe, individual operations are encoded as 32-bit words. A special stop bit in the 32-bit word indicates the end of a MultiOp. Common arithmetic operations have an immediate mode, where a sign or zero extended 8-bit constant may be used as an operand. For larger constants of 16, 32 or 64 bits, a special NOP code may be written into opcode field of the next operation and the low order bits may be used to store the constant. In that case, the predecoder concatenates the bits from 2 or more different words to assemble a constant. Figure 2 depicts the instruction format.

4.4 Instruction Dispersal and Issue

A traditional VLIW with fixed width MultiOps has no need to disperse operations. However, when using a compressed format like that of the Defoe, there is a need to expand the operations, and insert NOPs for function units to which no operation is to be issued. To make the dispersal task easy we make the following assumptions:

- A few bits in the opcode specify the type of function unit (i.e. load/store, simple arithmetic, complex arithmetic or branch) the operation needs.
- The compiler ensures that the instructions that comprise a MultiOp are sorted in the same order as the function units in the processor. This reduces the circuit complexity of the instruction dispersal stage. For example, if a MultiOp consists of a load, 32-bit divide and a branch, then the ordering (load, multiply, branch) is legal, but the ordering (load, branch, multiply) is not legal.
- The compiler ensures that all the operations in the same MultiOp are independent.
- The compiler ensures that the function units are not over subscribed. For example, it is legal to have two loads in a MultiOp, but it is not legal to have three loads.
- It is illegal to not have a stop bit in a sequence of more than 6 instructions.
- Basic blocks are aligned at 32-byte boundaries.

Apart from reducing wastage of memory, another reason to prefer a compressed format VLIW over an uncompressed one is that the former provides better I-Cache utilization. To improve performance, we use a predecode buffer that can hold up to 8 uncompressed MultiOps. The dispersal network can use a wide interface (say 512 bits) to the I-cache to uncompress up to 2 MultiOps every cycle and save them in the predecode buffer. Small loops of up to 8 MultiOps (maximum 48 operations) will experience repeated hits in the predecode buffer. It may also help lower the power consumption of a low-power VLIW processor. Defoe supports in-order issue and out of order completion. Further, all the operations in a MultiOp are issued simultaneously. If even one operation cannot be issued, issue of the whole MultiOp stalls.

4.5 Branch Prediction

Following the VLIW philosophy of enabling the software to communicate its needs to the hardware, branch instructions in Defoe can advise the processor about their expected behavior. A two bit hint associated with every branch may be interpreted as shown in Table 1.

Implementations of the Defoe architecture may provide branch prediction hardware, but a branch predictor is not required in a minimal implementation. If branch prediction hardware is provided, static branches need not be entered in the branch history table, thereby freeing up resources for dynamically predicted branches.

Opcode Modifier	Meaning
Stk	Static prediction. Branch is usually taken.
Sntk	Static prediction. Branch is usually not taken.
Dtk	Dynamic prediction. Assume branch is taken if no history is available.
Dntk	Dynamic prediction. Assume branch is not taken if no history is available.

Table 1: Branch Prediction Hints

4.6 Score board

To accommodate branch prediction and the variable latency of memory accesses because of cache hits and misses, some amount of score boarding is required. Though we will not describe the details of the scoreboard here, it should be emphasized that the scoreboard and control logic for a VLIW processor like the Defoe is much simpler than that of a modern super scalar processor because of the lack of out of order execution and speculation.

4.7 Assembly Language Syntax

The examples that follow use the following syntax for assembly language instructions.

(predicate_reg) opcode.modifier Rdest = Rsource1, Rsource2

If the predicate register is omitted, PR0 will be assumed. In addition, a semicolon following an instruction indicates that the stop bit is set for that operation, i.e., that operation is the last one in its MultiOp. The prefix “!” for a predicate implies that the opcode actually depends on the logical not of the value of the predicate register.

4.8 Example 1

This example demonstrates the execution model of the Defoe by computing the following set of expressions.

```
a = x + y - z
b = x + y - 2 * z
c = x + y - 3 * z
```

Register assignments: r1 = x, r2 = y, r3 = z, r32 = a, r33 = b, r34 = c

Line #	Code	Comments
1.	add r4 = r1, r2 // r4 = x + y	
2.	shl r5 = r3, 1 // r5 = z << 1, i.e. z * 2	
3.	mul r6 = r3, 3 ; // r6 = z * 3. Stop bit.	
4.	sub r32 = r4, r5 // r5 = a = gets x + y - z	
5.	sub r33 = r4, r5 ; // r33 = b = x + y - 2 * z.	
	// Stop bit.	

```

6. sub r34 = r4, r6 ; // r34 = c = x + y - 3 * z.
// Stop bit.

```

The first three lines are followed by a stop bit to indicate that those three operations constitute a MultiOp and that they should be executed in parallel. Unlike a super scalar processor where independent operations are detected by the processor, the programmer/compiler has indicated to the processor by means of the stop bit that these 3 operations are independent. The multiply operation will typically have a higher latency than the other instructions. In that case we have two different ways of scheduling this code. Since Defoe already uses score boarding to deal with variable load latencies, it is only natural for the scoreboard to stall issue till the multiply operation is done. In a traditional VLIW processor, the compiler will insert additional NOPs after the first MultiOp. Lines 4-6 show how structural hazards are handled in a VLIW system. The compiler is aware that Defoe has only two simple integer ALUs. Even though instruction 6 is independent of instructions 4 and 5, because of the unavailability of a suitable function unit, instruction 6 is issued as a separate MultiOp, one cycle after its two predecessors. In a super scalar processor, this decision will be handled at run-time by the hardware.

4.9 Example 2

This example contrasts the execution of an algorithm on Defoe and a super scalar processor (Intel Pentium). The C language function `absdiff` computes the sum of absolute difference of two arrays A and B which contain 256 elements each.

```

int absdiff(int *A, int *B)
{
    int sum, diff, i;
    sum = 0;
    for(i = 0; i<256; i++)
    {
        diff = A[i] - B[i];
        if(A[i] >= B[i])
            sum = sum + diff;
        else
            sum = sum - diff;
    }
    return sum;
}

```

A hand assembled version of `absdiff` in Defoe assembly language is shown below. For clarity, it has been left unoptimized. An optimizing compiler will unroll this loop and software pipeline it.

Register assignment: On entry, r1 = a, r2 = b. On exit, sum is in r4.

<i>Line #</i>	<i>Code</i>	<i>Comment</i>
1.	add r3 = r1, 2040 // r3 = End of array A	
2.	add r4 = r0, r0 ; // sum = r4 = 0	
	.L1:	

```

3. ld r5 = [r1]          // load A[i]
4. ld r6 = [r2]          // load B[i]
5. add r1 = r1, 8        // Increment A address
6. add r2 = r2, 8        // Increment B address
7. cmp.neq pr1 = r1, r3 ; // pr1 = (i != 255)
8. sub r7, r5, r6 // diff = A[i] - B[i]
9. cmp.gte pr2 = r5, r6 ; // pr2 = (A[i] >= B[i])
10. (pr2) add r4 = r4, r7 // if A[i] >= B[i]
    // sum = sum + diff
11. (!pr2) sub r4 = r4, r7 // else sum = sum - diff
12. (pr1) br.sptk .L1 ;

```

The corresponding code for an Intel processor is shown below. This is a snippet of actual code generated by the GCC compiler.

Stack assignment: On entry, 12(%ebp) = B, 8(%ebp) = A. On exit, sum is in the eax register.

<i>Line #</i>	<i>Code</i>	<i>Comment</i>
1.	movl 12(%ebp), %edi // edi = B	
2.	xorl %esi, %esi // esi sum = 0	
3.	xorl %ebx, %ebx // ebx = 0	
	.p2align 2	
.L6:		
4.	movl 8(%ebp), %eax // eax = A	
5.	movl (%eax,%ebx,4), %edx // edx = A[i]	
6.	movl %edx, %ecx // ecx = A[i]	
7.	movl (%edi,%ebx,4), %eax // eax = B[i]	
8.	subl %eax, %ecx // ecx = diff = A[i] - B[i]	
9.	cmpl %eax, %edx // A[i] < B[i]	
10.	jl .L7 // goto .L7 is A[i] < B[i]	
11.	addl %ecx, %esi // sum = sum + diff	
12.	jmp .L5	
	.p2align 2	
.L7:		
13.	subl %ecx, %esi // sum = sum - diff	
.L5:		
14.	incl %ebx // i++	
15.	cmpl \$255, %ebx // i <= 255 ?	
16.	jle .L6	
17.	popl %ebx	
18.	movl %esi, %eax	

The level of parallelism available in the Defoe listing lines 3-7 (five issue) can be achieved on a super scalar processor only if the processor can successfully isolate the five independent operations fast enough to issue them all during the same cycle. Dependency checking in h/w is extremely complex and adds to the delay of super scalar processors. The x86 being a register deficient CISC

architecture also incurs additional penalties because of register renaming and CISC to internal RISC format translation.

It is worth noticing that the Defoe listing contains only one branch (on line 12) whereas the x86 listing contains 3 branches. On a VLIW processor, we can often use predicated instructions to eliminate branches. In both listings, line 9 corresponds to the comparison of $A[i]$ and $B[i]$. The Pentium version does a conditional jump based on the result of the comparison. On the other hand, the VLIW uses the result of the comparison to set a predicate. The predicate is then used to selectively write back the result of either the add or the subtract operation and the result of the other operation is discarded. This technique of converting a control dependence into a data dependence is called ‘if conversion’. The benefits go beyond the single cycle saved by not doing a jump as in the case of the super scalar processor. The jumps on line 10 and 12 in the second listing depend on the condition code which in turn depends on the data. Such data dependent branches are difficult to predict. Assuming that $A[i] < B[i]$ and $A[i] \geq B[i]$ are equally likely, the super scalar processor is likely to experience a branch misprediction and the resulting branch penalty half of the time.

Going by the VLIW philosophy of conveying performance critical information from the compiler to the hardware, the final branch on line 12 uses the opcode modifier “sptk” to inform the processor that the branch is statically predicted to be taken. For that particular loop, a VLIW processor can therefore predict the loop accurately 255 times out of 256 loop iterations without any hardware branch predictor. Even when a hardware branch predictor is available, the instruction advises the processor not to waste a branch history table entry on that branch since its behavior is already known at compile time.

5 The Intel Itanium Processor

The Itanium-1 processor is Intel’s first implementation of the IA-64 ISA. IA-64 is an ISA for the Explicitly Parallel Instruction Computing (EPIC) style of VLIW developed jointly by Intel and HP. It is a 64-bit, 6 issue VLIW processor with four integer units, four multimedia units, two load/store units, two extended precision floating point units and two single precision floating point units. This processor running at 800 MHz on a 0.18 micron process has a 10 stage pipeline.

Unlike the Defoe, the IA-64 architecture uses a fixed-width bundled instruction format. Each MultiOp consists of one or more 128 bit bundles. Each 128 bit bundle consists of three operations and a template. Unlike the Defoe where the opcode in each operation specifies a type field, the template encodes commonly used combinations of operation types. Since the template field is only five bits wide, bundles do not support all possible combinations of instruction types. Much like the Defoe’s stop bit, in the IA-64, some template codes specify where in the bundle a MultiOp ends. In IA-64 terminology, MultiOps are called instruction groups. Like Defoe, the IA-64 uses a decoupling buffer to improve its issue rate. Though the IA-64 registers are nominally 64 bits wide, there is a hidden 65th bit called NaT (Not a Thing). This is used to support speculation. There are 128 general purpose registers and another set of 128, 82-bit wide floating point registers. Like the Defoe, all operations on the IA-64 are predicated. However, the IA-64 has 64 predicate registers.

The IA-64 register mechanism is more complex than the Defoe’s because it implements support for software pipelining using a method similar to the overlapped loop execution support pioneered by Bob Rau and implemented in the Cydra 5. On the IA-64, general purpose registers GPR0 to GPR31 are fixed. Registers 32-127 can be renamed under program control to support a register

stack or to do modulo scheduling for loops. When used to support software pipelining, this feature is called register rotation. Predicate registers 0 to 15 are fixed while predicate registers 16 to 63 can be made to rotate in unison with the general purpose registers. The floating point registers also support rotation.

Modulo scheduling is a software pipelining technique that can support overlapped execution of loop bodies while reducing tail code. In a pipelined function unit, each stage can hold a computation and successive items of data may be applied to the function unit before previous data is completely processed. To take advantage of pipelined operation, in a modulo scheduled loop, the loop body is unrolled and split into several stages. The compiler can schedule multiple iterations of a loop in a pipelined manner as long as data outputs of one stage flow into the inputs of the next stage in the software pipeline. Traditionally, this required unrolling the loop and renaming the registers used in successive iterations. IA-64 reduces the overhead of such a loop and avoids the need for register renaming by rotating registers forward, i.e., the rotating register base is incremented in the direction of increasing register index. After rotating by n registers, the value that was in register $X+n$ can be accessed from register X . When used in conjunction with predication, this allows a natural expression of software pipelines similar to their hardware counterparts.

The IA-64 supports software directed control and data speculation. To do control speculation, the compiler moves a load before its controlling branch. The load is then flagged as a speculative load. The processor does not signal exceptions on a speculative load. If the controlling branch is taken, the compiler uses a special opcode named `check.s` to determine if an exception occurred. If an exception occurred, the check operation transfers control to exception handling code.

To support data speculation, the processor supports a special kind of load called an advance load. If the compiler cannot disambiguate between the addresses of a store and a later load, it can issue an advance load ahead of the store. The processor uses a special hardware structure called the ALAT to keep track of whether a later store wrote to the same location as the advance load. In the original location where the load might naturally have been placed, the compiler inserts a special check operation to see if a store invalidated the result of the advance load. If the advance load was invalidated, the check operation transfers control to recovery code.

As in the case of Defoe, the IA-64 too supports both static and dynamic hints for branches. It also makes use of hardware branch prediction. There are also hints in load and store instructions that inform the processor about the cache behavior of a particular memory operation.

The IA-64 also includes SIMD instructions suitable for media processing. Special multimedia instructions similar to the MMX and SSE extensions for 80x86 processors treat the contents of a general purpose register as two 32-bit, four 16-bit or eight 8-bit operands and operate on them in parallel.

To improve performance, the IA-64 architecture includes several features that are not found in a traditional VLIW architecture. The Intel Itanium processor is probably the most complex VLIW ever designed. It is a matter of debate whether some of the control complexity of the IA-64 is justifiable in a VLIW architecture and whether the enhancements deliver commensurate performance improvements. Next, we will look at a simpler VLIW processor that has been designed with a totally different goal — that of reducing power consumption.

6 The Transmeta Crusoe Processor

The Crusoe processor from Transmeta corporation represents a very interesting point in the development of VLIW processors. Traditionally, VLIW processors were designed with the goal of maximizing ILP and performance. The designers of the Crusoe on the other hand needed to build a processor with moderate performance compared to the CPU of a desktop computer, but with the additional restriction that the Crusoe should consume very little power since it was intended for mobile applications. Another design goal was that it should be able to efficiently emulate the ISA of other processors, particularly the 80x86 and the Java virtual machine.

The designers left out features like out of order issue and dynamic scheduling that require significant power consumption. They set out to replace such complex mechanisms of gaining ILP with simpler and more power efficient alternatives. The end result was a simple VLIW architecture. Long instructions on the Crusoe are either 64 or 128 bits. A 128-bit instruction word called a molecule in Transmeta parlance encodes 4 operations called atoms. The molecule format directly determines how operations get routed to function units. The Crusoe has two integer units, a floating point unit, a load/store unit and a branch unit. Like the Defoe, the Crusoe has 64 general purpose registers and supports strictly in order issue. Unlike the Defoe which uses predication, the Crusoe uses condition flags which are identical to those of the x86 for ease of emulation.

Binary x86 programs, firmware and operating systems are emulated with the help of a run time binary translator called code morphing software. This makes the classical VLIW software compatibility problem a non-issue. Only the native code morphing software needs to be changed when the Crusoe architecture or ISA changes. As a power and performance optimization, the hardware and software together maintain a cache of translated code. The translations are instrumented to collect execution frequencies and branch history and this information is fed back to the code morphing software to guide its optimizations.

To correctly model the precise exception semantics of the x86 processor, the part of the register file that holds x86 register state is duplicated. The duplicate is called a shadow copy. Normal operations only affect the original registers. At the end of a translated section of code, a special commit operation is used to copy the working register values to the shadow registers. If an exception happens while executing a translated unit, the run time software uses the shadow copy to recreate the precise exception state. Store operations are implemented in a similar manner using a store buffer. As in the case of IA-64, the Crusoe provides alias detection hardware and data speculation primitives.

7 Scheduling Algorithms for VLIW

The difficulty of programming VLIW processors by hand should be evident even from the simple Defoe programming examples. One reason programming VLIWs is more difficult than writing code for a super scalar processor is that the program for a super scalar processor is inherently sequential and it is left to the hardware to extract parallelism from the sequential program. On the other hand, when generating code for a VLIW processor, the assembly language programmer or the compiler is faced with the task of extracting parallelism from a sequential algorithm and scheduling independent operations concurrently. For this reason, instruction scheduling algorithms are critical to the performance of a VLIW processor. We next describe three important scheduling algorithms

starting with the classic trace scheduling algorithm.

7.1 Trace Scheduling

Many compilers for first-generation ILP processors used a three phase method to generate code. The phases were:

- Generate a sequential program. Analyze each basic block in the sequential program for independent operations.
- Schedule independent operations within the same block in parallel if sufficient hardware resources are available.
- Move operations between blocks when possible.

This three phase approach fails to exploit much of the ILP available in the program for two reasons.

- Often times, operations in a basic block are dependent on each other. Therefore sufficient ILP may not be available within a basic block.
- Arbitrary choices made while scheduling basic blocks make it difficult to move operations between blocks.

Trace scheduling is a profile driven method developed by Joseph Fisher to circumvent this problem. In trace scheduling, a set of commonly executed sequence of blocks is gathered together into a trace and the whole trace is scheduled together.

The trace scheduling algorithm works as follows.

1. Generate a possibly unoptimized version of the program, run it on sample input and collect statistics. Estimate the probability of each conditional branch.
2. From the basic block level data precedence graph of the program (also commonly called DAG for Directed Acyclic Graph), select a loop free linear sequence of basic blocks which have a high probability of execution. Such a sequence is called a trace. The compiler may use other optimizations like loop unrolling or procedure inlining to generate DAGs from which suitable traces can be selected.
3. Consider the trace as if it were a basic block. Build a DAG for it considering branches like all other operations. If an operation controlled by a conditional jump could over write a value that is live on the off-trace edge, add an edge that makes the operation dependent on the branch so that the operation cannot be moved ahead of the branch. Also, add edges to preserve the relative order of conditional branches.
4. Schedule the resulting DAG as if it were a basic block doing register allocation and function unit selection as each operation is scheduled.
5. Generate compensation code for mistakes made by considering the trace as a basic block. In particular:
 - a. If an operation that used to precede a conditional branch in the sequential code is moved after that branch, then add a copy of that operation preceding the off-trace target of the conditional jump.

- b. If an operation that succeeded a point of entry into the trace from outside the trace is moved ahead of that point of entry, place a copy of that operation outside the trace, on the path that leads to that point of entry.
 - c. Ensure that rejoins that used to enter the trace enter the new trace only at a point after which no operation is found in the new trace that were not below the rejoin point in the old trace.
6. Link the new trace back into the old DAG.
 7. After scheduling the very first trace, new operations would have been added to the original DAG. Pick a different frequent trace and schedule it. Repeat till the DAG has been covered using disjoint traces and no unscheduled operations remain.

7.2 Trace Scheduling - 2

Trace scheduling - 2 goes beyond the original trace scheduling in that it allows nonlinear code motion, i.e. it allows operations from both sides of a conditional branch to be moved above the branch. Trace scheduling usually misses code motions that are speculative or moves operations from one trace to another. Trace scheduling - 2 on the other hand uses an expected value function called speculative yield to consider the cost of speculative execution and decide whether or not to move operations from one block to another. Unlike trace scheduling which operates on a linear sequence of blocks, the newer algorithm works by picking clusters of operations where each cluster is a maximal set of operations that are connected without back edges in the flow graph of the program. The actual details of the algorithm are beyond the scope of this article.

7.3 Super Block Scheduling

Super block scheduling is a region scheduling algorithm developed in conjunction with the Impact compiler at the University of Illinois. Like trace scheduling, super block scheduling is based on the premise that extracting ILP from sequential programs requires code motion across multiple basic blocks. Unlike trace scheduling, super block scheduling is driven by static branch analysis, not profile data. A super block is a set of basic blocks in which control may enter only at the top, but may exit at more than one point. Super blocks are identified by first identifying traces and then eliminating side entries into a trace by a process called tail duplication. Tail duplication works by creating a separate off-trace copy of the basic blocks in between a side entrance and the trace exit and redirecting the edge corresponding to the side entry to the copy. Traces are identified using static branch analysis based on loop detection, heuristic hazard avoidance and heuristics for path selection. Loop detection identifies loops and marks loop back edges as taken and loop exits as not taken. Hazard avoidance uses a set of heuristics to detect situations like ambiguous stores and procedure calls that could cause a compiler to use conservative optimization strategies and then predicts the branches so as to avoid having to optimize hazards. Path selection heuristics use the opcode of a branch, its operands and the contents of its successor blocks to predict its direction if no other method can be used to predict the outcome of the branch. These are based on common programming patterns like the fact that pointers are unlikely to be NULL, floating point comparisons are unlikely to be equal etc. Once branch information is available, traces are grown and super blocks created by tail duplication followed by scheduling of the super block. Studies have shown that static analysis based super block scheduling can achieve results that are comparable to profile based methods.

7.4 The Future of VLIW Processors

VLIW processors have enjoyed moderate commercial success in recent times as exemplified by the Philips Trimedia, TI TMS320C62x DSPs, Intel Itanium and to a lesser extend the Transmeta Crusoe. However, the role of VLIW processors has changed since the days of Cydrome and Multiflow. Even though early VLIW processors were developed to be scientific super computers, newer processors have been used mainly for stream, image and digital signal processing, multimedia codec hardware, low power mobile computers etc. VLIW compiler technology has made major advances during the last decade. However, most of the compiler techniques developed for VLIW are equally applicable to super scalar processors. Stream and media processing applications are typically very regular with predictable branch behavior and large amounts of ILP. They lend themselves easily to VLIW style execution. The ever increasing demand for multimedia applications will continue to fuel development of VLIW technology. However, in the short term, super scalar processors will probably dominate in the role of general purpose processors. Increasing wire delays in deep sub micron processes will ultimately force super scalar processors to use simpler and more scalable control structures and seek more help from software. It is reasonable to assume that in the long run, much of the VLIW technology and design philosophy will make its way into main stream processors.

7.5 References

1. Joseph A. Fisher, *Global code generation for instruction-level parallelism: Trace Scheduling-2*. Tech. Rep. HPL-93-43, Hewlett-Packard Laboratories, June 1993.
2. Joseph A. Fischer, *Very Long Instruction Word Architectures and the ELI-512*, Proceedings of the 10'th Symposium on Computer Architectures, pp. 140-150, IEEE, June, 1983.
3. Joseph A. Fisher, *Very Long Instruction Word Architectures and the ELI-512. 25 Years ISCA: Retrospectives and Reprints 1998*: 263-273
4. M. Schlansker, B. R. Rau, S. Mahlke, V. Kathail, R. Johnson, S. Anik, and S. G. Abraham, *Achieving high levels of instruction-level parallelism with reduced hardware complexity*. Technical report, Technical Report HPL-96-120, Hewlett Packard Laboratories, Feb. 1997.
5. M. Schlansker, B. R. Rau. *Epic: An Architecture for Instruction Level Parallel Processors*. Technical report, Technical Report HPL-1999-111, Hewlett Packard Laboratories, Feb. 2000.
6. Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek, Lopez-Lagunas, Abelardo, Peter R. Mattson, and John D. Owens. *A bandwidth-efficient architecture for media processing*. In Proc. 31st Annual International Symposium on Microarchitecture, Dallas, TX, November 1998.
7. Intel Corporation. *Itanium Processor Microarchitecture Reference for Software Optimization*. Intel Corporation, March 2000
8. Intel Corporation. *Intel IA-64 Architecture Software Developer's Manula, Volume 3: Instruction Set Reference*. Intel Corporation, January 2000
9. Intel Corporation. *IA-64 Application Developer's Architecture Guide*. Intel Corporation, May 1999
10. P.G. Lowney, S.M. Freudenberger, T.J. Karzes, W.D. Lichtenstein, R.P. Nix, J.S. O'Donnell, and J.C. Ruttenberg. *The Multiflow trace scheduling compiler*. Journal of Supercomputing, 7, 1993.

11. R. E. Hank, S. A. Mahlke, J. C. Gyllenhaal, R. Bringmann, and W. W. Hwu, *Superblock formation using static program analysis*, in Proc. 26th Ann. Int'l. Symp. on Microarchitecture, (Austin, TX), pp. 247–255, Dec. 1993.
12. S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, *Effective compiler support for predicated execution using the hyperblock*, in Proceedings of the 25th International Symposium on Microarchitecture, pp. 45–54, December 1992.
13. James C. Dehnert, Peter Y. T. Hsu, Joseph P. Bratt, *Overlapped Loop Support in the Cydra 5* in Proc. ASPLOS 89, pp. 26-38.
14. Alexander Klaiber, *The Technology Behind Crusoe Processors*. Transmeta Corp, 2000.