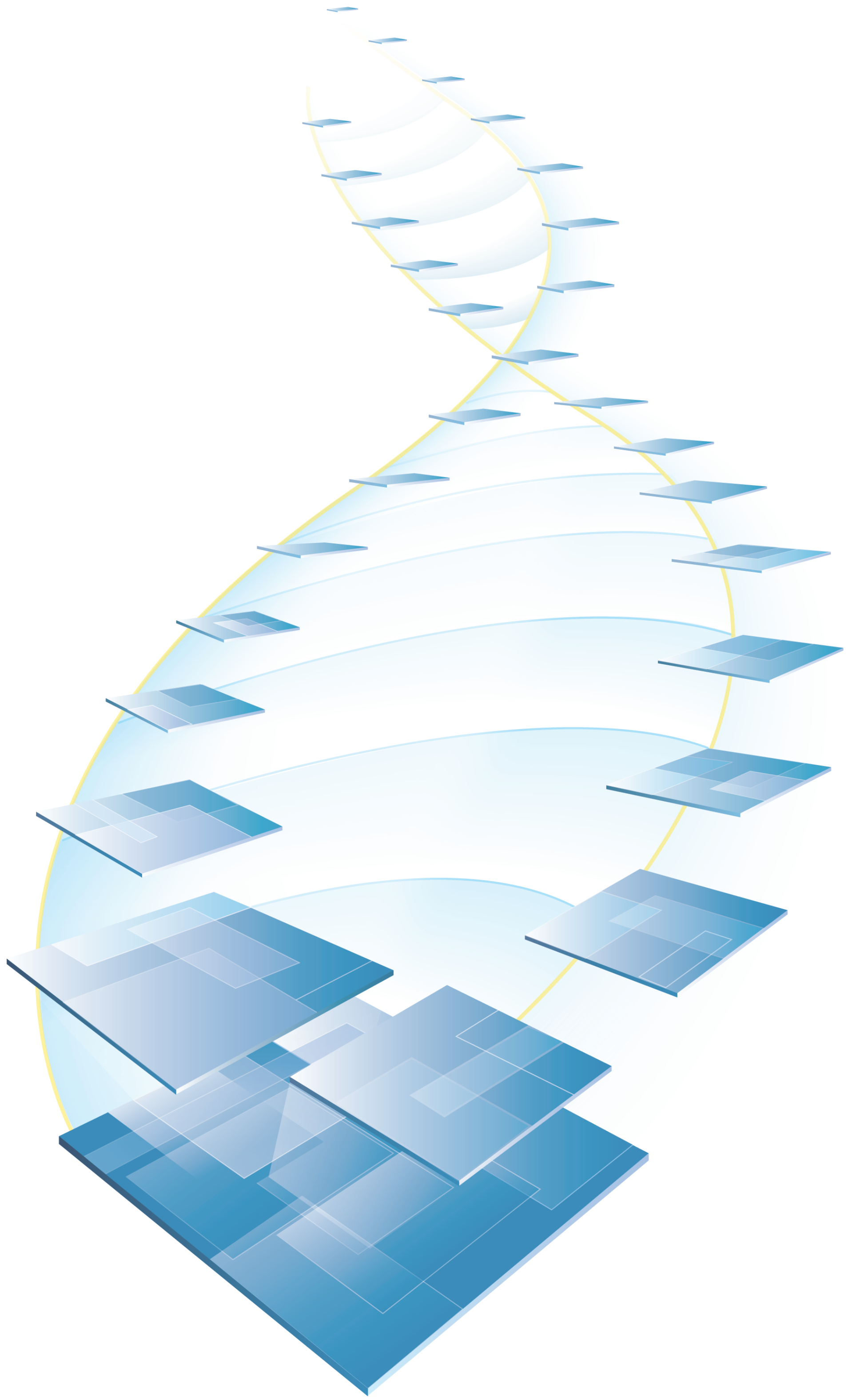
Intel® Technology Journal

DECEMBER 2009

Addressing the Challenges of Tera-scale Computing



Intel Technology Journal

Publisher
Richard Bowles

Managing Editor
David King

Content Architect
Jim Held

Program Manager
Stuart Douglas

Technical Editor
Marian Lacey

Technical Illustrators
InfoPros

Technical and Strategic Reviewers

Terry A. Smith
Jim Hurley
Ali-Reza Adl-Tabatabai
Jesse Fang
Sridhar Iyengar
Joe Schutz
Shekhar Borkar
Greg Taylor

Intel Technology Journal

Copyright © 2009 Intel Corporation. All rights reserved.
ISBN 978-1-934053-23-2, ISSN 1535-864X

Intel Technology Journal
Volume 13, Issue 4

This book is printed on acid-free paper. ∞

Publisher: Richard Bowles
Managing Editor: David King

Library of Congress Cataloging in Publication Data:

Printed in United States of America

10 9 8 7 6 5 4 3 2 1

First printing: December, 2009

INTEL® TECHNOLOGY JOURNAL
ADDRESSING THE CHALLENGES OF TERA-SCALE COMPUTING

Articles

FOREWORD

Jim Held PhD.
Intel Fellow
Director, Tera-scale Computing Research
Intel Labs
Intel Corporation

The Intel® Tera-scale Computing Research Program is Intel's overarching effort to shape the future of Intel processors and platforms, in order to accelerate the shift from frequency to parallelism for performance improvement. Intel researchers worldwide are already working on R&D projects to address the hardware and software challenges of building and programming systems with teraFLOPS of parallel performance that can process tera-bytes of data. This level of performance will enable exciting new and emerging applications, but will also require addressing challenges in everything from program architecture to circuit technologies. This issue of the Intel Technology Journal includes results from a range of research that walks down the 'stack' from application design to circuits.

"Systems with teraFLOPS of parallel performance that can process tera-bytes of data."

Emerging visual-computing applications require tera-scale performance in order to simulate worlds based on complex physical models. They use rich user interfaces with video recognition and 3D graphics synthesis, and they are highly parallel. How can we build them? Architecting designs for such applications that fully exploit their inherent parallelism is a major software engineering challenge. As with most kinds of architecture, new programs will be based on a combination of preexisting patterns and an exploitation of application frameworks that support them. Tim Mattson and Kurt Koetzer describe their work to find the parallel patterns that are needed for concurrent software in their article entitled "A Design Pattern Language for Engineering (Parallel) Software."

The non-deterministic nature of concurrent execution has made debugging one of the toughest parts of delivering a parallel program. Gilles Pokam and his colleagues, in their article "Hardware/Software Approaches for Deterministic Multi-processor Replay of Concurrent Programs" describe their work on hardware and software to support debugging by recording and replaying execution in order to allow analysis and discovery of the subtle timing errors that come with the many possible executions of parallelism.

"Visual-computing applications require tera-scale performance in order to simulate worlds based on complex physical models."

Future tera-scale platforms may be heterogeneous with a mixture of types of compute elements. Our August 2007 issue of the Intel Technology Journal included articles that described support for mixed-ISA co-processing. In "Programming Model for Heterogeneous Intel® x86 Platforms" in this issue, Bratin Saha's and his colleagues describe work in IA-ISA to provide support for shared memory with a mixture of cache coherence models.

Mani Azimi and his colleagues' article "Flexible and Adaptive On-Chip Interconnect for Tera-scale Architectures," describes research into on-die network fabric, and they show our evolution from an analysis of the challenges and alternatives to the development of the protocols to exploit the potential of a network on chip. Effective use of a mesh network will require sophisticated support to provide the routing and configuration management for fairness, load balancing, and congestion management.

"Effective use of a mesh network will require sophisticated support to provide the routing and configuration management for fairness, load balancing, and congestion management."

Perhaps the largest platform hardware challenge for tera-scale computing is the longstanding one of access to memory to match the tremendous compute density of many cores on a die. Moreover, an effective solution must also meet the declining cost and power consumption targets of the mainstream market segments. Dave Dunning and his colleagues, in the article "Tera-scale Memory Challenges and Solutions" outline the problems and our research agenda in this critical area.

The continuing challenge for the core of tera-scale platforms is how to continue to increase energy efficiency. As process technology advances continue to give us more transistors, we can add more cores, but unless we improve their efficiency, we won't be able to use them. Ram Krishnamurthy's team continues to make progress in improving the energy efficiency of computations with designs for ALUs that exploit near-threshold voltage circuits and extremely fine-grained power management. Their work is described in an article "Ultra-low Voltage Technologies for Energy-efficient Special-purpose Hardware Accelerators."

"For some research questions there is no substitute for a silicon implementation."

Finally, for some research questions there is no substitute for a silicon implementation: therefore, we built the Tera-scale Research Processor to explore a tile-based design methodology as well as to understand the performance and power efficiency that is possible with intensive floating-point engines and an on-die network. In our final article "Lessons Learned from the 80-core Tera-scale Research Processor," Saurabh Dighe and his colleagues review these results and discuss what conclusions we draw from them. They summarize what we learned from many experiments with this chip. We recently announced our second-generation many-core research prototype, the Single-chip Cloud Computer, which builds on this work.

I hope you find these articles informative, and the future they are part of creating, as exciting as we do at Intel Labs. We look forward to continuing work with academia and the industry to meet the challenges of mainstream parallel computing.

A DESIGN PATTERN LANGUAGE FOR ENGINEERING (PARALLEL) SOFTWARE

Contributors

Kurt Keutzer
UC Berkeley

Tim Mattson
Intel Corporation

Index Words

Design Pattern Language
Software Architecture
Parallel Algorithm Design
Application Frameworks

Abstract

The key to writing high-quality parallel software is to develop a robust software design. This applies not only to the overall architecture of the program, but also to the lower layers in the software system where the concurrency and how it is expressed in the final program is defined. Developing technology to systematically describe such designs and reuse them between software projects is the fundamental problem facing the development of software for tera-scale processors. The development of this technology is far more important than programming models and their supporting environments, since with a good design in hand, most any programming system can be used to actually generate the program's source code.

In this article, we develop our thesis about the central role played by the software architecture. We show how design patterns provide a technology to define the reusable design elements in software engineering. This leads us to the ongoing project centered at UC Berkeley's Parallel Computing Laboratory (Par Lab) to pull the essential set of design patterns for parallel software design into a Design Pattern Language. After describing our pattern language, we present a case study from the field of machine learning as a concrete example of how patterns are used in practice.

The Software Engineering Crisis

The trend has been well established [1]: parallel processors will dominate most, if not every, niche of computing. Ideally, this transition would be driven by the needs of software. Scalable software would demand scalable hardware and that would drive CPUs to add cores. But software demands are not driving parallelism. The motivation for parallelism comes from the inability of integrated circuit designers to deliver steadily increasing frequency gains without pushing power dissipation to unsustainable levels. Thus, we have a dangerous mismatch: the semiconductor industry is banking its future on parallel microprocessors, while the software industry is still searching for an effective solution to the parallel programming problem.

The parallel programming problem is not new. It has been an active area of research for the last three decades, and we can learn a great deal from what has *not* worked in the past.

- *Automatic parallelism*. Compilers can speculate, prefetch data, and reorder instructions to balance the load among the components of a system. However, they cannot look at a serial algorithm and create a different algorithm better suited for parallel execution.
- *New languages*. Hundreds of new parallel languages and programming environments have been created over the last few decades. Many of them are excellent and provide high-level abstractions that simplify the expression of parallel algorithms. However, these languages have not dramatically grown the pool of parallel programmers. The fact is, in the one community with a long tradition of parallel computing (high-performance computing), the old standards of MPI [2] and OpenMP [3] continue to dominate. There is no reason to believe new languages will be any more successful as we move to more general-purpose programmers; i.e., it is not the quality of our programming models that is inhibiting the adoption of parallel programming.

"It is not the quality of our programming models that is inhibiting the adoption of parallel programming."

The central cause of the parallel programming problem is fundamental to the enterprise of programming itself. In other words, we believe that our challenges in programming parallel processors point to deeper challenges in programming software in general. We believe the only way to solve the programming problem in general is to first understand how to architect software. Thus, we feel that the way to solve the parallel programming problem is to first understand how to architect parallel software. Given a good software design grounded in solid architectural principles, a software engineer can produce high-quality and scalable software. Starting with an ill-suited sense of the architecture for a software system, however, almost always leads to failure. Therefore, it follows that the first step in addressing the parallel programming problem is to focus on software architecture. From that vantage point, we have a hope of choosing the right programming models and building the right software frameworks that will allow the general population of programmers to produce parallel software.

"Given a good software design grounded in solid architectural principles, a software engineer can produce high-quality and scalable software."

In this article, we describe our work on software architecture. We use the device of a pattern language to write our ideas down and put them into a systematic form that can be used by others. After we present our pattern language [4], we present a case study to show how these patterns can be used to understand software architecture.

Software Architecture and Design Patterns

Productive, efficient software follows from good software architecture. Hence, we need to better formalize how software is architected, and in order to do this we need a way to write down architectural ideas in a form that groups of programmers can study, debate, and come to consensus on. This systematic process has at its core the peer review process that has been instrumental in advancing scientific and engineering disciplines.

"Design patterns give names to solutions to recurring problems that experts in a problem-domain gradually learn and take for granted."

The prerequisite to this process is a systematic way to write down the design elements from which an architecture is defined. Fortunately, the software community has already reached consensus on how to write these elements down in the important work *Design Patterns* [5]. Our aim is to arrive at a set of patterns whose scope encompasses the entire enterprise of software development from architectural description to detailed implementation.

Design Patterns

Design patterns give names to solutions to recurring problems that experts in a problem-domain gradually learn and take for granted. It is the possession of this tool-bag of solutions, and the ability to easily apply these solutions, that precisely defines what it means to be an expert in a domain.

For example, consider the *Dense-Linear-Algebra* pattern. Experts in fields that make heavy use of linear algebra have worked out a family of solutions to these problems. These solutions have a common set of design elements that can be captured in a *Dense-Linear-Algebra* design pattern. We summarize the pattern in the sidebar, but it is important to know that in the full text to the pattern [4] there would be sample code, examples, references, invariants, and other information needed to guide a software developer interested in dense linear algebra problems.

Computational Pattern: Dense-Linear-Algebra
Solution: A computation is organized as a sequence of arithmetic expressions acting on dense arrays of data. The operations and data access patterns are well defined mathematically so data can be pre-fetched and CPUs can execute close to their theoretically allowed peak performance. Applications of this pattern typically use standard building blocks defined in terms of the dimensions of the dense arrays with vectors (BLAS level 1), matrix-vector (BLAS level 2), and matrix-matrix (BLAS level 3) operations.

The *Dense-Linear-Algebra* pattern is just one of the many patterns a software architect might use when designing an algorithm. A full design includes high-level patterns that describe how an application is organized, mid-level patterns about specific classes of computations, and low-level patterns describing specific execution strategies. We can take this full range of patterns and organize them into a single integrated pattern language — a web of interlocking patterns that guide a designer from the beginning of a design problem to its successful realization [6, 7].

To represent the domain of software engineering in terms of a single pattern language is a daunting undertaking. Fortunately, based on our studies of successful application software, we believe software architectures can be built up from a manageable number of design patterns. These patterns define the building blocks of all software engineering and are fundamental to the practice of architecting parallel software. Hence, an effort to propose, argue about, and finally agree on what constitutes this set of patterns is the seminal intellectual challenge of our field.

"A full design includes high-level patterns that describe how an application is organized, mid-level patterns about specific classes of computations, and low-level patterns describing specific execution strategies."

Our Pattern Language

Software architecture defines the components that make up a software system, the roles played by those components, and how they interact. Good software architecture makes design choices explicit, and the critical issues addressed by a solution clear. A software architecture is hierarchical rather than monolithic. It lets the designer localize problems and define design elements that can be reused in other architectures.

"It is important to appreciate that many design problems do not lend themselves to a top-down or bottom-up analysis."

The goal of Our Pattern Language (OPL) is to encompass the complete architecture of an application from the structural patterns (also known as architectural styles) that define the overall organization of an application [8, 9] to the basic computational patterns (also known as computational motifs) for each stage of the problem [10, 1], to the low-level details of the parallel algorithm [7]. With such a broad scope, organizing our design patterns into a coherent pattern language was extremely challenging.

Our approach is to use a layered hierarchy of patterns. Each level in the hierarchy addresses a portion of the design problem. While a designer may in some cases work through the layers of our hierarchy in order, it is important to appreciate that many design problems do not lend themselves to a top-down or bottom-up analysis. In many cases, the pathway through our patterns will be to bounce around between layers with the designer working at whichever layer is most productive at a given time (so called, opportunistic refinement). In other words, while we use a fixed layered approach to organize our patterns into OPL, we expect designers will work though the pattern language in many different ways. This flexibility is an essential feature of design pattern languages.

As shown in Figure 1, we organize OPL into five major categories of patterns. Categories 1 and 2 sit at the same level of the hierarchy and cooperate to create one layer of the software architecture.

1. Structural patterns: Structural patterns describe the overall organization of the application and the way the computational elements that make up the application interact. These patterns are closely related to the architectural styles discussed in [8]. Informally, these patterns correspond to the "boxes and arrows" an architect draws to describe the overall organization of an application. An example of a structural pattern is *Pipe-and-Filter*, described in the sidebar.
2. Computational patterns: These patterns describe the classes of computations that make up the application. They are essentially the thirteen motifs made famous in [10] but described more precisely as patterns rather than simply computational families. These patterns can be viewed as defining the "computations occurring in the boxes" defined by the structural patterns. A good example is the *Dense-Linear-Algebra* pattern described in an earlier sidebar. Note that some of these patterns (such as *Graph-Algorithms* or *N-Body-Methods*) define complicated design problems in their own right and serve as entry points into smaller design pattern languages focused on a specific class of computations. This is yet another example of the hierarchical nature of the software design problem.

Structural Pattern: Pipe-and-Filter
Solution: Structure an application as a fixed sequence of filters that take input data from preceding filters, carry out computations on that data, and then pass the output to the next filter. The filters are side-effect free; i.e., the result of their action is only to transform input data into output data. Concurrency emerges as multiple blocks of data move through the Pipe-and-Filter system so that multiple filters are active at one time.

Concurrent Algorithm Strategy Pattern: Data-Parallelism
Solution: An algorithm is organized as operations applied concurrently to the elements of a set of data structures. The concurrency is in the data. This pattern can be generalized by defining an index space. The data structures within a problem are aligned to this index space and concurrency is introduced by applying a stream of operations for each point in the index space.

In OPL, the top two categories, the structural and computational patterns, are placed side by side with connecting arrows. This shows the tight coupling between these patterns and the iterative nature of how a designer works with them. In other words, a designer thinks about his or her problem, chooses a structural pattern, and then considers the computational patterns required to solve the problem. The selection of computational patterns may suggest a different overall structure for the architecture and may force a reconsideration of the appropriate structural patterns. This process, moving between structural and computational patterns, continues until the designer settles on a high-level design for the problem.

Structural and computational patterns are used in both serial and parallel programs. Ideally, the designer working at this level, even for a parallel program, will not need to focus on parallel computing issues. For the remaining layers of the pattern language, parallel programming is a primary concern.

Implementation Strategy Pattern: Loop-Parallel
Solution: An algorithm is implemented as loops (or nested loops) that execute in parallel. The challenge is to transform the loops so that iterations can safely execute concurrently and in any order. Ideally, this leads to a single source code tree that generates a serial program (by using a serial compiler) or a parallel program (by using compilers that understand the parallel loop constructs).

Parallel programming is the art of using concurrency in a problem to make the problem run to completion in less time. We divide the parallel design process into the following three layers.

3. Concurrent algorithm strategies: These patterns define high-level strategies to exploit concurrency in a computation for execution on a parallel computer. They address the different ways concurrency is naturally expressed within a problem by providing well-known techniques to exploit that concurrency. A good example of an algorithm strategy pattern is the *Data-Parallelism* pattern.
4. Implementation strategies: These are the structures that are realized in source code to support (a) how the program itself is organized and (b) common data structures specific to parallel programming. The *Loop-Parallel* pattern is a well-known example of an implementation strategy pattern.
5. Parallel execution patterns: These are the approaches used to support the execution of a parallel algorithm. This includes (a) strategies that advance a program counter and (b) basic building blocks to support the coordination of concurrent tasks. The single instruction multiple data (SIMD) pattern is a good example of a parallel execution pattern.

Patterns in these three lower layers are tightly coupled. For example, software designs using the *Recursive-Splitting* algorithm strategy often utilize a *Fork/Join* implementation strategy pattern which is typically supported at the execution level with the *thread-pool* pattern. These connections between patterns are a key point in the text of the patterns.

Parallel Execution Pattern: SIMD
Solution: An implementation of a strictly data parallel algorithm is mapped onto a platform that executes a single sequence of operations applied uniformly to a collection of data elements. The instructions execute in lockstep by a set of processing elements but on their own streams of data. SIMD programs use specialized data structure, data alignment operations, and collective operations to extend this pattern to a wider range of data parallel problems.

OPL draws from a long history of research on software design. The structural patterns of Category 1 are largely taken from the work of Garlan and Shaw on architectural styles [8, 9]. That these architectural styles could also be viewed as design patterns was quickly recognized by Buschmann [11]. We added two structural patterns that have their roots in parallel computing to Garlan and Shaw's architectural styles: *Map-Reduce*, influenced by [12] and *Iterative-Refinement*, influenced by Valiant's bulk-synchronous-processing pattern [13]. The computation patterns of Category 2 were first presented as "dwarfs" in [10] and their role as computational patterns was only identified later [1]. The identification of these computational patterns in turn owes a debt to Phil Colella's unpublished work on the "Seven Dwarfs of Parallel Computing." The lower three categories within OPL build on earlier and more traditional patterns for parallel algorithms by Mattson, Sanders, and Massingill [7]. This work was somewhat inspired by Gamma's success in using design patterns for object-oriented programming [5]. Of course all work on design patterns has its roots in Alexander's ground-breaking work identifying design patterns in civil architecture [6].

"All work on design patterns has its roots in Alexander's ground-breaking work identifying design patterns in civil architecture."

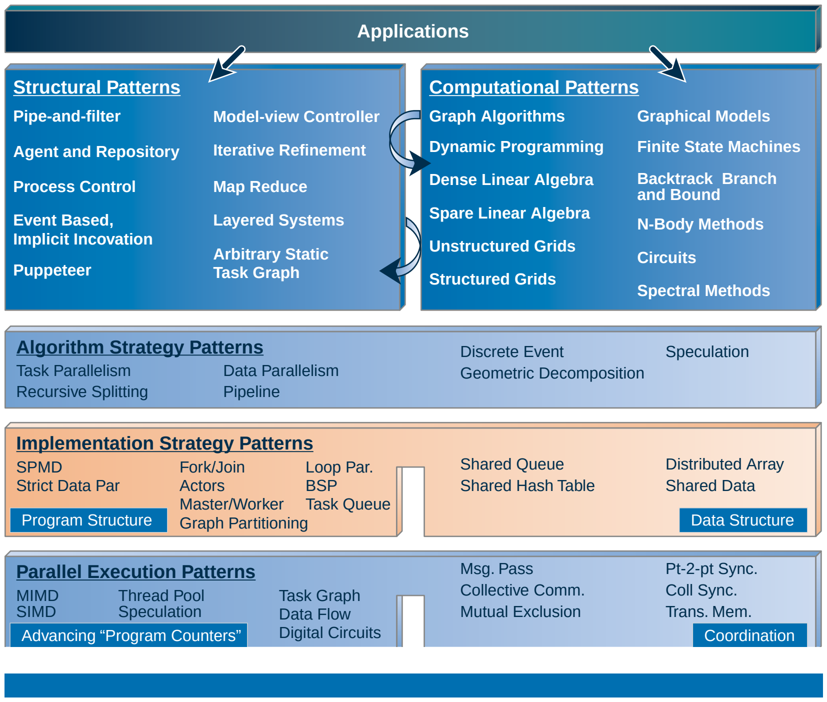


Figure 1: The Structure of OPL and the Five Categories of Design Patterns. Details About Each of the Patterns can be Found in [4].
Source: UC Berkeley ParLab, 2009

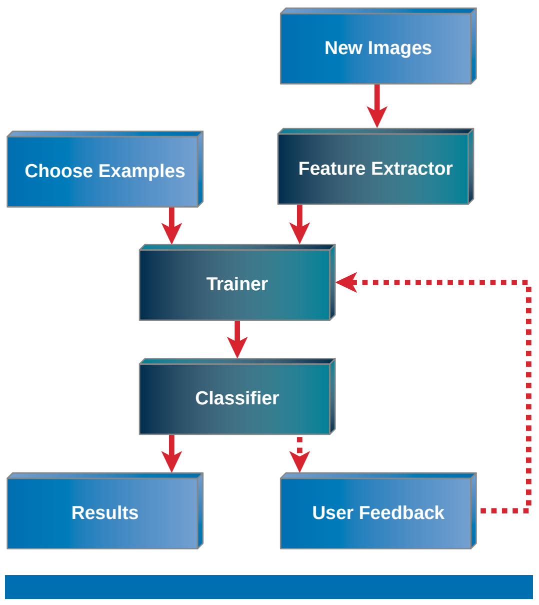


Figure 2: The CBIR Application Framework
Source: UC Berkeley ParLab, 2009

Case Study: Content-based Image Retrieval

Experience has shown that an easy way to understand patterns and how they are used is to follow an example. In this section we describe a problem and its parallelization by using patterns from OPL. In doing so we describe a subset of the patterns and give some indication of the way we make transitions between layers in the pattern language.

In particular, to understand how OPL can help software architecture, we use a content-based image retrieval (CBIR) application as an example. From this example (drawn from [14]), we show how structural and computational patterns can be used to describe the CBIR application and how the lower-layer patterns can be used to parallelize an exemplar component of the CBIR application.

In Figure 2 we see the major elements of our CBIR application as well as the data flow. The key elements of the application are the feature extractor, the trainer, and the classifier components. Given a set of new images the feature extractor will collect features of the images. Given the features of the new images, chosen examples, and some classified new images from user feedback, the trainer will train the parameters necessary for the classifier. Given the parameters from the trainer, the classifier will classify the new images based on their features. The user can classify some of the resulting images and give feedback to the trainer repeatedly in order to increase the accuracy of the classifier. This top-level organization of CBIR is best represented by the *Pipe-and-Filter* structural pattern. The feature-extractor, trainer, and classifier are filters or computational elements that are connected by pipes (data communication channels). Data flows through the succession of filters that do not share state and only take input from their input pipe(s). The filters perform the appropriate computation on those data and pass the output to the next filter(s) via its output pipe. The choice of *Pipe-and-Filter* pattern to describe the top-level structure of CBIR is not unusual. Many applications are naturally described by *Pipe-and-Filter* at the top level.

In our approach we architect software by using patterns in a hierarchical fashion. Each filter within the CBIR application contains a complex set of computations. We can parallelize these filters using patterns from OPL. Consider, for example, the classifier filter. There are many approaches to classification, but in our CBIR application we use a support-vector machine (SVM) classifier. SVM is widely used for classification in image recognition, bioinformatics, and text processing. The SVM classifier evaluates the function:

$$\hat{z} = \text{sgn}\left\{ b + \sum_{i=l}^{l} y_i\, \alpha_i\, \Phi\,(x_i\,,\, z) \right\}$$

where x_i is the i^th support vector, z is the query vector, Φ is the kernel function, α_i is the weight, y_i in {-1, 1} is the label attached to support vector x_i, b is a parameter, and sgn is the sign function. In order to evaluate the function quickly, we identified that the kernel functions are operating on the products and norms of x_i and z. We can compute the products between a set of query

"Many applications are naturally described by Pipe-and-Filter *at the top level."*

vectors and the support vectors by a BLAS level-3 operation with higher throughput. Therefore, we compute the products and norms first, use the results for computing the kernel values, and sum up the weighted kernel values. We architect the SVM classifier as shown in Figure 3. The basic structure of the classifier filter is itself a simple *Pipe-and-Filter* structure with two filters: the first filter takes the test data and the support vectors needed to calculate the dot products between the test data and each support vector. This dot product computation is naturally performed by using the *Dense-Linear-Algebra* computational pattern. The second filter takes the resulting dot products, and the following steps are to compute the kernel values, sum up all the kernel values, and scale the final results if necessary. The structural pattern associated with these computations is *Map-Reduce* (see the *Map-Reduce* sidebar).

In a similar way the feature-extractor and trainer filters of the CBIR application can be decomposed. With that elaboration we would consider the "high-level" architecture of the CBIR application complete. In general, to construct a high-level architecture of an application, we decompose the application hierarchically by using the structural and computational patterns of OPL.

Constructing the high-level architecture of an application is essential, and this effort improves not just the software viability but also eases communication regarding the organization of the software. However, there is still much work to be done before we have a working software application. To perform this work we move from the top layers of OPL (structural and computational patterns) down into lower layers (concurrent algorithmic strategy patterns etc.). To illustrate this process we provide additional detail on the SVM classifier filter.

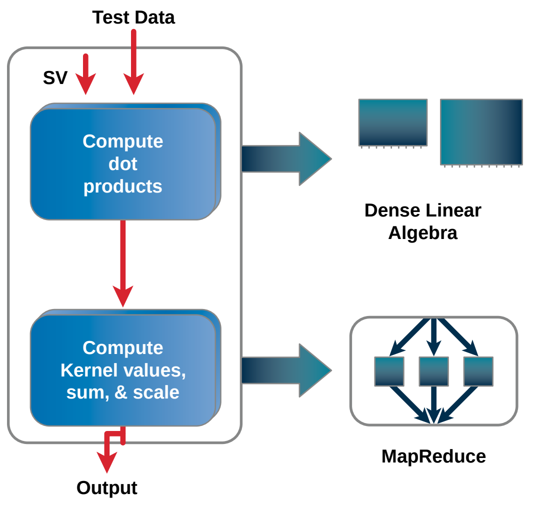


Figure 3: The Major Computations of the SVM Classifier
Source: UC Berkeley ParLab, 2009

Concurrent Algorithmic Strategy Patterns

After identifying the structural patterns and the computational patterns in the SVM classifier, we need to find appropriate strategies to parallelize the computation. In the *Map-Reduce* pattern the same computation is *mapped* to different non-overlapping partitions of the state set. The results of these computations are then gathered, or *reduced*. If we are interested in arriving at a parallel implementation of this computation, then we define the *Map-Reduce* structure in terms of a Concurrent Algorithmic Strategy. The natural choices for Algorithmic Strategies are the *Data-Parallelism* and *Geometric-Decomposition* patterns. By using the *Data-Parallelism* pattern we can compute the kernel value of each dot product in parallel (see the *Data-Parallelism* sidebar). Alternatively, by using the *Geometric-Decomposition* pattern (see the *Geometric-Decomposition* sidebar) we can divide the dot products into regular chunks of data, apply the dot products locally on each chunk, and then apply a global reduce to compute the summation over all chunks for the final results. We are interested in designs that can utilize large numbers of cores. Since the solution based on the *Data-Parallelism* pattern exposes more concurrent tasks (due to the large numbers of dot products) compared to the more coarse-grained geometric decomposition solution, we choose the *Data-Parallelism* pattern for implementing the map reduce computation.

Structural Pattern: *Map-Reduce*
Solution: A solution is structured in two phases: (1) a map phase where items from an "input data set" are mapped onto a "generated data set" and (2) a reduction phase where the generated data set is reduced or otherwise summarized to generate the final result. It is easy to exploit concurrency in the map phase, since the map functions are applied independently for each item in the input data set. The reduction phase, however, requires synchronization to safely combine partial solutions into the final result.

Algorithm Strategy Pattern:
Geometric-Decomposition
Solution: An algorithm is organized by (1) dividing the key data structures within a problem into regular chunks, and (2) updating each chunk in parallel. Typically, communication occurs at chunk boundaries so an algorithm breaks down into three components: (1) exchange boundary data, (2) update the interiors or each chunk, and (3) update boundary regions. The size of the chunks is dictated by the properties of the memory hierarchy to maximize reuse of data from local memory/cache.

The use of the *Data-Parallelism* algorithmic strategy pattern to parallelize the Map-Reduce computation is shown in the pseudo code of the kernel value calculation and the summation. These computations can be summarized as shown in Figure 4. Line 1 to line 4 is the computation of the kernel value on each dot product, which is the map phase. Line 5 to line 13 is the summation over all kernel values, which is the reduce phase. Function NeedReduce checks whether element "i" is a candidate for the reduction operation. If so, the ComputeOffset function calculates the offset between element "i" and another element. Finally, the Reduce function conducts the reduction operation on element "i" and "i+offset".

Implementation Strategy Patterns

To implement the data parallelism strategy from the Map-Reduce pseudo-code, we need to find the best Implementation Strategy Pattern. Looking at the patterns in OPL, both the *Strict-Data-Parallel* and *Loop-Parallel* patterns are applicable.

Whether we choose the *Strict-data-parallel* or *Loop-parallel* patterns in the implementation layer, we can use the *SIMD* pattern for realizing the execution. For example, we can apply SIMD on line 2 in Code Listing 1 for calculating the kernel value of each dot product in parallel. The same concept can be used on line 7 in Code Listing 1 for conducting the checking procedure in parallel. Moreover, in order to synchronize the computations on different processing elements on line 4 and line 12 in Code Listing 1, we can use the barrier construct described within the *Collective-Synchronization* pattern for achieving this goal.

Implementation Strategy Pattern:
Strict-Data-Parallel
Solution: Implement a data parallel algorithm as a single stream of instructions applied concurrently to the elements of a data set. Updates to each element are either independent, or they involve well-defined collective operations such as reductions or prefix scans.

```
function ComputeMapReduce( DotProdAndNorm, Result) {
1  for i ← 1 to n {
2    LocalValue[i] ←
                ComputeKernelValue(DotProdAndNorm[i]);
3      }
4      Barrier();
5      for reduceLevel ← 1 to MaxReduceLevel {
6      for i ← 1 to n {
7        if (NeedReduce(i, reduceLevel) ) {
8              offset ← ComputeOffset(i, reduceLevel);
9              LocalValue[i] ← Reduce(LocalValue[i],
                    LocalValue[i+offset]);
10           }
11       }
12       Barrier();
13  }
14}
```

Code Listing 1: Pseudo Code of the Map Reduce Computation
Source: Intel Corporation, 2009

In summary, the computation of the SVM classifier can be viewed as a composition of the *Pipe-and-Filter*, *Dense-Linear-Algebra*, and *Map-Reduce* patterns. To parallelize the *Map-Reduce* computation, we used the *Data-Parallelism* pattern. To implement the *Data-Parallelism* Algorithmic Strategy, both the *Strict-Data-Parallel* and *Loop-Parallel* patterns are applicable. We choose the *Strict-Data-Parallel* pattern, since it seemed a more natural choice given the fact we wanted to expose large amounts of concurrency for use on many-core chips with large numbers of cores. It is important to appreciate, however, that this is a matter of style, and a quality design could have been produced by using the *Loop-Parallel* pattern as well. To map the *Strict-Data-Parallel* pattern onto a platform for execution, we chose a *SIMD* pattern. While we did not show the details of all the patterns used, along the way we used the *Shared-Data* pattern to define the synchronization protocols for the reduction and the *Collective-Synchronization* pattern to describe the barrier construct. It is common that these functions (reduction and barrier) are provided as part of a parallel programming environment; hence, while a programmer needs to be aware of these constructs and what they provide, it is rare that they will need to explore their implementation in any detail.

"We choose the Strict-Data-Parallel pattern …. however, that is a matter of style … a quality design could have been produced using the Loop-Parallelism pattern as well."

Other Patterns

OPL is not complete. Currently OPL is restricted to those parts of the design process associated with architecting and implementing applications that target parallel processors. There are countless additional patterns that software development teams utilize. Probably the best known example is the set of design patterns used in object-oriented design [8]. We made no attempt to include these in OPL. An interesting framework that supports common patterns in parallel object-oriented design is Thread Building Blocks (TBB) [15].

"OPL focuses on patterns that are ultimately expressed in software."

OPL focuses on patterns that are ultimately expressed in software. These patterns do not, however, address methodological patterns that experienced parallel programmers use when designing or optimizing parallel software. The following are some examples of important classes of methodological patterns.

- *Finding Concurrency patterns* [7]. These patterns capture the process that experienced parallel programmers use when exploiting the concurrency available in a problem. While these patterns were developed before our set of Computational patterns was identified, they appear to be useful when moving from the Computational patterns category of our hierarchy to the Parallel Algorithmic Strategy category. For example, applying these patterns would help to indicate when geometric decomposition is chosen over data parallelism as a dense linear algebra problem moves toward implementation.

- *Parallel Programming "Best Practices" patterns*. This describes a broad range of patterns we are actively mining as we examine the detailed work in creating highly-efficient parallel implementations. Thus, these patterns appear to be useful when moving from the Implementation Strategy patterns to the Concurrent Execution patterns. For example, we are finding common patterns associated with optimizing software to maximize data locality.

"We can define a systematic methodology for software architecture in terms of design patterns *and a* pattern language.*"*

There is a growing community of programmers and researchers involved in the creation of OPL. The current status of OPL, including the most recent updates of patterns, can be found at: ***http://parlab.eecs.berkeley.edu/wiki/patterns/patterns***. This website also has links to details on our shorter monthly patterns workshop as well as our longer, two-day, formal patterns workshop. We welcome your participation.

Summary, Conclusions, and Future Work

We believe that the key to addressing the challenge of writing software is to architect the software. In particular, we believe that the key to addressing the new challenge of programming multi-core and many-core processors is to carefully architect the parallel software. We can define a systematic methodology for software architecture in terms of *design patterns* and a *pattern language*. Toward this end we have taken on the ambitious project of creating a comprehensive pattern language that stretches all the way from the initial software architecture of an application down to the lowest-level details of software implementation.

"We also need to continue mining patterns from existing parallel software to identify patterns that may be missing from our language."

OPL is a work in progress. We have defined the layers in OPL, listed the patterns at each layer, and written text for many of the patterns. Details are available online [4]. On the one hand, much work remains to be done. On the other hand, we feel confident that our structural patterns capture the critical ways of composing software, and our computational patterns capture the key underlying computations. Similarly, as we move down through the pattern language, we feel that the patterns at each layer do a good job of addressing most of the key problems for which they are intended. The current state of the textual descriptions of the patterns in OPL is somewhat nascent. We need to finish writing the text for some of the patterns and have them carefully reviewed by experts in parallel applications programming. We also need to continue mining patterns from existing parallel software to identify patterns that may be missing from our language. Nevertheless, last year's effort spent in mining five applications netted (only) three new patterns for OPL. This shows that while OPL is not fully complete, it is not, with the caveats described earlier, dramatically deficient.

Complementing the efforts to mine existing parallel applications for patterns is the process of architecting new applications by using OPL. We are currently using OPL to architect and implement a number of applications in areas such as machine learning, computer vision, computational finance, health, physical modeling, and games. During this process we are watching carefully to identify where OPL helps us and where OPL does not offer patterns to guide the kind of design decisions we must make. For example, mapping a number of computer-vision applications to new generations of many-core architectures helped identify the importance of a family of data layout patterns.

The scope of the OPL project is ambitious. It stretches across the full range of activities in architecting a complex application. It has been suggested that we have taken on too large of a task; that it is not possible to define the complete software design process in terms of a single design pattern language. However, after many years of hard work, nobody has been able to solve the parallel programming problem with specialized parallel programming languages or tools that automate the parallel programming process. We believe a different approach is required, one that emphasizes how people think about algorithms and design software. This is precisely the approach supported by design patterns, and based on our results so far, we believe that patterns and a pattern language may indeed be the key to finally resolving the parallel programming problem.

"Mapping a number of computer-vision applications to new generations of many-core architectures helped identify the importance of a family of data layout patterns."

While this claim may seem grandiose, we have an even greater aim for our work. We believe that our efforts to identify the core computational and structural patterns for parallel programming has led us to begin to identify the core computational elements (computational patterns, analogous to atoms) and means of assembling them (structural patterns, analogous to molecular bonding) of *all* electronic systems. If this is true, then these patterns not only serve as a means to assist software design but can be used to architect a curriculum for a true discipline of computer science.

References

[1] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. "A View of the Parallel Computing Landscape." *Communications of the ACM, volume 51*, pages 56-67, 2009.

[2] B. Chapman, G. Jost, and R van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT press, Cambridge, Massachusetts, 2008.

[3] W. Gropp, E. Lusk, A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. 2nd edition, MIT Press, Cambridge, Massachusetts, 1999.

[4] ***http://parlab.eecs.berkeley.edu/wiki/patterns/patterns***

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1994.

[6] C. Alexander, S. Ishikawa, M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, New York, 1977.

[7] T. G. Mattson, B. A. Sanders, B. L. Massingill. *Patterns for Parallel Programming*. Addison Wesley, Boston, Massachusetts, 2004.

[8] D. Garlan and M. Shaw. "An introduction to software architecture." *Carnegie Mellon University Software Engineering Institute Report* CMU SEI-94-TR-21, Pittsburg, Pennsylvania, 1994.

[9] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River, New Jersey, 1995.

[10] K. Asanovic, et al. "The landscape of parallel computing research: A view from Berkeley." *EECS Department, University of California, Berkeley*, Technical Report UCB/EECS-2006-183, 2006.

[11] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley, Hoboken, New Jersey, 1996.

[12] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." *In Proceedings of OSDI '04: 6th Symposium on Operating System Design and Implementation*. San Francisco, CA, December 2004.

[13] L. G. Valiant, "A Bridging Model for Parallel Computation." *Communication of the ACM*, volume 33, pages 103-111, 1990.

[14] Catanzaro, B., B. Su, N. Sundaram, Y. Lee, M. Murphy, and K. Keutzer. "Efficient, High-Quality Image Contour Detection." *IEEE International Conference on Computer Vision* (ICCV09), pages 2381-2388, Kyoto Japan, 2009.

[15] J. Reinders. *Intel Threaded Building Blocks*. O'Reilly Press, Sebastopol, California, 2007.

Acknowledgments

The evolution of OPL has been strongly influenced by the collaborative environment provided by Berkeley's Par Lab. The development of the language has been positively impacted by students and visitors in two years of graduate seminars focused on OPL: Hugo Andrade, Chris Batten, Eric Battenberg, Hovig Bayandorian, Dai Bui, Bryan Catanzaro, Jike Chong, Enylton Coelho, Katya Gonina, Yunsup Lee, Mark Murphy, Heidi Pan, Kaushik Ravindran, Sayak Ray, Erich Strohmaier, Bor-yiing Su, Narayanan Sundaram, Guogiang Wang, and Youngmin Yi. The development of OPL has also received a boost from Par Lab faculty — particularly Krste Asanovic, Jim Demmel, and David Patterson. Monthly pattern workshops in 2009 also helped to shape the language. Special thanks to veteran workshop moderator Ralph Johnson as well as to Jeff Anderson-Lee, Joel Jones, Terry Ligocki, Sam Williams, and members of the Silicon Valley Patterns Group.

Authors' Biographies

Kurt Keutzer. After receiving his Ph.D. degree in Computer Science from Indiana University in 1984, Kurt joined AT&T Bell Laboratories where he was a Member of Technical Staff in the last of the golden era of Bell Labs Research. In 1991 he joined Synopsys, Inc. where he served in a number of roles culminating in his position as a Chief Technical Officer and Senior Vice-President of Research. Kurt left Synopsys in January 1998 to become Professor of Electrical Engineering and Computer Science at the University of California at Berkeley. At Berkeley he worked with Richard Newton to initiate the MARCO-funded Gigascale Silicon Research Center and was Associate Director of the Center from 1998 until 2002. He is currently a principal investigator in Berkeley's Universal Parallel Computing Research Center.

Kurt has researched a wide number of areas related to both the design and programming of integrated circuits, and his research efforts have led to four best-paper awards. He has published over 100 refereed publications and co-authored six books, his latest being *Closing the Power Gap Between ASIC and Custom*. Kurt was made a Fellow of the IEEE in 1996.

Tim Mattson. Tim received a Ph.D. degree for his work on quantum molecular scattering theory from UC Santa Cruz in 1985. Since then he has held a number of commercial and academic positions working on the application of parallel computers to mathematics libraries, exploration geophysics, computational chemistry, molecular biology, and bioinformatics.

Dr. Mattson joined Intel in 1993. Among his many roles he was applications manager for the ASCI Red Computer (the world's first TeraFLOP computer), helped create OpenMP, founded the Open Cluster Group, led the applications team for the first TeraFLOP CPU (the 80-core tera-scale processor), launched Intel's programs in computing for the Life Sciences, and helped create OpenCL.

Currently, Dr. Mattson is a Principal Engineer in Intel's Visual Applications Research Laboratory. He conducts research on performance modeling and how different programming models map onto many-core processors. Design patterns play a key role in this work and help keep the focus on technologies that help the general programmer solve real parallel programming problems.

Copyright

HARDWARE AND SOFTWARE APPROACHES FOR DETERMINISTIC MULTI-PROCESSOR REPLAY OF CONCURRENT PROGRAMS

Contributors

Gilles Pokam
Intel Corporation

Cristiano Pereira
Intel Corporation

Klaus Danne
Intel Corporation

Lynda Yang
University of Illinois at Urbana-Champaign

Sam King
University of Illinois at Urbana-Champaign

Josep Torrellas
University of Illinois at Urbana-Champaign

Index Words

Concurrent Programs
Deterministic Replay Debugging
Fault Tolerance
Non-determinism
Memory Race Recording
Chunks

Abstract

As multi-processors become mainstream, software developers must harness the parallelism available in programs to keep up with multi-core performance. Writing parallel programs, however, is notoriously difficult, even for the most advanced programmers. The main reason for this lies in the non-deterministic nature of concurrent programs, which makes it very difficult to reproduce a program execution. As a result, reasoning about program behavior is challenging. For instance, debugging concurrent programs is known to be difficult because of the non-determinism of multi-threaded programs. Malicious code can hide behind non-determinism, making software vulnerabilities much more difficult to detect on multi-threaded programs.

In this article, we explore hardware and software avenues for improving the programmability of Intel® multi-processors. In particular, we investigate techniques for reproducing a non-deterministic program execution that can efficiently deal with the issues just mentioned. We identify the main challenges associated with these techniques, examine opportunities to overcome some of these challenges, and explore potential usage models of program execution reproducibility for debugging and fault tolerance of concurrent programs.

Introduction

A common assumption of many application developers is that software behaves deterministically: given program A, running A on the same machine several times should produce the same outcome. This assumption is important for application performance, as it allows one to reason about program behavior. Most single-threaded programs executing on uni-processor systems exhibit this property because they are inherently sequential. However, when executed on multi-core processors, these programs need to be re-written to take advantage of all available computing resources to improve performance. Writing parallel programs, however, is a very difficult task because parallel programs tend to be non-deterministic by nature: running the same parallel program A on the same multi-core machine several times can potentially lead to different outcomes for each run. This makes both improving performance and reasoning about program behavior very challenging.

Deterministic multi-processor replay (DMR) can efficiently deal with the non-deterministic nature of parallel programs. The main idea behind DMR is reproducibility of program execution. Reproducing a multi-threaded program execution requires recording all sources of non-determinism, so that during replay, these threads can be re-synchronized in the same way as in the original execution. On modern chip multi-processor (CMP) systems, the sources of non-determinism can be either input non-determinism (data inputs, keyboard, interrupts, I/O, etc.) or memory non-determinism (access interleavings among threads). These sources of non-determinism can be recorded by using either software or hardware, or a combination of both.

Software-only implementations of DMR can run on legacy machines without hardware changes, but they suffer from performance slowdowns that can restrict the applicability of DMR. To achieve performance levels comparable to hardware schemes, software approaches can be backed up with hardware support. In this article, we describe what the software-only approaches for DMR may look like, and what types of hardware support may be required to mitigate their performance. Our discussion starts with the details of DMR: we focus on the usage models and on the main challenges associated with recording and replaying concurrent programs. We then describe several ways in which DMR schemes can be implemented in software, and we elaborate on the various tradeoffs associated with these approaches. Finally, we describe hardware extensions to software-only implementations that can help mitigate performance and improve the applicability of DMR.

"Deterministic multi-processor replay (DMR) can efficiently deal with the non-deterministic nature of parallel programs."

Why Record-and-Replay Matters

Recording and deterministically replaying a program execution gives computer users the ability to travel backward in time, recreating past states and events in the computer. Time travel is achieved by recording key events when the software runs, and then restoring to a previous checkpoint and replaying the recorded log to force the software down the same execution path.

This mechanism enables a wide range of applications in modern systems, especially in multi-processor systems in which concurrent programs are subject to non-deterministic execution: such execution makes it very hard to reason about or reproduce a program behavior.

- *Debugging*. Programmers can use time travel to help debug programs [36, 39, 15, 4, 1] including programs with non-determinism [20, 33], since time travel can provide the illusion of reverse execution and reverse debugging.
- *Security*. System builders can use time travel to replay the past execution of applications looking for exploits of newly discovered vulnerabilities [19], to inspect the actions of an attacker [12], or to run expensive security checks in parallel with the primary computation [9].
- *Fault tolerance*. System designers can use replay as an efficient mechanism for recreating the state of a system after a crash [5].

"Recording and deterministically replaying a program execution gives computer users the ability to travel backward in time."

Non-determinism of Concurrent Programs

"Some of the input is not deterministic across different runs of the program, even if the program's command line arguments are the same."

The goal of deterministic replay is to be able to reproduce the execution of a program in the way it was observed during recording. In order to reproduce an execution, each instruction should see the same input operands as in the original run. This should guarantee the same execution paths for each thread. During an execution, a program reads data from either memory or register values. Some of the input is not deterministic across different runs of the program, even if the program's command line arguments are the same. Hence, in order to guarantee determinism these inputs need to be recorded in a log and injected at replay. In this section, we describe these sources of non-determinism.

Deterministic replay can be done at different levels of the software stack. At the top level, one can replay only the user-level instructions that are executed. These include application code and system library code. This is the approach taken by BugNet [26], Capo [25], iDNA [3], and PinPlay [29]. At the lowest level, a system can record and replay all instructions executed in the machine, including both system-level and user-level instructions. Regardless of the level one is looking at, the sources of non-determinism can be divided into two sets: input read by the program and memory interleavings across different threads of execution. We now describe each source in more detail.

Input Non-determinism

"User-level replay has different requirements from those of system-level replay."

Input non-determinism differs, depending on which layer of the system is being recorded for replay. User-level replay has different requirements from those of system-level replay. Conceptually, the non-deterministic inputs are all the inputs that are consumed by the system layer being recorded that are not produced by the same layer. For instance, for user-level replay, all inputs coming from the operating system are non-deterministic, because there is no guarantee of repeatability across two runs. A UNIX* system call, such as *gettimeofday*, is inherently non-deterministic across two runs, for instance. For a system-level record, all inputs that are external to the system are non-deterministic inputs. External inputs are inputs coming from external devices (I/O, interrupts, DMAs). We now discuss the source of non-determinism at each level.

"For a system-level record, all inputs that are external to the system are non-deterministic inputs."

For user-level replay, the sources of non-determinism are listed as follows:

- *System calls*. Many system calls are non-deterministic. An obvious example is a timing-dependent call, such as the UNIX call *gettimeofday*. Other system calls can also be non-deterministic. A system call reading information from a network card may return different results, or a system-call reading from a disk may return different results.
- *Signals*. Programs can receive asynchronous signals that can be delivered at different times across two runs, making the control flow non-deterministic.

- *Special architectural instructions*. On x86 architecture, some instructions are non-deterministic, such as RDTSC (read timestamp) and RDPMC (read performance counters). Across processor generations of the same architecture, CPUID will also return different values, if the replay happens in a processor other than the one in which the recording happened.

In addition to the non-deterministic inputs just mentioned, other sources of non-determinism at the user-level are the location of the program stack that can change from run to run and the locations where dynamic libraries are loaded during execution. Although these are not inputs to the program, they also change program behavior and need to be taken care of for deterministic replay.

"Other sources of non-determinism at the user-level are the location of the program stack that can change from run to run and the locations where dynamic libraries are loaded during execution."

At the system-level, the major sources of non-determinism are the following:

- *I/O*. It is common for most architectures to allow memory mapped I/O: loads and stores effectively read from and write to devices. If one is replaying the operating system code, the reads from I/O devices are not guaranteed to be repeatable. As a result, the values read by those load instructions need to be recorded.
- *Hardware interrupts*. Hardware interrupts trigger the execution of an interrupt service routine, which changes the control flow of the execution. Interrupts are used to notify the processor that some data (e.g., disk read) are available to be consumed. An interrupt is delivered at any point in time during the execution of the operating system code. A recorder needs to log the point at which the interrupt arrived and the content of the interrupt (what its source is: e.g., disk I/O, network I/O, timer interrupt, etc.).
- *Direct Memory Access (DMA)*. Direct memory accesses perform writes directly to memory without the intervention of the processor. The values written by DMA as well as the timestamp at which those values were written need to be recorded to be reproducible during replay.

In addition, the results of processor-specific instructions, such as x86 RDTSC, also need to be recorded as is the case with user-level code, in order to ensure repeatability.

"A recorder needs to log the point at which the interrupt arrived and the content of the interrupt."

Memory Interleaving

Input non-determinism is present on single-core and multi-core machines. However, in multi-core machines, an additional source of non-determinism is present and that is the order in which all threads in the system access shared memory. This is typically known as memory races, where different runs of a program may result in different threads winning the race when trying to access a piece of shared memory. Memory races occur between synchronization operations (synchronization races) or between data accesses (data races). At the user-level, threads access memory in a different order, because the operating system may schedule them in a different order. This is due to interrupts being delivered at different times, because of differences in the architectural state (cache line misses, memory latencies, etc.) and also because of the load in the system. As a result, the shared memory values seen by each thread in different runs can change, resulting in different behavior for each thread across runs. This is the major source of non-determinism in multi-threaded programs. Races also occur among threads within the operating system, and the behavior across two runs is also not guaranteed to be the same. Hence the order in which races occur within the operating system code also needs to be recorded to guarantee deterministic replay.

"In multi-core machines, an additional source of non-determinism is present and that is the order in which all threads in the system access shared memory."

"The order in which races occur within the operating system code also needs to be recorded to guarantee deterministic replay."

Software Approaches for Deterministic Replay

Software-only approaches to record-and-replay (R&R) can be deployed on current commodity hardware at no cost. As described in the previous section, an R&R solution needs to tackle two issues: logging and replaying non-deterministic inputs and enforcing memory access interleavings. We describe software-only solutions to both of these challenges next, and we provide details on the techniques used in recent deterministic replay approaches extant in literature. Because there are more software-only R&R-like systems than can possibly be discussed in this article, we choose to mention those that best characterize our focus. Once we've surveyed the literature, we discuss the remaining open challenges in software-only solutions.

Reproducing Input Non-determinism

Systems and programs execute non-deterministically due to the external resources they are exposed to and the timing of these resources. Thus, these external resources can be all viewed as *inputs*, whether they are user inputs, interrupts, system call effects, etc. Given the same inputs and the same initial state, the behavior of the system or application is *deterministic*. The approach to R&R, therefore, is to log these inputs during the logging phase and inject them back during replay.

"An R&R solution needs to tackle two issues: logging and replaying non-deterministic inputs and enforcing memory access interleavings."

Table 1 summarizes the replay systems under discussion in terms of the level of replay (user-level or system-level), usage model, and how they are implemented for replaying inputs.

Replay System	**Level of Replay**	**Usage Model**	**Implementation**
Bressoud and Schneider [5]	System	Fault-tolerance	Virtual machine
CapoOne [25]	User	General notion of "time travel" for multiple purposes	Kernel modifications, *ptrace*
Flashback [36]	User	Debugging	Kernel modifications
iDNA [3]	User	Debugging, profiling	Dynamic instrumentation
Jockey [34]	User	Debugging	Library-based, rewrites system calls
Liblog [16]	User	Debugging	Library-based, intercepts calls to *libc*
ODR [2]	User	Debugging	Kernel modifications, *ptrace*
PinPlay [29]	User	Debugging, profiling	Dynamic instrumentation
R2 [17]	User	Debugging	Library-based, stubs for replayed function calls
ReVirt [13]	System	Security	Virtual machine
TTVM [20]	System	Debugging	Virtual machine
VMWare [38]	System	General replay	Virtual machine

Table 1: Summary of Approaches to Replaying Input Non-determinism
Source: Intel Corporation, 2009

User-level Input Non-determinism

First, let us consider user application replay. For the most part, we discuss how several approaches handle system calls and signals, since together they represent a large part of the non-deterministic external resources exposed to the application. They also represent resources that have inherently deterministic timing and non-deterministic timing, respectively.

System Calls

An application's interaction with the external system state is generally confined to its system calls. We discuss in detail how two recent replay systems — Flashback [36] and CapoOne [25] — handle these system calls. Flashback can roll back the memory state of a process to user-defined checkpoints, and it supports replay by logging the process's interaction with the system. Flashback's usage model is for debugging software. CapoOne can log and replay multiple threads and processes in a given replay group, cohesively, while concurrently supporting multiple independent replay groups. It re-executes the entire application during replay. CapoOne requires additional hardware to support multi-processor replay; however, its technique for enforcing an application's external inputs is completely software-based.

"Flashback can roll back the memory state of a process to user-defined checkpoints."

Both Flashback and CapoOne interpose on system call routines: they log the inputs, results, and side-effects (copy_to_user) of each system call, and they inject the data back in during re-execution of system call entry and exit points. If the effect of a given system call is isolated to only the user application (e.g., *getpid*()), the actual call is bypassed during replay, and its effects are emulated by injecting the values retrieved from the log. On the other hand, if a system call modifies a system state that is outside of the replayed application (e.g., *fork*()), the system call is re-executed during replay in a manner such that its effect on the application is the same as during the logging phase. CapoOne interposes on system calls in user space via the *ptrace* mechanism, while Flashback does so with kernel modifications. Another replay scheme called ODR [2] describes similar techniques to handle system calls, by using both *ptrace* and kernel modules. Jockey [34], a replay debugger, is slightly different from Flashback and CapoOne in that Jockey links its own shared-object file to the replayed application and then rewrites the system calls of interest.

"Functions above the user-defined interface are re-executed during replay."

While all of these approaches automatically define the interface at which logging and replay occur, namely the system call boundary, R2 [17] is a library-based replay debugger tool that allows the user to choose this demarcation. Functions above the user-defined interface are re-executed during replay, while those below it are emulated by using data from log files. Implementation-wise, R2 generates, and later calls, the stub associated with each function that needs to be logged or replayed. The authors of R2 also address the issue of preserving order between function calls that are executed by different threads. R2 uses a Lamport clock [21] to either serialize all calls or allow them to occur concurrently, as long as causal-order is maintained.

Signals

"Since signals are asynchronous and can occur at any point during the application's execution, they are a good example of a non-deterministic input that is time-related."

With system calls, we are only interested in recording their effects, since they always execute at the same point in a given application. This is, however, untrue for signals. The purpose of a signal is to notify an application of a given event, and since signals are asynchronous and can occur at any point during the application's execution, they are a good example of a non-deterministic input that is time-related. Although Flashback does not support signal replay, Flashback's developers suggest using the approach described in [35]: i.e., use the processor's instruction counter to log exactly when the signal occurred. During replay, the signal would be re-delivered when the instruction counter reaches the logged value. Jockey, on the other hand, delays all signals encountered during the logging phase until the end of the next system call, which it logs with that system call. Thus, during replay, the signal is re-delivered at the end of the same system call. CapoOne and liblog [16], another replay debugger, use a similar technique.

Dynamic Instrumentation

PinPlay [29] and iDNA [3] are replay systems that focus on the application debugging usage model: they are based on the dynamic binary instrumentation of a captured program trace. Non-deterministic input is logged and replayed, by tracking and restoring changes to registers and main memory. PinPlay replays asynchronous signals by logging the instruction count of where signals occur.

"Non-deterministic input is logged and replayed, by tracking and restoring changes to registers and main memory."

Full-system Input Non-determinism

We move on to consider approaches for software-based, full-system replay, which include ReVirt [13], TTVM [20], the system described by [5], and VMWare [38]. The first three were designed for the usage models of security, debugging, and fault tolerance. Perhaps, unsurprisingly, all of these methods take advantage of virtual machines.

ReVirt uses UMLinux [6], a virtual machine that runs as a process on the host. Hardware components and events of the guest are emulated by software analogues. For example, the guest hard disk is a host file, the guest CD-ROM is a host device, and guest hardware interrupt events are simulated by the host delivering a signal to the guest kernel. With these abstractions, ReVirt is able to provide deterministic replay by checkpointing the virtual disk and then logging and replaying the inputs that are external to the virtual machine. Similar to user-application replay, each external input may require that only the data associated with it need be logged, or additionally, it may require that a timing-factor for those that are asynchronous be logged as well. ReVirt logs the input from external devices such as the keyboard and CD-ROM, non-deterministic results returned by system calls from the guest kernel to the host kernel, and non-deterministic hardware instructions such as *RDTSC*. Guest hardware interrupts, emulated by signals, are asynchronous, and thus ReVirt has to ensure that these are delivered at the same point in the execution path. The authors chose to use the program counter and the hardware retired branches counter to uniquely identify the point to deliver the signal.

"ReVirt is able to provide deterministic replay by checkpointing the virtual disk and then logging and replaying the inputs that are external to the virtual machine."

TTVM uses ReVirt for its logging and replaying functionality, but makes changes that make it more suitable for its debugging usage model; for example, TTVM provides support for greater and more frequent checkpoints.

Reproducing Memory Access Non-determinism

The techniques we just described guarantee determinism for replaying single-threaded applications or multi-threaded applications where the threads are independent from one another. Deterministic replay of multi-threaded applications, with threads communicating via synchronization or through shared memory, require additional support.

"Deterministic replay of multi-threaded applications, with threads communicating via synchronization or through shared memory, require additional support."

Table 2 summarizes the replay systems we describe next in terms of usage model, multi-processor support, support for replaying data-races without additional analysis, and support for immediate replay without a state-exploration stage.

Replay System	**Usage Model**	**Multiprocessor Support?**	**Data Race Support?**	**Immediate Replay (no offline state-exploration stage)?**
DejaVu [8]	Debugging	No	Yes	Yes
iDNA [3]	Debugging, profiling	Yes	No	Yes
Instant Replay [23]	Debugging	Yes	No	Yes
Kendo [27]	Debugging, fault-tolerance	Yes	No	Yes
Liblog [16]	Debugging	No	Yes	Yes
ODR [2]	Debugging	Yes	Yes	No
PinPlay [29]	Debugging	Yes	Yes	Yes
PRES [28]	Debugging	Yes	Yes	No
RecPlay [32]	Debugging	Yes	No	Yes
Russinovich and Cogswell [33]	Debugging	No	Yes	Yes
SMP-ReVirt [11]	General replay	Yes	Yes	Yes

Table 2: Summary of Approaches to Replaying Memory Access Non-determinism
Source: Intel Corporation, 2009

Replay in Uniprocessors

In a uni-processor system, it was observed that since only one thread can run at any given time, recording the order of how the threads were scheduled on the processor is sufficient for later replaying of the memory access interleaving [16, 8, 33]. These solutions have been implemented at the operating-system level [33], virtual-machine level [8], and user level [16].

"On a multi-processor, thread-scheduling information is not sufficient for deterministic replay, since different threads can be running on different processors or cores concurrently."

Replay of Synchronized Accesses

On a multi-processor, thread-scheduling information is not sufficient for deterministic replay, since different threads can be running on different processors or cores concurrently. Earlier proposals, such as Instant Replay [23] and RecPlay [32], recorded the order of operations at a coarse granularity; that is, at the level of user-annotated shared objects and synchronization operations, respectively. Therefore, these schemes were only able to guarantee deterministic replay for data-race free programs. Both proposals were designed with debugging in mind. As an illustrative example, Instant Replay used the concurrent-read-exclusive-write (CREW) [10] protocol when different threads wanted access to a shared object. CREW guarantees that when a thread has permission to write to a shared object, no other threads are allowed to write to or read from that object. On the other hand, multiple threads can read from the object concurrently. Instant Replay uses the recorded sequence of write operations and the "version" number of the object for each read operation during replay.

Some recent proposals also do not support deterministic replay of programs with data races. iDNA [3] schedules a thread's execution trace according to instruction sequences that are ordered via synchronization operations. Kendo [27] offers deterministic multi-threading in software by assuring the same sequence of lock acquisition for a given input. While not technically a replay system, Kendo also requires that programs be correctly synchronized. Kendo's usage models include debugging and support for fault-tolerant replicas.

"iDNA [3] schedules a thread's execution trace according to instruction sequences that are ordered via synchronization operations."

Replay with State-exploration

ODR [2] and PRES [28] are two novel approaches that facilitate replay debugging, but are not able to immediately replay an application, given the log data during the logging phase. Instead, they intelligently explore the space of possible program execution paths until the original output or bug is reproduced. Such analysis must be done off-line, but ODR and PRES gain in having smaller logging phase overtimes (since they log less data) compared to software schemes that provide for immediate replay.

PinPlay and SMP-ReVirt

PinPlay [29] and SMP-ReVirt [11, 14] provide for immediate replay, and they order shared memory operations rather than coarse-grained objects.

PinPlay's approach is to implement a software version of the flight data recorder (FDR) [37]. FDR exploits cache coherence messages to find memory access dependencies and to order pairs of instructions.

"FDR exploits cache coherence messages to find memory access dependencies and to order pairs of instructions."

SMP-ReVirt is a generalization of the CREW protocol for shared objects in Instant Replay [23] to shared pages of memory. A given page in memory can only be in a state that is concurrently read or exclusively written during the logging phase. These access controls are implemented by changing a thread's page permissions — read-access, write-access, or no-access for a given page — during the system's execution. For example, if a thread wants to write to a page and thus needs to elevate its permission to write-access, all other threads must have their permissions reduced to no-access first. Each thread has its own log. When a thread has its page permission elevated during logging, it logs the point at which it received the elevated permission and the points where the other threads reduced their page permissions. Additionally, the threads that had their permissions reduced log the same points where their permissions were reduced. SMP-ReVirt specifies these "points" in the execution of the system by means of instruction counts. The instructions count of each processor is also updated in a globally visible vector. Thus, during replay, when a thread encounters a page permission elevation entry, it waits until the other permission-reducing threads reach the instruction count value indicated in the log. On the other hand, when a thread encounters a page permission reduction entry, it updates the global vector with its instruction count.

Challenges in Software-only Deterministic Replay

It has been shown that designing a software-only solution for recording and replaying input non-determinism is reasonable in terms of execution speed, and it can be done with an overhead of less than ten percent [20, 28, 36]. It is difficult to compare and summarize input log size growth rates for the different approaches discussed here, since different approaches log different events, may compress the log differently, and use different applications as their benchmarks. However, it can be noted that Flashback's [36] log size is linear to the number of system call invocations. Other similar input logging techniques may likely exhibit similar behavior. In short, enforcing input determinism in software seems to be a reasonable approach, considering the low overhead.

"Enforcing input determinism in software seems to be a reasonable approach, considering the low overhead."

Conversely, the overhead incurred in enforcing memory access interleaving in software is a different story. SMP-ReVirt [11, 14] and PinPlay [29] allow for the most flexible and immediate replay, but they incur a huge overhead. Since SMP-ReVirt instruments and protects shared memory at the page level of granularity, it has issues with false sharing and page contention [28], especially as the number of processors increases [14]. With four CPUs, the logging phase runtime of an application in SMP-ReVirt can be up to 9 times that of a native run [14]. PinPlay, like iDNA, which uses dynamic instrumentation and has a 12 to 17 times slowdown [3], cannot be turned on all the time.

The rest of the schemes previously described for replaying multi-threaded applications are either less flexible (uniprocessor only [8, 16, 33], data-race free programs only [3, 23, 27, 32]), or they trade off short on-line recording times with potentially long off-line state exploration times for replay [2, 28].

"Another challenge with software-based schemes is their ability to pinpoint asynchronous events during replay."

Another challenge with software-based schemes is their ability to pinpoint asynchronous events during replay. This issue was exemplified earlier in reference to asynchronous signals and interrupts. While some replay schemes choose to use hardware performance counters in their implementation [36, 35, 13], others choose to delay the event until a later synchronous event occurs [25, 34, 16]. The latter solution, though simpler, can theoretically affect program correctness, while the former solution requires the use of performance counters that are often inaccurate and non-deterministic [11, 27].

In the end, the selection of an appropriate replay system depends on the usage model. If we are to assume a debugging model where a programmer may not mind waiting a while for a bug to be reproduced, large replay overheads, though not desirable, may be reasonable. In fact, for most of the methods described here, the developers assumed a debugging usage model. Alternatively, a fault-tolerance replay model would require that backup replicas be able to keep up with the production replica, and thus good performance would be much more important. Note that performance is not the only factor that should be considered when determining which replay system works best with a usage model. For example, if the usage model is to replay system intrusions, it would be more suitable to use a full-system replay scheme rather than a user-application replay scheme.

"In the end, the selection of an appropriate replay system depends on the usage model."

Hardware Support for Recording Memory Non-determinism

Deterministically replaying a program execution is a very difficult problem, as we just described. In addition to logging input non-determinism, existing software approaches have to record the interleavings of shared memory accesses. This can be done at various levels of granularity (e.g., page level or individual memory operations), but as discussed previously, the overhead incurred can be prohibitive and therefore detrimental to applications of R&R, such as fault-tolerance. For this reason, there has been a lot of emphasis on providing hardware support for logging the interleavings of shared memory accesses more efficiently. We call the proposed mechanisms for logging the order in which memory operations interleave memory race recorders (MRR).

"There has been a lot of emphasis on providing hardware support for logging the interleavings of shared memory accesses more efficiently."

Prior work on hardware support for MRR piggybacks on timestamps located on cache coherence messages and logs the outcome of memory races by using either a *point-to-point* or a *chunk-based* approach. In this section we describe these two approaches and suggest directions for making them practical in modern multi-processor systems.

Point-to-point Approach

In point-to-point approaches [26, 37], memory dependencies are tracked at the granularity level of individual shared memory operations. In this approach, each memory block has a timestamp, and each memory operation updates the timestamp of the accessed block. In general, a block can be anything ranging from a memory word to multiple memory words [37, 31]. We now describe the FDR [37], a state-of-the-art implementation of a point-to-point MRR approach.

FDR augments each core in a multi-processor system with an instruction counter (IC) that counts retired instructions. FDR further augments each cache line with a cache instruction count (CIC) that stores the IC of the last store or load instruction that accessed the cache line (see Figure 1). When a core receives a remote coherence request to a cache line, it includes the corresponding CIC and its core ID in the response message. The requesting core can then log a dependency by storing the ID and CIC of the responding core and the current IC of the requesting core. To reduce the amount of information logged by the requesting core, a dependency is logged only if it cannot be inferred by a previous one. This optimization is called *transitive reduction*. For example, in Figure 1, only the dependency from *T1:W(b) to T2:R(b)* is logged, as *T1:R(a) to T2:W(a)* is consequentially implied by *T1:W(b) to T2:R(b)*. Transitive reduction is implemented by augmenting each core with a vector instruction count that keeps track of the latest CIC received by each core.

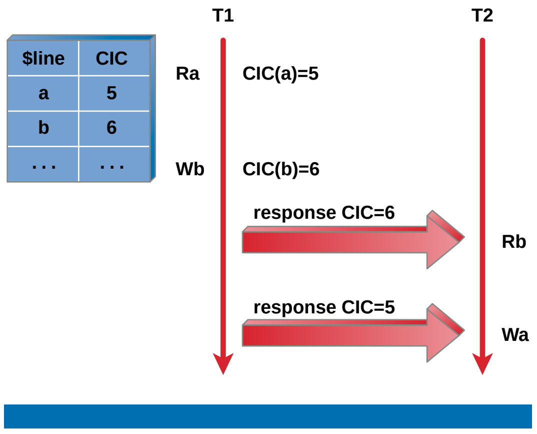


Figure 1: Point-to-point Approach
Source: Intel Corporation, 2009

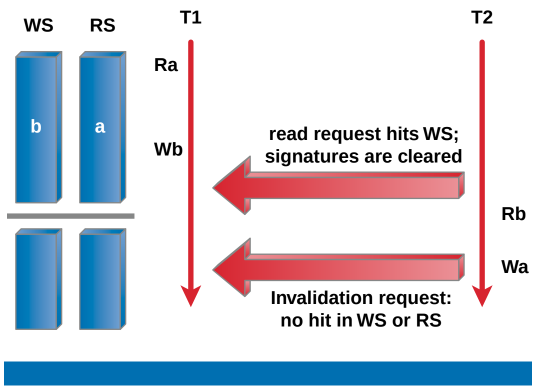

WS
RS
T1
T2
b
a
Ra
Wb
read request hits WS; signatures are cleared
Rb
Wa
Invalidation request: no hit in WS or RS

Figure 2: Chunk-based Approach
Source: Intel Corporation, 2009

Chunk-based Approach

A chunk defines a block of memory instructions that executes in isolation, i.e., without a remote coherence request intervening and causing a conflict. Chunks are represented by using signatures, which are hardware implementations of Bloom Filters. Signatures are used to compactly represent sets of locally accessed read or write memory addresses and to disambiguate a remote shared memory reference against them. A conflict with the signatures ends a chunk and clears the signatures.

Similar to point-to-point approaches, chunk-based approaches can also take advantage of transitive reduction to reduce the amount of logged information. As shown in Figure 2, the remote read *T2:R(b)* conflicts with the write signature of T1 and causes T1 to end its chunk and to clear its signatures. Consequently, the request *T2:W(a)* does not conflict and the dependency *T1:R(a) to T2:W(a)* is implied. In contrast to point-to-point approaches in which a timestamp is stored with each memory block, chunk-based approaches only need to store a timestamp per core to order chunks between threads.

We now describe two similar implementations of chunk-based approaches.

Rerun

In Rerun [18], episodes are like chunks. Rerun records a memory dependency by logging the length of an episode along with a timestamp. To identify the episodes that need to be logged, Rerun augments each core with a read and a write signature that keep track of the cache lines read and written by that core during the current episode, respectively. When a cache receives a remote coherence request, it checks its signatures to detect a conflict. If a conflict is detected, the core ends its current episode, which involves clearing the read and write signatures, creating a log entry containing the length of the terminating episode along with its timestamp, and updating the timestamp value. The timestamp represents a scalar clock maintained by each core to provide a total order among the episodes. The cores keep their timestamp up to date by piggybacking them on each cache coherence reply.

Deterministic replay is achieved by sequentially executing the episodes in order of increasing timestamps. To do so, a replayer typically examines the logs to identify which thread should be dispatched next and how many instructions it is allowed to execute until an episode of a different thread needs to be replayed.

"If a conflict is detected with a chunk, its signatures are cleared and the chunk is squashed and re-executed."

DeLorean

Similar to Rerun, DeLorean [24] also logs chunks by using signatures, but does so in a different multi-processor execution environment. In this environment, cores continuously execute chunks that are separated by register checkpoints. A chunk execution in this environment is speculative, i.e., its side effects are visible to other cores only until after commit. Before a chunk can commit, however, its signatures are compared against the signatures of other chunks to detect conflicts. If a conflict is detected with a chunk, its signatures are cleared and the chunk is squashed and re-executed. While such an execution environment is not standard in today's multi-processors, it has been shown to perform well [7]. The required hardware extensions are similar to hardware-supported transactional memory systems [22].

To enable deterministic replay, DeLorean logs the total order of chunk commits. Because in this execution environment chunks have a fixed size, e.g., 1000 dynamic instructions, no additional information needs to be logged, except in the rare cases where chunks need to end early because of events such as interrupts. Consequently, the log size of DeLorean is about one order of magnitude smaller than in Rerun. DeLorean can even reduce the log size by another order of magnitude when operating in *PicoLog* mode. In this execution mode, the architecture commits chunks in a deterministic order, e.g., round robin. Although this execution mode sacrifices performance, DeLorean only needs to log the chunks that end due to non-deterministic events, such as interrupts.

"The log size of DeLorean is about one order of magnitude smaller than in Rerun."

Making Memory Race Recorders Practical for Modern CMPs

MRR approaches discussed so far are effective in logging the events required for deterministic replay, but they also impose some non-negligible amount of complexity and performance cost that can preclude hardware vendors from deploying similar solutions in real products. In this section, we pinpoint some of these issues and discuss possible ways to remedy them.

Implementation Complexity

Showstoppers with previous MRR approaches are the implementation complexity of proposed techniques and their associated hardware cost.

With point-to-point approaches, for instance, a hardware estate for storing the timestamp of each accessed memory block is required. If the granularity of a memory block is a cache line, then each cache line must be augmented with storage for the timestamp. Because a cache line eviction throws away the information stored into it, the timestamp must also be stored at the next cache level to reduce logging frequency. FDR estimates this cost to be ~6 percent of the capacity of a 32KB L1 cache.

"Showstoppers with previous MRR approaches are the implementation complexity of proposed techniques and their associated hardware cost."

The main hardware cost associated with chunk-based approaches lies in the storage required for signatures. In Rerun, for instance, the authors suggest using 1024-bit signatures to store read memory addresses and 256-bit signatures to store written memory addresses. In contrast to point-to-point approaches, these changes do not require modifications to the cache sub-system and are therefore less invasive. However, there is some complexity involved in implementing signatures. The authors in [30] show that implementing signatures in modern CMPs involves subtle interactions with the hierarchy and policy decisions of on-chip caches. The authors show that the signature placement in a multi-level cache hierarchy can degrade performance by increasing the traffic to the caches. They propose hybrid L1/L2 signature placement strategies to mitigate this performance degradation.

"The main hardware cost associated with chunk-based approaches lies in the storage required for signatures."

Performance Overhead

With the exception of DeLorean, all MRR approaches discussed in this section must piggyback on cache coherence messages to maintain ordering among events in the system. For instance, in FDR, the core ID and the CIC are piggybacked on each coherence reply to log a point-to-point dependency between two instructions from different threads, whereas in Rerun a timestamp is piggybacked on each coherence reply to maintain causal ordering between chunks. This overhead can hurt performance by putting a lot of pressure on the bandwidth. FDR and Rerun, for instance, report a performance cost of ~10 percent, an estimation based on functional simulation. Ideally, we want this coherence traffic overhead to be nonexistent in real implementations of MRR. One way to attain this objective with a chunk-based approach has recently been proposed in [30]. The authors make the observation that maintaining causality at the chunk boundary is all that is needed to order chunks. Doing so eliminates the requirement to piggyback a timestamp on each coherence message. Using this approach, they show that the coherence traffic overhead can be reduced by several orders of magnitude compared to Rerun or FDR.

"The authors make the observation that maintaining causality at the chunk boundary is all that is needed to order chunks."

"DeLorean and FDR can replay a program at production run speed."

Replay Performance

As discussed previously, there are plenty of applications that can benefit from deterministic replay. Each of these applications places different replay speed requirements on the system. For instance, while an application developer can easily accommodate slow replay during debugging, this is not the case for high-availability applications in which the downtime window during recovery must be shortened. Slow replay in this case can have devastating effects on the system. Instead, we would like a second machine to continuously replay the execution of the primary machine at a similar speed, and to be able to take over instantly if the primary fails. Ideally, we do not want a R&R system to be constrained by speed, because such a constraint would limit the system's scope and restrict its applicability. Therefore, techniques are needed to improve the replay speed of MRR approaches.

DeLorean and FDR can replay a program at production run speed. In DeLorean, this is achieved by replaying chunks in parallel and re-synchronizing them according to the same commit order as recorded during their original execution. With FDR, threads are replayed in parallel and are only re-synchronized at the locations corresponding to the point-to-point dependencies recorded during their original execution. Neither FDR nor DeLorean, however, is a likely choice for a practical MRR implementation today. As discussed previously, the complexity of FDR is a major showstopper in modern CMPs. For DeLorean, the execution environment it assumes is not standard in today's multi-processors.

Replaying episodes in Rerun is done sequentially, following increasing timestamp order. As such, Rerun cannot therefore meet the replay speed requirement of DMR usage models, such as fault-tolerance. As an alternative to Rerun, the authors in [30] have proposed a chunk-based replay scheme called a concurrent chunk region. A concurrent chunk region defines a set of chunks that can be replayed in parallel, because each chunk in such a region features the same timestamp as the chunk in other regions. To build such concurrent chunk regions, whenever a chunk must terminate due to a conflict, for instance, all chunks with similar timestamps must also be terminated simultaneously. Therefore, concurrent chunk regions trade off replay speed for log size. The authors in [30] have shown that, by using concurrent chunk regions, replay speed can be improved by several orders of magnitude at the cost of moderate log size increases.

"A concurrent chunk region defines a set of chunks that can be replayed in parallel."

Conclusions

In this article we presented a comprehensive survey of DMR techniques to deal with multi-threaded program execution on CMP machines. We showed that software-only implementations of DMR are quite effective in recording and replaying concurrent programs, but they suffer from performance limitations that can restrict their applicability. To improve on performance, the memory non-determinism of multi-threaded programs must be recorded more efficiently. We described the hardware support needed to deal with fine-grained logging of memory interleavings more efficiently, using either point-to-point approaches or chunk-based approaches. Combined with software approaches, these hardware techniques can provide better performance and address a wider range of usage models. However, there are still several remaining challenges that need to be met before a complete solution can be deployed on real hardware. One such challenge involves recording memory non-determinism with non-sequentially consistent memory models. We hope that the discussions presented here help foster the research on DMR and that they stimulate a broader interest in DMR usage models.

"To improve on performance, the memory non-determinism of multi-threaded programs must be recorded more efficiently.."

References

[1] H. Agrawal. "Towards automatic debugging of computer programs." PhD thesis, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, 1991.

[2] G. Altekar and I. Stoica. "ODR: Output-deterministic replay for multicore debugging." In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 193-206, 2009.

[3] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinic, D. Mihocka, and J. Chau. "Framework for instruction-level tracing and analysis of program executions." In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 154-163, 2006.

[4] B. Boothe. "A fully capable bidirectional debugger." *ACM SIGSOFT Software Engineering Notes*, 25(1), pages 36-37, 2000.

[5] T. C. Bressoud and F. B. Schneider. "Hypervisor-based fault tolerance." In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 1-11, 2009.

[6] K. Buchacker and V. Sieh. "Framework for testing the fault-tolerance of systems including OS and network aspects." In *Proceedings of the 6th IEEE International Symposium on High-Assurance Systems Engineering: Special Topic: Impact of Networking*, pages 95-105, 2001.

[7] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. "BulkSC: Bulk enforcement of sequential consistency." *ACM SIGARCH Computer Architecture News*, 35(2), pages 278-289, 2007.

[8] J. Choi and H. Srinivasan. "Deterministic replay of Java multithreaded applications." In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48-59, 1998.

[9] J. Chow, T. Garfinkel, and P. M. Chen. "Decoupling dynamic program analysis from execution in virtual environments." In *Proceedings of the USENIX Annual Technical Conference*, pages 1–14, 2008.

[10] P. Courtois, F. Heymans, and D. Parnas. "Concurrent control with readers and writers." *Communications of the ACM*, 14(10), pages 667-668, 1971.

[11] G. Dunlap. "Execution replay for intrusion analysis." PhD thesis, EECS Department, University of Michigan, Ann Arbor, Michigan, 2006. Available at ***http://www.eecs.umich.edu/~pmchen/papers/dunlap06.pdf***

[12] S. King, G. Dunlap, and P. Chen. "Operating system support for virtual machines." In *Proceedings of the 2003 USENIX Technical Conference*, pages 71-84, 2003.

[13] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. "ReVirt: Enabling intrusion analysis through virtual-machine logging and replay." *ACM SIGOPS Operating Systems Review*, 36(SI), pages 211-224, 2002.

[14] G. Dunlap, D. Lucchetti, M. Fetterman, and P. Chen. "Execution replay of multiprocessor virtual machines." In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 121-130, 2008.

[15] S. Feldman and C. Brown. "IGOR: a system for program debugging via reversible execution." In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 112-123, 1988.

[16] D. Geels, G. Altekar, S. Shenker, and I. Stoica. "Replay debugging for distributed applications." In *Proceedings of the USENIX '06 Annual Technical Conference*, 27, 2006.

[17] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, F. Kaashoek, and Z. Zhang. "R2: An application-level kernel for record and replay." In *Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation*, pages 193-208, 2008.

[18] D. Hower and M. Hill. "Rerun: Exploiting episodes for lightweight memory race recording." *ACM SIGARCH Computer Architecture News*, 36(3), pages 265-276, 2008.

[19] A. Joshi, S. King, G. Dunlap, and P. Chen. "Detecting past and present intrusions through vulnerability-specific predicates." In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, pages 91-104, 2005.

[20] S. King, G. W. Dunlap, and P. M. Chen. "Debugging operating systems with time-traveling virtual machines." In *Proceedings of the USENIX Annual Technical Conference*, 1, 2005.

[21] L. Lamport. "Time, clocks and the ordering of events in a distributed system." *CACM*, 21(7):558–565, 1978.

[22] J. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[23] T. LeBlanc and J. Mellor-Crummey. "Debugging parallel programs with instant replay." *IEEE Transactions on Computers*, 36(4), pages 471-482, 1987.

[24] P. Montesinos, L. Ceze, and J. Torrellas. "DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently." In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 289-300, 2008.

[25] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. "Capo: Abstractions and software-hardware interface for hardware-assisted deterministic multiprocessor replay." In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 73-84, 2009.

[26] S. Narayanasamy, G. Pokam, and B. Calder. "Bugnet: Continuously recording program execution for deterministic replay debugging." In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 284-295, 2005.

[27] M. Olszewski, J. Ansel, and S. Amarasinghe. "Kendo: Efficient deterministic multithreading in software." In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 97-108, 2009.

[28] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. Lee, and S. Lu. "PRES: Probabilistic replay with execution sketching on multiprocessors." In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 177-192, 2009.

[29] C. Pereira. "Reproducible user-level simulation of multi-threaded workloads." PhD thesis, Department of Computer Science and Engineering, University of California – San Diego, San Diego, California, 2007. Available at ***http://cseweb.ucsd.edu/~calder/papers/thesis-cristiano.pdf***.

[30] G. Pokam, C. Pereira, K. Danne, R. Kassa, and A. Adl-Tabatabai. "Architecting a chunk-based memory race recorder in modern CMPs." In *Proceedings of the 42nd International Symposium on Microarchitecture*, 2009.

[31] M. Prvulovic. "CORD: Cost-effective (and nearly overhead-free) order-recording and data race detection." In *Proceedings of the Twelfth International Symposium on High-Performance Computer Architecture*, 2006.

[32] M. Ronsse and K. De Bosschere. "RecPlay: a fully integrated practical record/replay system." *ACM Transactions on Computer Systems*, 17(2), pages 133–152, 1999.

[33] M. Russinovich and B. Cogswell. "Replay for concurrent non-deterministic shared-memory applications." In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming language Design and Implementation*, pages 258-266, 1996.

[34] Y. Saito. "Jockey: A user-space library for record-replay debugging." In *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging*, pages 69-76, 2005.

[35] J. Slye and E. Elnozahy. "Supporting nondeterministic execution in fault-tolerant systems." In *Proceedings of the Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing*, pages 250, 1996.

[36] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. "Flashback: A lightweight extension for rollback and deterministic replay for software debugging." In *Proceedings of the USENIX Annual Technical Conference*, 3, 2004.

[37] M. Xu, R. Bodik, and M. Hill. "A flight data recorder for enabling full-system multiprocessor deterministic replay." In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 122-135, 2003.

[38] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, B. Weissman, and VMWare Inc. "Retrace: Collecting execution trace with virtual machine deterministic replay." In *Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation*, 2007.

[39] M. Zelkowitz. "Reversible execution." *Communications of the ACM*, 16(9):566, 1973.

Authors' Biographies

Gilles Pokam is a Senior Research Scientist in the Microprocessor & Programming Research at Intel Labs. His research interests are in multi-core architectures and software, with a current focus on concurrent programming and programmer productivity. Before joining Intel Labs, he was a researcher at IBM T.J. Watson Research Center in NY, and a postdoctoral research scientist at the University of California, San Diego. He received a PhD degree from INRIA Lab and the University of Rennes I, in France. He is the recipient of the IEEE MICRO Top Picks Award 2006 that recognizes the most significant papers in computer architecture. Gilles is a member of IEEE and ACM. His e-mail is gilles.a.pokam at intel.com.

Cristiano Pereira is an Engineer in the Technology Pathfinding and Innovation team, at Intel's Software and Services Group. His research interests are in the areas of hardware support for better programmability of multi-core architectures and software tools to improve programmer's productivity. He received a PhD degree from the University of California, San Diego, in 2007 and a Masters degree from the Federal University of Minas Gerais, Brazil, in 2000. Prior to that, Cristiano worked for a number of small companies in Brazil. He is a member of IEEE. His e-mail is cristiano.l.pereira at intel.com.

Klaus Danne is an Engineer in the Microprocessor & Programming Research Group at Intel Labs. His research interests are in multi-core architectures, deterministic replay, design emulation, and reconfigurable computing systems. He received a PhD degree and Masters degree from the University of Paderborn Germany in 2006 and 2002, respectively. His e-mail is klaus.danne at intel.com.

Lynda Yang is a graduate student in Computer Science at the University of Illinois at Urbana-Champaign. Her research interests are in multi-processor architectures and operating systems. She received a BS degree in Computer Science in 2008 from the University of North Carolina at Chapel Hill. Her e-mail is yang61 at illinois.edu.

Samuel T. King is an Assistant Professor in the Computer Science Department at the University of Illinois. His research interests include security, experimental software systems, operating systems, and computer architecture. His current research focuses on defending against malicious hardware, deterministic replay, designing and implementing secure web browsers, and applying machine learning to systems problems. Sam received his PhD degree in Computer Science and Engineering from the University of Michigan in 2006.

Josep Torrellas is a Professor of Computer Science and Willett Faculty Scholar at the University of Illinois, Urbana-Champaign. He received a PhD degree from Stanford University in 1992. His research area is multi-processor computer architecture. He has participated in the Stanford DASH and the Illinois Cedar experimental multi-processor projects, and in several DARPA initiatives in novel computer architectures. Currently, he leads the Bulk Multi-core Architecture project for programmability in collaboration with Intel. He has published over 150 papers in computer architecture and received several best-paper awards. He has graduated 27 PhD students, many of whom are now leaders in academia and industry. He is an IEEE Fellow.

Copyright

A PROGRAMMING MODEL FOR HETEROGENEOUS INTEL® X86 PLATFORMS

Contributors

Bratin Saha
Intel Corporation

Xiaocheng Zhou
Intel Corporation

Hu Chen
Intel Corporation

Ying Gao
Intel Corporation

Shoumeng Yan
Intel Corporation

Sai Luo
Intel Corporation

Index Words

Heterogeneous Platforms
Programming Model
Larrabee
Shared Virtual Memory
Runtime
GPGPU

Abstract

The client computing platform is moving towards a heterogeneous architecture that consists of a combination of cores focused on scalar performance, and of a set of throughput-oriented cores. The throughput-oriented cores (such as those in the Intel® microarchitecture codename Larrabee processor) may be connected over both coherent and non-coherent interconnects, and they may have different instruction set architectures (ISAs). This article describes a programming model for such heterogeneous platforms. We discuss the language constructs, runtime implementation, and the memory model for such a programming environment. We implemented this programming environment in an Intel® x86 heterogeneous platform simulator and we ported a number of workloads to our programming environment. We present the performance of our programming environment on these workloads.

Introduction

Client computing platforms are moving towards a heterogeneous architecture with some processing cores focused on scalar performance and other cores focused on throughput performance. For example, desktop and notebook platforms may ship with one or more central processing units (CPUs), primarily focused on scalar performance, along with a graphics processing unit (GPU) that can be used for accelerating highly-parallel data kernels. These heterogeneous platforms can be used to provide a significant performance boost on highly-parallel non-graphics workloads in image processing, medical imaging, data mining [6], and other domains [10]. Several vendors have also come out with programming environments for such platforms, such as CUDA [11], CTM [2], and OpenCL [12].

Heterogeneous platforms have a number of unique architectural constraints:

- The throughput-oriented cores (e.g., GPU) may be connected in both integrated and discrete forms. A system may also have a hybrid configuration where a low-power, lower-performance GPU is integrated with the chipset, and a higher-performance GPU is attached as a discrete device. Finally, a platform may also have multiple discrete GPU cards. The platform configuration determines many parameters, such as bandwidth and latency between the different kinds of processors, the cache coherency support, etc.

- The scalar and throughput-oriented cores may have different operating systems. For example, Intel's upcoming processor, codename Larrabee [16], can have its own kernel. (The Larrabee processor is a general-throughput computing device that includes a software stack for high-performance graphics rendering.) This means that the virtual memory translation schemes (virtual to physical address translation) can be different between the different kinds of cores. The same virtual address can be simultaneously mapped to two different physical addresses — one residing in CPU memory and the other residing in the Larrabee processor memory. This also means that the system environment (loaders, linkers, etc.) can be different between the two cores. For example, the loader may load the application at different base addresses on different cores.
- The scalar and throughput-oriented cores may have different ISAs, and hence the same code cannot be run on both kinds of cores.

"The Larrabee processor is a general-throughput computing device that includes a software stack for high-performance graphics rendering."

A programming model for heterogeneous platforms must address all of the aforementioned architectural constraints. Unfortunately, existing programming models such as Nvidia Compute Unified Device Architecture (CUDA) and ATI Close To the Metal (CTM) address only the ISA heterogeneity, by providing language annotations to mark code that must run on GPUs; they do not take other constraints into account. For example, CUDA does not address the memory management issues between CPU and GPU. It assumes that the CPU and GPU are separate address spaces and that the programmer uses separate memory allocation functions for the CPU and the GPU. Further, the programmer must explicitly serialize data structures, decide on the sharing protocol, and transfer the data back and forth.

"CUDA does not address the memory management issues between CPU and GPU."

In this article, we propose a new programming model for heterogeneous Intel® x86 platforms that addresses all the issues just mentioned. First, we propose a uniform programming model for different platform configurations. Second, we propose using a shared memory model for all the cores in the platform (e.g., between the CPU and the Larrabee cores). Instead of sharing the entire virtual address space, we propose that only a part of the virtual address space be shared to enable an efficient implementation. Finally, like conventional programming models, we use language annotations to demarcate code that must run on the different cores, but we improve upon conventional models by extending our language support to include features such as function pointers.

We break from existing CPU-GPU programming models and propose a shared memory model, since a shared memory model opens up a completely new programming paradigm that improves overall platform performance. A shared memory model allows pointers and data structures to be seamlessly shared between the different cores (e.g., CPU and Larrabee cores) without requiring any marshalling. For example, consider a game engine that includes physics, artificial intelligence (AI), and rendering. A shared memory model allows the physics, AI, and game logic to be run on the scalar cores (e.g., CPU), while

"We break from existing CPU-GPU programming models and propose a shared memory model."

the rendering runs on the throughput cores (e.g., Larrabee core), with both the scalar and throughput cores sharing the scene graph. Such an execution model is not possible in current programming environments, since the scene graph would have to be serialized back and forth.

"The scene graph and associated structures are placed in shared memory and used concurrently by all the cores in the platform."

We implemented our full programming environment, including the language and runtime support, and ported a number of highly parallel non-graphics workloads to this environment. In addition, we ported a full gaming application to our system. By using existing models, we spent one and a half weeks coding this application for data management for each new game feature: the game itself included about a dozen features. The serialization arises since the rendering is performed on the Larrabee processor, while the physics and game logic are executed on the CPU. All of this coding (and the associated debugging, etc.) is unnecessary in our system, since the scene graph and associated structures are placed in shared memory and used concurrently by all the cores in the platform. Our implementation works with different operating system kernels running on the scalar and throughput-oriented cores.

We ported our programming environment to a heterogeneous Intel x86 platform simulator that simulates a set of Larrabee, throughput-oriented cores attached as a discrete PCI-Express device to the CPU. We used such a platform for two reasons. First, we believe the Larrabee core is more representative of how GPUs are going to evolve into throughput-oriented cores. Second, the platform poses greater challenges, due to the heterogeneity in the system software stack, as opposed to simply ISA heterogeneity. Later in this article, we present performance results on a variety of workloads.

To summarize, in this article, we discuss the design and implementation of a new programming model for heterogeneous Intel x86 platforms. We make the following contributions:

- Provide shared memory semantics between the CPU and the Larrabee processor by allowing pointers and data structures to be shared seamlessly. This extends previous work in the areas of distributed shared memory (DSM) and partitioned global address space (PGAS) languages by providing shared memory semantics in a platform with heterogeneous ISA, operating system kernels, etc. We also improve application performance by allowing user-level communication between the CPU and the Larrabee core.
- Provide a uniform programming model for different platform configurations.

"We also improve application performance by allowing user-level communication between the CPU and the Larrabee core."

In the remainder of this article, we first provide a brief overview of the Larrabee architecture: then we discuss the proposed memory model, and we describe the language constructs for programming this platform. We then describe our prototype implementation, and finally we present performance numbers.

Larrabee Architecture

The Larrabee architecture is a many-core x86 visual computing architecture that is based on in-order cores that run an extended version of the x86 instruction set, including wide-vector processing instructions and some specialized scalar instructions. Each of the cores contains a 32 KB instruction cache and a 32 KB L1 data cache, and each core accesses its own subset of a coherent L2 cache to provide high-bandwidth L2 cache access. The L2 cache subset is 256 KB, and the subsets are connected by a high-bandwidth, on-die ring interconnect. Data written by a CPU core are stored in their own L2 cache subset and are flushed from other subsets, if necessary. Each ring data path is 512 bits wide in each direction. The fixed function units and memory controller are spread across the ring to reduce congestion.

Each core has four hardware threads with separate register sets for each thread. Instruction issue alternates between the threads and it covers cases where the compiler is unable to schedule code without stalls. The core uses a dual-issue decoder, and the pairing rules for the primary and secondary instruction pipes are deterministic. All instructions can issue on the primary pipe, while the secondary pipe supports a large subset of the scalar x86 instruction set, including loads, stores, simple ALU operations, vector stores, etc. The core supports 64-bit extensions and the full Intel Pentium® processor x86 instruction set. The Larrabee architecture consists of a 16-wide vector processing unit that executes integer, single precision float, and double precision float instructions. The vector unit supports gather-scatter and masked instructions, and it supports instructions with up to three source operands.

"Each core has four hardware threads with separate register sets for each thread."

Memory Model

The memory model for our system provides a window of shared addresses between the CPU and Larrabee cores, such as in PGAS [17] languages, but enhances it with additional ownership annotations. Any data structure that is shared between the CPU and Larrabee core must be allocated by the programmer in this space. The system provides a special malloc function that allocates data in this space. Static variables can be annotated with a type qualifier so that they are allocated in the shared window. However, unlike PGAS languages, there is no notion of affinity in the shared window. This is because data in the shared space must migrate between the CPU and Larrabee caches as they get used by each processor. Also, the representation of pointers does not change between the shared and private spaces.

"The memory model for our system provides a window of shared addresses between the CPU and Larrabee cores."

The remaining virtual address space is private to the CPU and Larrabee processor. By default, data get allocated in this space, and they are not visible to the other side. We choose this partitioned address space approach since it cuts down on the amount of memory that needs to be kept coherent, and it enables a more efficient implementation for discrete devices.

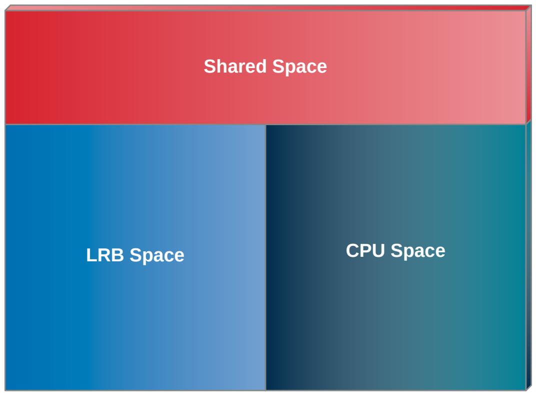


Figure 1: CPU-larrabee Processor Memory Model
Source: Intel Corporation, 2009

The proposed memory model can be extended in a straightforward way to multiple Larrabee processor configurations. The window of shared virtual addresses extends across all the devices. Any data structures allocated in this shared address window are visible to all agents, and pointers in this space can be freely exchanged between the devices. In addition, every agent has its own private memory as shown in Figure 1.

We propose using release consistency in the shared address space for several reasons. First, the system only needs to remember all the writes between successive release points, not the sequence of individual writes. This makes it easier to do bulk transfers at release points (e.g., several pages at a time). This is especially important in the discrete configuration, since it is more efficient to transfer bulk data over PCI-Express. Second, release consistency allows memory updates to be kept completely local until a release point, which is again important in a discrete configuration. In general, the release consistency model is a good match for the programming patterns in CPU-GPU platforms, since there are natural release and acquisition points in such programs. For example, a call from the CPU into the GPU is one such point. Making any of the CPU updates visible to the GPU before the call does not serve any purpose, and neither does it serve any purpose to enforce any order on how the CPU updates become visible, as long as all of them are visible before the GPU starts executing. Finally, the proposed C/C++ memory model [5] can be mapped easily to our shared memory space.

"The proposed memory model can be extended in a straightforward way to multiple Larrabee processor configurations."

We augment our shared memory model with ownership rights to enable further coherence optimizations. Within the shared virtual address window, the CPU or Larrabee processor can specify at a particular point in time that it owns a specific chunk of addresses. If an address range in the shared window is owned by the CPU, then the CPU knows that the Larrabee processor cannot access those addresses and hence does not need to maintain coherence of those addresses with the Larrabee processor: for example, it can avoid sending any snoops or other coherence information to the Larrabee processor. The same is true of addresses owned by the Larrabee processor. If a CPU-owned address is accessed by the Larrabee processor, then the address becomes un-owned (with symmetrical behavior for those addresses owned by the Larrabee processor). We provide these ownership rights to leverage common usage models. For example, the CPU first accesses some data (e.g., initializing a data structure), and then hands them over to the Larrabee processor (e.g., computing on the data structure in a data parallel manner), and then the CPU analyzes the results of the computation and so on. The ownership rights allow an application to inform the system of this temporal locality and to optimize the coherence implementation. Note that these ownership rights are optimization hints only, and it is legal for the system to ignore these hints.

"The ownership rights allow an application to inform the system of this temporal locality and to optimize the coherence implementation."

Language Constructs

To deal with platform heterogeneity, we add constructs to C that allow the programmer to specify whether a particular data item should be shared or private, and to specify whether a particular code chunk should be run on the CPU or on the Larrabee processor.

The first construct is the shared type qualifier, similar to UPC [17], which specifies a variable that is shared between the CPU and Larrabee processor. The qualifier can also be associated with pointer types to imply that the target of the pointer is in shared space. For example, one can write:

```
shared int var1;            // int is in shared space

int var2;                   // int is not in shared space

shared int* ptr1;           // ptr1 points to a shared location

int* ptr2;                  // ptr2 points to private space

shared int *shared ptr1;    // ptr1 points to shared and is shared
```

"The actual virtual address range in each space is decided by the system and is transparent to the user."

The compiler allocates globally shared variables in the shared memory space, while the system provides a special malloc function to allocate data in the shared memory. The actual virtual address range in each space is decided by the system and is transparent to the user.

It is legal for a language implementation to allocate all data in the shared space — that is, map all malloc calls to the sharedMalloc and allocate all globals in the shared space. A programmer then deals only with shared data. The key point is that our system provides the hooks needed for programmers to demarcate private and shared data, should they want to do that.

We use an attribute, __attribute(Larrabee), to mark functions that should be executed on the Larrabee processor. For such functions, the compiler generates code that is specific to the Larrabee processor. When a non-annotated function calls a Larrabee annotated function, it implies a call from the CPU to the Larrabee processor. The compiler checks that all pointer arguments have shared type and invokes a runtime API for the remote call. Function pointer types are also annotated with the attribute notation, implying that they point to functions that are executed on the Larrabee processor. Non-annotated function pointer types point to functions that execute on the CPU. The compiler checks type equivalence during an assignment; for example, a function pointer with the Larrabee attribute must always be assigned the address of a function that is annotated for the Larrabee processor.

"We use an attribute, __ attribute(Larrabee), to mark functions that should be executed on the Larrabee processor."

"The wrapper attribute allows the programmer to stop the porting effort at any point in the call tree by calling back into the CPU."

Our third construct denotes functions that execute on the CPU but can be called from the Larrabee processor. These functions are denoted by using __attribute(wrapper). We used this function in two ways. First, many programs link with precompiled libraries that can execute on the CPU. The functions in these libraries are marked as wrapper calls so that they execute on the CPU if called from Larrabee code. Second, while porting large programs from a CPU-only execution mode to a CPU plus Larrabee processor mode, it is very helpful to incrementally port the program. The wrapper attribute allows the programmer to stop the porting effort at any point in the call tree by calling back into the CPU. When a Larrabee function calls a wrapper function, the compiler invokes a runtime API for the remote call from the Larrabee processor to the CPU. Making Larrabee-to-CPU calls explicit allows the compiler to check that any pointer arguments have the shared type.

"We also provide a construct and the corresponding runtime support for making asynchronous calls from the CPU to the Larrabee processor."

We also provide a construct and the corresponding runtime support for making asynchronous calls from the CPU to the Larrabee processor. This allows the CPU to avoid waiting for Larrabee computation to finish. Instead, the runtime system returns a handle that the CPU can query for completion. Since this does not introduce any new design issues, we focus mostly on synchronous calls in the remainder of this article.

Data Annotation Rules

These rules apply to data that can be allocated in the shared virtual space:

- Shared can be used to qualify the type of variables with global storage. Shared cannot be used to qualify a variable with automatic storage unless it qualifies a pointer's referenced type.
- A pointer in private space can point to any space. A pointer in shared space can only point to shared space but not to private space.
- A structure or union type can have the shared qualifier which then requires all fields to have the shared qualifier as well.

The following rules are applied to pointer manipulations:

- Binary operator (+,-,,==,......) is only allowed between two pointers pointing to the same space. The system provides API functions that perform dynamic checks. When an integer expression is added to or subtracted from a pointer, the result has the same type as the pointer.
- Assignment/casting from pointer-to-shared to pointer-to-private is allowed. If a type is not annotated, we assume that it denotes a private object. This makes it difficult to pass shared objects to legacy functions, since their signature requires private objects. The cast allows us to avoid copying between private and shared spaces when passing shared data to a legacy function.
- Assignment/casting from pointer-to-private to pointer-to-shared is allowed only through a dynamic_cast. The dynamic_cast checks at runtime that the pointer-to-shared actually points to shared space. If the check fails, an error is thrown and the user has to explicitly copy the data from private space to shared space. This cast allows legacy code to efficiently return values.

"The cast allows us to avoid copying between private and shared spaces when passing shared data to a legacy function."

Our language can allow casting between the two spaces (with possibly a dynamic check) since our data representation remains the same regardless of whether the data are in shared or private space. Even pointers have the same representation regardless of whether they are pointing to private or shared space. Given any virtual address V in the shared address window, both the CPU and Larrabee processor have their own local physical address corresponding to this virtual address. Pointers on the CPU and Larrabee processor read from this local copy of the address, and the local copies get synced up as required by the memory model.

"Even pointers have the same representation regardless of whether they are pointing to private or shared space."

Code Annotation Rules

These rules apply to the code or function, where they execute.

- A __attribute(Larrabee) function is not allowed to call a non-annotated function. This is to ensure that the compiler knows about all the CPU functions that can be called from the Larrabee processor.
- A __attribute(wrapper) function is not allowed to call into a __ attribute(Larrabee) function. This is primarily an implementation restriction in our system.
- Any pointer parameter of a Larrabee- or wrapper-annotated function must point to shared space.

The calling rules for functions also apply to function pointers. For example, a __attribute(Larrabee) function pointer called from a non-annotated function results in a CPU-to-Larrabee processor call. Similarly, un-annotated function pointers cannot be called from Larrabee functions.

"Acquire and release points follow naturally from the language semantics."

The runtime also provides APIs for mutexes and barriers to allow the application to perform explicit synchronization. These constructs are always allocated in the shared area.

Acquire and release points follow naturally from the language semantics. For example, a call from the CPU to the Larrabee processor is a release point on the CPU followed by an acquire point on the Larrabee processor. Similarly, a return from the Larrabee processor is a release point on the Larrabee and an acquire point on the CPU. Taking ownership of a mutex and releasing a mutex are acquire and release points, respectively, for the processor doing the mutex operation, while hitting a barrier and getting past a barrier are release and acquire points as well.

Ownership Constructs

"We also provide an API that allows the user to allocate chunks of memory (hereafter called arenas) inside the shared virtual region."

We also provide an API that allows the user to allocate chunks of memory (hereafter called arenas) inside the shared virtual region. The programmer can dynamically set the ownership attributes of an arena. Acquiring ownership of an arena also acts as an acquire operation on the arena, and releasing ownership of an arena acts as the release operation on the arena. The programmer can allocate space within an arena by passing the arena as an argument to a special malloc function. The runtime grows the arena as needed. The runtime API is shown here:

```
// Owned by the caller or shared
Arena *allocateArena(OwnershipType type);
//Allocate and free within arena
shared void *arenaMalloc ( Arena*, size_t);
void arenaFree( Arena *, shared void *);
// Ownership for arena. If null changes ownership of entire shared area
OwnershipType acquireOwnership(Arena*);
OwnershipType releaseOwnership(Arena*);
//Consistency for arena
void arenaAcquire(Arena *);
void arenaRelease(Arena *);
```

"This prevents coherence messages from being exchanged between the CPU and Larrabee processor while the frames are being processed."

The programmer can optimize the coherence implementation by using the ownership API. For example, in a gaming application, while the CPU is generating a frame, the Larrabee processor may be rendering the previous frame. To leverage this pattern, the programmer can allocate two arenas, with the CPU acquiring ownership of one arena and generating the frame into that arena, while the Larrabee processor acquires ownership of the other arena and renders the frame in that arena. This prevents coherence messages from being exchanged between the CPU and Larrabee processor while the frames are being processed. When the CPU and Larrabee processor are finished with their current frames, they exchange ownership of their arenas, so that they continue to work without incurring coherence overhead.

Implementation

"The compiler generates two binaries – one for execution on the Larrabee processor and another for CPU execution."

The compiler generates two binaries – one for execution on the Larrabee processor and another for CPU execution. We generate two different executables since the two operating systems can have different executable formats. The Larrabee binary contains the code that will execute on the Larrabee processor (annotated with the Larrabee attribute), while the CPU binary contains the CPU functions which include all un-annotated and wrapper-annotated functions. Our runtime library has a CPU and Larrabee component that are linked with the CPU and Larrabee application binaries to create the CPU and Larrabee executables. When the CPU binary starts executing, it calls a runtime function that loads the Larrabee executable. Both the CPU and Larrabee binaries create a daemon thread that is used for the communication.

Implementing Shared Memory between the CPU and Larrabee Processor

Our implementation reuses some ideas from software distributed shared memory [9, 4] schemes, but there are some significant differences as well. Unlike DSMs, our implementation is complicated by the fact that the CPU and Larrabee processor can have different page tables and different virtual-to-physical memory translations. Thus, when we want to sync up the contents of virtual address V between the CPU and Larrabee processor (e.g., at a release point), we may need to sync up the contents of different physical addresses, such as P1 on CPU and P2 on the Larrabee processor. Unfortunately, the CPU does not have access to the Larrabee processor's page tables (hence the CPU has no knowledge of P2), and the Larrabee processor cannot access the CPU's page tables and has no knowledge of P1.

We solve the aforementioned problem by leveraging the PCI aperture in a novel way. During initialization we map a portion of the PCI aperture space into the user space of the application and instantiate it with a task queue, a message queue, and copy buffers. When we need to copy pages, for example, from the CPU to the Larrabee processor, the runtime copies the pages into the PCI aperture copy buffers and tags the buffers with the virtual address and the process identifier. On the Larrabee side, the daemon thread copies the contents of the buffers into its address space by using the virtual address tag. Thus, we perform the copy in a two-step process—the CPU copies from its address space into a common buffer (PCI aperture) that both the CPU and Larrabee processor can access, while the Larrabee processor picks up the pages from the common buffer into its address space. Copies from the Larrabee processor to the CPU are done in a similar way. Note that since the aperture is pinned memory, the contents of the aperture are not lost if the CPU or Larrabee processor gets the context switched out. This allows the two processors to execute asynchronously, which is critical, since the two processors can have different operating systems and hence the context switches cannot be synchronized. Finally, note that we map the aperture space into the user space of the application, thus enabling user-level communication between the CPU and Larrabee processor. This makes the application stack vastly more efficient than going through the operating system driver stack. To ensure security, the aperture space is partitioned among the CPU processes that want to use Larrabee processor. At present, a maximum of eight processes can use the aperture space.

"During initialization we map a portion of the PCI aperture space into the user space of the application and instantiate it with a task queue, a message queue, and copy buffers."

"This allows the two processors to execute asynchronously."

"We put a directory in the PCI aperture that contains metadata about the pages in the shared address region."

We exploit one other difference between traditional software DSMs and CPU plus Larrabee processor platforms. Traditional DSMs were designed to scale on medium-to-large clusters. In contrast, CPU plus Larrabee processor systems are very small-scale clusters. We expect that no more than a handful of Larrabee cards and CPU sockets will be used well into the future. Moreover, the PCI aperture provides a convenient shared physical memory space between the different processors. Thus, we are able to centralize many data structures and make the implementation more efficient. For example, we put a directory in the PCI aperture that contains metadata about the pages in the shared address region. The metadata states whether the CPU or Larrabee processor holds the golden copy of a page (home for the page), contains a version number that tracks the number of updates to the page, mutexes that are acquired before updating the page, and other miscellaneous metadata. The directory is indexed by the virtual address of a page. Both the CPU and the Larrabee runtime systems maintain a private structure that contains local access permissions for the pages and the local version numbers of the pages.

"Between acquire and release points, the Larrabee processor and CPU operate out of their local memory and communicate with each other only at the explicit synchronization points."

When the Larrabee processor performs an acquire operation, the corresponding pages are set to no-access on the Larrabee processor. At a subsequent read operation, the page fault handler on the Larrabee processor copies the page from the home location, if the page has been updated and released since the last Larrabee acquire. The directory and private version numbers are used to determine this. The page is then set to read-only. At a subsequent write operation, the page fault handler creates the backup copy of the page, marks the page as read-write, and increments the local version number of the page. At a release point, we perform a *diff* with the backup copy of the page and transmit the changes to the home location, while incrementing the directory version number. The CPU operations are symmetrical. Thus, between acquire and release points, the Larrabee processor and CPU operate out of their local memory and communicate with each other only at the explicit synchronization points.

At startup the implementation decides the address range that will be shared between the CPU and Larrabee processor, and it ensures that this address range always remains mapped. This address range can grow dynamically and does not have to be contiguous; although in a 64-bit address space, the runtime system can reserve a continuous chunk upfront.

Implementing Shared Memory Ownership

Every arena has associated metadata that identify the pages that belong to the arena. Suppose the Larrabee processor acquires ownership of an arena, we then make the corresponding pages non-accessible on the CPU. We copy from the home location any arena pages that have been updated and released since the last time the Larrabee processor performed an acquire operation. We set the pages to read-only so that subsequent Larrabee writes will trigger a fault, and the system can record which Larrabee pages are being updated. In the directory, we note that the Larrabee processor is the home node for the arena pages. On a release operation, we simply make the pages accessible again on the other side and update the directory version number of the pages. The CPU ownership operations are symmetrical.

"Every arena has associated metadata that identify the pages that belong to the arena."

Note the performance advantages of acquiring ownership. At a release point we no longer need to perform diff operations, and we do not need to create a backup copy at a write fault, since we know that the other side is not updating the page. Second, since the user provides specific arenas to be handed over from one side to the other, the implementation can perform a bulk copy of the required pages at an acquire point. This leads to a more efficient copy operation, since the setup cost is incurred only once and gets amortized over a larger copy.

"Note the performance advantages of acquiring ownership."

Implementing Remote Calls

A remote call between CPU and Larrabee cores is complicated by the fact that the two processors can have different system environments: for example, they can have different loaders. Larrabee and CPU binary code is also loaded separately and asynchronously. Suppose that the CPU code makes some calls into the Larrabee processor when the CPU binary code is loaded, the Larrabee binary code has still not been loaded and hence the addresses for Larrabee functions are still not known. Therefore, the operating system loader cannot patch up the references to Larrabee functions in the CPU binary code. Similarly, when the Larrabee binary code is being loaded, the Larrabee loader does not know the addresses of any CPU functions being called from Larrabee code and hence cannot patch those addresses.

"A remote call between CPU and Larrabee cores is complicated by the fact that the two processors can have different system environments."

We implement remote calls by using a combination of compiler and runtime techniques. Our language rules ensure that any function involved in remote calls (Larrabee or wrapper attribute functions) is annotated by the user. When compiling such functions, the compiler adds a call to a runtime API that registers function addresses dynamically. The compiler creates an initialization function for each file that invokes all the different registration calls. When the binary code gets loaded, the initialization function in each file gets called. The shared address space contains a jump table that is populated dynamically by the registration function. The table contains one slot for every annotated function. The format of every slot is <funcName, funcAddr> where funcName is a literal string of the function name, and funcAddr is the runtime address of the function.

The translation scheme works as follows.

"We implement remote calls by using a combination of compiler and runtime techniques."

- If a Larrabee (CPU) function is being called within a Larrabee (CPU) function, the generated code will do the call as is.
- If a Larrabee function is being called within a CPU function, the compiler-generated code will do a remote call to the Larrabee processor:
  - The compiler-generated code will look up the jump table with the function name and obtain the function address.
  - The generated code will pack the arguments into an argument buffer in shared space. It will then call a dispatch routine on the Larrabee side passing in the function address and the argument buffer address.

There is a similar process for a wrapper function: if a wrapper function is called in a Larrabee code, a remote call is made to the CPU.

"If a Larrabee function pointer is being called within a CPU function, the compiler-generated code will do a remote call to the Larrabee side."

For function pointer invocations, the translation scheme works as follows. When a function pointer with Larrabee annotation is assigned, the compiler-generated code will look up the jump table with the function name and assign the function pointer with obtained function address. Although the lookup can be optimized when a Larrabee-annotated function pointer is assigned within Larrabee code, we forsake such optimization to use a single strategy for all function pointer assignments. If a Larrabee function pointer is being called within a Larrabee function, the compiler-generated code will do the call as is. If a Larrabee function pointer is being called within a CPU function, the compiler-generated code will do a remote call to the Larrabee side. The process is similar for a wrapper function pointer: if a wrapper function pointer is called in a Larrabee function, a remote call is made to the CPU side.

The signaling between CPU and Larrabee processor happens with task queues in the PCI aperture space. The daemon threads on both sides poll their respective task queues and when they find an entry in the task queue, they spawn a new thread to invoke the corresponding function. The API for remote invocations is described in this code:

```
// Synchronous and asynchronous remote calls
RPCHandler callRemote (FunctionType, RPCArgType);
int resultReady (RPCHandler);
Type getResult (RPCHandler);
```

"The signaling between CPU and Larrabee processor happens with task queues in the PCI aperture space."

Finally, the CPU and Larrabee processor cooperate while allocating memory in the shared area. Each processor allocates memory from either side of the shared address window. When one processor consumes half of the space, the two processors repartition the available space.

Experimental Evaluation

We used a heterogeneous platform simulator for measuring the performance of different workloads on our programming environment. This platform simulates a modern out-of-order CPU and a Larrabee system. The CPU simulation uses a memory and architecture configuration similar to that of the Intel® Core™2 Duo processor. The Larrabee system was simulated as a discrete PCI-Express device with an interconnect latency and bandwidth similar to those of PCI-Express 2.0, and the instruction set was modeled on the Intel Pentium® processor. It did not simulate the new Larrabee instructions and parts of the Larrabee memory hierarchy, such as the ring interconnect. The simulator ran a production-quality software stack on the two processors. The CPU ran Windows* Vista*, while Larrabee processor ran a lightweight operating-system kernel.

We used a number of well-known parallel non-graphics workloads [6] to measure the performance of our system. These include the Black Scholes* financial workload that does option pricing by using the Black Scholes method; the fast fourier transform (FFT) workload that does a radix-2 FFT algorithm used in many domains, such as signal processing; the Equake* workload, part of SpecOMP*, that performs earthquake modeling, and that is representative of high-performance computer (HPC) applications; and Art, also part of SpecOMP, that performs image recognition. The reported numbers are based on using the standard input sets for each of the applications. All these workloads were rewritten by using our programming constructs and were compiled with our tool chain.

"We used a number of well-known parallel non-graphics workloads [6] to measure the performance of our system."

Figure 2 shows the fraction that total memory accesses were to shared data in the aforementioned workloads. The vast majority of the accesses were to private data. Note that read-only data accessed by multiple threads were privatized manually. This manual privatization helped in certain benchmarks like Black Scholes. It is not surprising that most of the accesses are to private data, since the computation threads in the workloads privatize the data that they operate on to get better memory locality. We expect workloads that scale to a large number of cores to behave similarly, since the programmer must be conscious of data locality and avoid false sharing in order to get good performance. The partial virtual address sharing memory model lets us leverage this access pattern by cutting down on the amount of data that need to be kept coherent.

"We expect workloads that scale to a large number of cores to behave similarly."

We next show the performance of our system on the set of workloads. We ran the workloads on a simulated system with 1 CPU core and varied the number of Larrabee cores from 6 to 24. The workload computation was split between the CPU and Larrabee cores, with the compute-intensive portions executed on Larrabee cores. For example, all the option pricing in Black Scholes and the earthquake simulation in Equake is offloaded to Larrabee cores. We present the performance improvement relative to a single CPU and Larrabee core. Figure 3 compares the performance of our system, when the application does not use any ownership calls, to the performance when the user optimizes the application further by using ownership calls. The bars labeled "Mine/Yours" represent the performance with ownership calls: (Mine implies pages were owned by CPU and Yours implies pages were owned by the Larrabee processor). The bars labeled "Ours" represent the performance without any ownership calls.

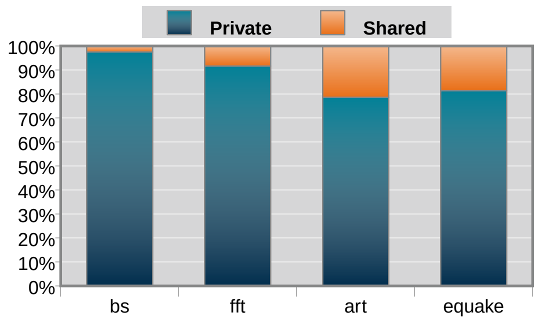


Figure 2: Percent of Shared Data in Memory Accesses
Source: Intel Corporation, 2009

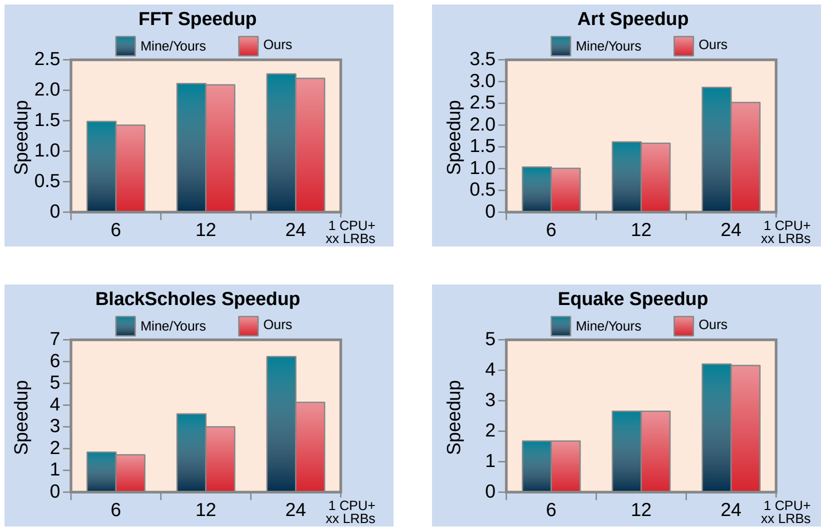


Figure 3: Ownership Performance Comparison
Source: Intel Corporation, 2009

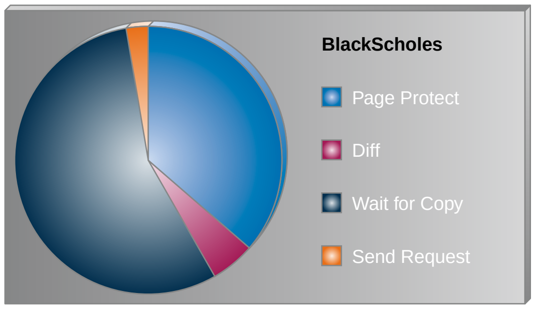


Figure 4: Overhead Breakdown without Ownership
Source: Intel Corporation, 2009

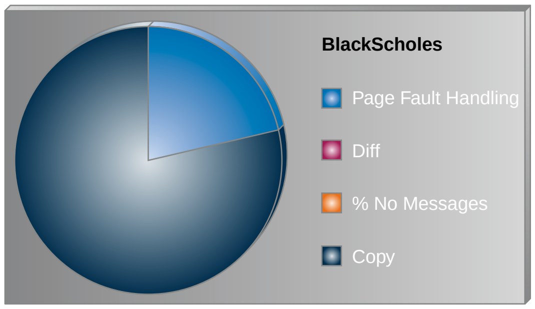


Figure 5: Overhead Breakdown with Ownership
Source: Intel Corporation, 2009

As expected, the applications perform better with ownership calls than without. To understand the reason for this, we broke down the overhead of the system when the application was not using any ownership calls. Figure 4 shows the breakdown for Black Scholes. We show the breakdown for only one benchmark, but the ratios of the different overheads are very similar in all the benchmarks.

We break up the overhead into four categories. The first one relates to handling the page faults, since we use a virtual memory-based shared memory implementation, and reads/writes to a page after an acquire point triggers a fault. The second relates to the diff operation performed at release points to sync up the CPU and Larrabee copies of a page. The third is the amount of time spent in copying data from one side to the other. The copy operation is triggered from the page fault handler when either processor needs the latest copy of a page. We do not include the copy overhead as part of the page fault overhead, but present it separately, since we believe different optimizations can be applied to optimize it.

Finally, the fourth one shows the overhead spent in synchronizing messages. Note that in a discrete setting, the Larrabee processor is connected to the CPU over the PCI-Express. The PCI-Express protocol does not include atomic read-modify-write operations. Therefore we have to perform some synchronization and hand shaking between the CPU and Larrabee processor by passing messages.

When the application uses ownership of arenas, the diff overhead is completely eliminated. The page fault handling is reduced, since the write page-fault handler does not have to create a backup copy of the page. Moreover, since we copy all the pages in one step when we acquire ownership of an arena, we do not incur read page faults.

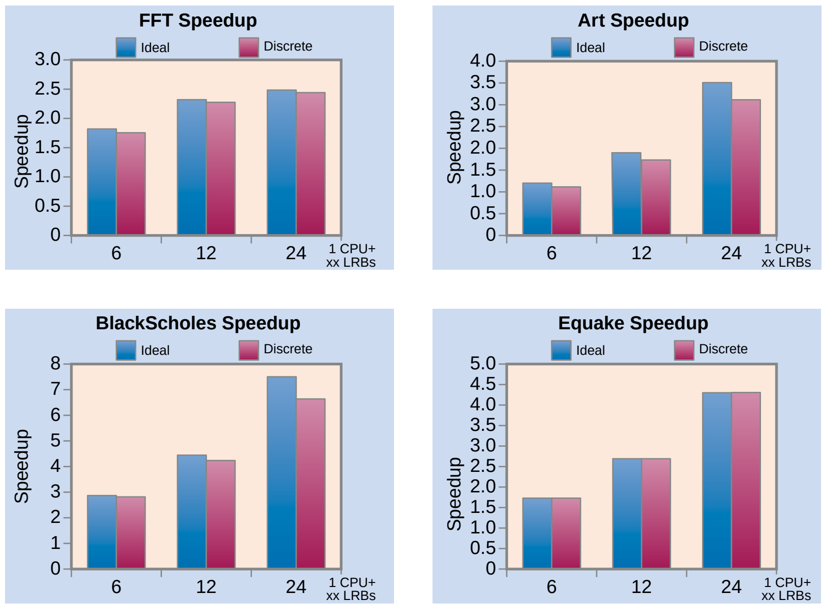


Figure 6: Overall performance comparison
Source: Intel Corporation, 2009

This also significantly reduces the synchronization message overhead since the CPU and Larrabee processor perform the handshaking at only ownership acquisition points rather than at many intermediate points (e.g., whenever pages are transferred from one side to the other). Figure 5 shows the overhead breakdown with ownership calls.

"The page fault handling is reduced, since the write page-fault handler does not have to create a backup copy of the page."

Finally, Figure 6 shows the overall performance of our system. All the workloads used the ownership APIs. The "ideal" bar represents hardware-supported cache coherence between the CPU and Larrabee cores—in other words, this is the best performance that our shared memory implementation can provide. For Equake, since the amount of data transferred is very small compared to the computation involved, we notice that "ideal" and "discrete" times are almost identical.

In all cases our shared memory implementation has low overhead and performs almost as well as the ideal case. Black Scholes shows the highest comparable overhead, since it has the lowest compute density: i.e., the amount of data transferred per unit computation time was the highest. Using Black Scholes we transfer about 13MB of data per second of computation time, while we transfer about 0.42MB of data per second of computation time when using Equake. Hence, the memory coherence overhead is negligible in Equake. The difference between the ideal scenario and our shared memory implementation increases with the number of cores, mainly due to synchronization overhead. In our implementation, synchronization penalties increase non-linearly with the number of cores.

"In all cases our shared memory implementation has low overhead and performs almost as well as the ideal case."

"Just as our technology does, OpenCL uses a weakly consistent shared memory model, but it is restricted to the GPU."

Related Work

There exists other research that is closely related to the work explicated in this article. Most closely related are the CUDA [11], OpenCL [12] and CTM [2] programming environments. Just as our technology does, OpenCL uses a weakly consistent shared memory model, but it is restricted to the GPU. Our work differs from CUDA, OpenCL, and CTM in several ways: unlike these environments we define a model for communication between the CPU and Larrabee processor; we provide direct user-level communication between the CPU and Larrabee processor, and we consider a bigger set of C language features, such as function pointers. Implementing a similar memory model is challenging on current GPUs due to their inherent limitations.

The Cell processor [8] is another heterogeneous platform. While the power processor unit (PPU) is akin to a CPU, the synergistic processing units (SPUs) are much simpler than the Larrabee cores. For example, they do not run an operating system kernel. Unlike the SPU-PPU pair, the Larrabee processor and CPU pair is much more loosely coupled, since the Larrabee processor can be paired as a discrete GPU with any CPU running any operating system. Unlike our model, Cell programming involves explicit direct memory access (DMA) between the PPU and SPU. Our memory model is similar to that of PGAS languages [14, 17], and hence our language constructs are similar to those of Unified Parallel C (UPC) language [17]. However, UPC does not consider ISA or operating system heterogeneity. Higher-level PGAS languages such as X10 [14] do not support the ownership mechanism that is crucial for a scalable, coherent implementation in a discrete scenario. Our implementation has similarities to software-distributed shared memory [9, 4] which also leverages virtual memory. Many of these S-DSM systems also use release consistency and they copy pages lazily on demand. The main differences with S-DSM systems is the level of heterogeneity. Unlike S-DSM systems, our system needs to consider a computing system where the processors have different ISAs and system environments. In particular, we need to support different processors with different virtual-to-physical page mappings. Finally, the performance tradeoffs between S-DSMs and CPU with Larrabee processor systems are different: S-DSMs were meant to scale on large clusters, while CPU with Larrabee processor systems should remain small scale clusters for some time in the future. The CUBA* [7] architecture proposes hardware support for faster communication between the CPU and GPU. However, the programming model assumes that the CPU and GPU are separate address spaces. The EXO* [18] model provides shared memory between a CPU and accelerators, but it requires the page tables to be kept in sync, which isn't feasible in a discrete accelerator.

"Unlike the SPU-PPU pair, the Larrabee processor and CPU pair is much more loosely coupled, since the Larrabee processor can be paired as a discrete GPU with any CPU running any operating system."

Conclusions

Heterogeneous computing platforms composed of a general-purpose scalar oriented CPU and throughput-oriented cores (e.g., a GPU) are increasingly being used in client computing systems. These platforms can be used for accelerating highly parallel workloads. There have been several programming model proposals for such platforms, but none of them address the CPU-GPU memory model. In this article we propose a new programming model for a heterogeneous Intel x86 system with the following key features:

- A shared memory model for all the cores in the platform.
- A uniform programming model for different configurations.
- User annotations to demarcate code for CPU and Larrabee execution.

We implemented the full software stack for our programming model including compiler and runtime support. We ported a number of parallel workloads to our programming model and evaluated the performance on a heterogeneous Intel x86 platform simulator. We show that our model can be implemented efficiently so that programmers are able to benefit from shared memory programming without paying a performance penalty.

"Heterogeneous computing platforms composed of a general-purpose scalar oriented CPU and throughput-oriented cores (e.g., a GPU) are increasingly being used in client computing systems."

References

[1] Adve S, Adve V, Hill M.D. and Vernon M.K. "Comparison of Hardware and Software Cache Coherence Schemes." *International Symposium on Computer Architecture* (ISCA), 1991.

[2] AMD CTM Guide. "Technical Reference Manual." 2006 Version 1.01. Available at ***http://www.amd.com***

[3] AMD Stream SDK. Available at ***ati.amd.com/technology/streamcomputing***.

[4] Amza C., Cox A.L., Dwarkadas S., Keleher P., Lu H., Rajamony R., Yu W., Zwaenepoel W. "TreadMarks: Shared Memory Computing on Networks of Workstations." *IEEE Computer*, 29(2): pages 18-28, February 1996.

[5] Boehm H., Adve S. "Foundations of the C++ memory model." *Programming Language Design and Implementation (PLDI)*, 2008.

[6] Dubey P. "Recognition, Mining, and Synthesis moves computers to the era of tera." Available at Technology@Intel, February 2005.

[7] Gelado I., Kelm J.H., Ryoo S., Navarro N., Lumetta S.S., Hwu W.W. "CUBA: An Architecture for Efficient CPU/Co-processor Data Communication." *ICS*, June 2008.

[8] Gschwind M., Hofstee H.P., Flachs B., Hopkins M., Watanabe Y., Yamakazi T. "Synergistic Processing in Cell's Multicore Architecture." *IEEE Micro*, April 2006.

[9] Kontothanasis L., Stets R., Hunt G., Rencuzogullari U., Altekar G., Dwarkadas S., Scott M.L. “Shared Memory Computing on Clusters with Symmetric Multiprocessors and System Area Networks.” *ACM Transactions on Computer Systems*, August 2005.

[10] Luebke, D., Harris, M., Krüger, J., Purcell, T., Govindaraju, N., Buck, I., Woolley, C., and Lefohn, A. “GPGPU: general purpose computation on graphics hardware.” *SIGGRAPH*, 2004.

[11] Nvidia Corporation. “CUDA Programming Environment.” Available at ***www.nvidia.com/***

[12] OpenCL 1.0. Available at ***http://www.khronos.org/opencl/***

[13] Ryoo S., Rodrigues C.I., Baghsorki S.S., Stone S.S., Kirk D.B., Hwu W.W. “Optimization Principles and Application Performance Evaluation of a Multithreaded Larrabee using CUDA.” *Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[14] Saraswat, V. A., Sarkar, V., and von Praun, C. “X10: concurrent programming for modern architectures.” *PPoPP*, 2007.

[15] Saha, B., Adl-Tabatabai, A., Ghuloum, A., Rajagopalan, M., Hudson, R. L., Petersen, L., Menon, V., Murphy, B., Shpeisman, T., Sprangle, E., Rohillah, A., Carmean, D., and Fang, J. ”Enabling scalability and performance in a large scale CMP environment.” *Eurosys*, 2007.

[16] 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., Hanrahan P. “Larrabee: A Many-Core x86 Architecture for Visual Computing.” *ACM Transactions on Graphics*, August 2008.

[17] “UPC Consortium, UPC language specifications.” *Lawrence Berkeley National Lab Tech Report* LBNL-59208, 2005.

[18] Wang P., Collins J.D., Chinya G. N., Jiang H., Tian X., Girkar M., Yang N. Y., Lueh G., Wang H. “Exochi: Architecture and programming environment for a heterogeneous multi-core multithreaded system.” *Programming Language Design and Implementation (PLDI)*, 2007.

Acknowledgments

We thank the following people who provided us with much support in this work: Jesse Fang, Ian Lewis, Antony Arciuolo, Kevin Myers, Robert Geva, Ravi Narayanaswamy, Milind Girkar, Rajiv Deodhar, Stephen Junkins, David Putzolu, Allen Hux, Peinan Zhang, Patrick Xi, Dean Macri, Nikhil Rao, Rob Kyanko, Madan Venugopal, Srinivas Chennupaty, Stephen Van Doren, Allen Baum, Martin Dixon, Ronny Ronen, Stephen Hunt, Larry Seiler, and Mike Upton.

Authors' Biographies

Bratin Saha is a Principal Engineer in Intel Labs and leads a team developing a new programming model for heterogeneous platforms. Prior to this he was a key architect and implementer for a scalable many-core runtime, the x86 memory model, transactional memory, and Nehalem synchronization. He holds a PhD degree in Computer Science from Yale University and a BS degree in Computer Science from the Indian Institute of Technology, Kharagpur. His e-mail is bratin.saha at intel.com.

Xiaocheng Zhou is a Researcher at Intel Labs, China. His current research mainly focuses on advanced programming language and runtime techniques for Intel many-core and heterogeneous platforms. Xiaocheng joined Intel in 2007. He received his PhD degree in Computer Science from the Institute of Computing Technology, Chinese Academy Sciences, in 2007 and his BS degree in Computer Science from Xiangtan University of China in 2001. His e-mail is xiaocheng.zhou at intel.com.

Hu Chen is a Research Scientist in Intel Labs, China. His current research focuses on system runtime, memory, and execution models in many-core architectures, and GPGPU. Hu joined Intel in 2004 where he has worked in the field of high-performance computing. He received MS and BS degrees from the University of Science and Technology of China. His e-mail is hu.tiger. chen at intel.com.

Ying Gao is a Research Scientist in Intel Labs, China. His current research areas are programming modes, system software, and runtime implementation. Ying received a BS degree in Mathematics in 2002 and a PhD degree in Computer Science from the University of Science and Technology of China in 2007. His e-mail is ying.gao at intel.com.

Shoumeng Yan is a Researcher in Intel Labs, China and works on language design, compiler implementation, and system optimization. His research interests include programming model design, language/compiler/runtime implementation, and many-core architectures. He received his PhD degree from Northwestern Polytechnical University in late 2005, and then joined Intel in early 2006. His e-mail is shoumeng.yan at intel.com.

Sai Luo is a Research Scientist in Intel Labs, China. His current research focuses on system runtime, memory models, and hardware design. Sai joined Intel in 2006 where he has worked on many-core runtimes. He received PhD, MS, and BS degrees from the University of Science and Technology of China. His e-mail is sai.luo at intel.com.

Copyright

FLEXIBLE AND ADAPTIVE ON-CHIP INTERCONNECT FOR TERA-SCALE ARCHITECTURES

Contributors

Mani Azimi
Intel Corporation

Donglai Dai
Intel Corporation

Akhilesh Kumar
Intel Corporation

Andres Mejia
Intel Corporation

Dongkook Park
Intel Corporation

Roy Saharoy
Intel Corporation

Aniruddha S. Vaidya
Intel Corporation

Index Words

Tera-scale Architecture
On-chip Interconnect
Flexible Topology
Adaptive Routing

Abstract

Tera-scale architectures represent a set of designs that enable high levels of parallelism to address the demands of existing and mostly emerging workloads. The on-chip interconnect element of such space is an essential ingredient with the desired flexibility and adaptivity to entertain the requirements of various designs. Highly integrated heterogeneous designs depend on a flexible interconnect topology to satisfy many different constraints. Demanding workloads create hot-spots and congestion in the network; thus, they desire an adaptive interconnect to respond gracefully to such transients. Other challenges, such as manufacturing defects, on-chip variation, and dynamic power management can also be better served through a flexible and adaptive interconnect.

In this article we present the design of an on-chip interconnect with aggressive latency, bandwidth, and energy characteristics that is also flexible and adaptive. We present the design choices and policies within the constraints of an on-chip interconnect and demonstrate the effectiveness of these choices for different usage scenarios.

Introduction

Tera-scale architecture provides the foundation that can be used across a wide array of application domains taking advantage of increasing device densities offered by Moore's law. A high degree of system integration, along with an architecture that can exploit different types of parallelism, characterizes this evolution. A typical implementation of such an architecture may include tens to hundreds of general-purpose compute elements, suitable for different types of parallelism, multiple levels of memory hierarchy to mitigate memory latency and bandwidth bottlenecks, and interfaces to off-chip memory and I/O devices.

One tractable implementation for such an architecture is a modular design where the building blocks are implemented with well-defined physical and logical interfaces and are connected through an on-chip interconnect to realize a specific product. The definition of specific products is determined by cost, power, and performance goals. A large set of products targeting the needs of a variety of applications can be derived from different combinations of a few building blocks. A flexible and powerful on-chip interconnect is an essential building block to realize this vision.

There are a few possible interconnect design options for the implementation space just suggested. One could partition the overall design into smaller sub-systems, design interconnects suitable for each sub-system, and use a more suitable interconnect for interconnecting the sub-systems in their entirety. Such a methodology requires the design, analysis, and composition of multiple interconnects fitting well together to provide a cohesive overall connectivity. This approach takes a static design time view of interconnects, which may be acceptable in some contexts, but is specific to the need and not general-purpose.

An alternative approach is to use a parameterized single interconnect design and apply it across all the sub-systems with specific parameters tuned for each sub-system. The design of such an interconnect is substantially more challenging. On the other hand, a flexible and adaptive general-purpose interconnect has the potential to meet the needs of such systems in a more systematic and tractable manner. We outline such an interconnect in this article.

In this context, we first discuss a few example usage scenarios and show how these favor the use of such a general-purpose interconnect.

Usage Scenarios

As indicated earlier, the on-chip interconnect architecture discussed here is targeted towards a general-purpose design that can meet the needs of different sub-systems in a highly integrated and highly parallel architecture. The challenge of the underlying architecture is to be competitive with the application-specific designs in the specific context of the applications. Here we discuss a few example scenarios that are potential targets for tera-scale architecture.

Cloud Computing or a Virtualized Data Center

The aggregation of compute capacity in tera-scale architectures can be partitioned and virtualized to provide cloud computing services to multiple applications sharing the infrastructure. Such an environment should allow dynamic allocation and management of compute, memory, and IO resources with as much isolation between different partitions as possible. A large set of allocation and de-allocation of resources can create fragmentation that may not provide a clean and regular boundary between resources allocated for different purposes. The bridging interconnect housing these resources should be flexible enough to allow such cases without degrading service levels and without causing undue interference between different partitions.

"Use a parameterized single interconnect design and apply it across all the sub-systems with specific parameters tuned for each sub-system."

Scientific Computing

Scientific computing applications cover the space from molecular dynamics to cosmic simulations, fluid dynamics, signal and image processing, data visualization, and other data and compute-intensive applications. These demanding applications are typically executed over large clusters of systems. The compute density per chip and energy per operation provided by tera-scale architecture can greatly enhance the capability of systems targeting this application domain. A typical design for this domain would be dominated by a large number of processing elements on chip with either hardware- or software-managed memory structures and provision for very high off-chip memory and system bandwidth. An interconnect designed for such applications should provide high bandwidth across all traffic patterns, even patterns that may be adversarial to a given topology.

Visual Computing

Visual computing applications are similar to scientific computing applications in some aspects. However, these applications may additionally have real-time requirements including bounded-time and/or bandwidth guarantees. Also, different portions of an application (e.g., AI, physical simulation, rendering, etc.) have distinct requirements. This heterogeneity, reflected architecturally, would create topological irregularities and traffic hot-spots that must be handled to provide predictable performance for visual workloads.

"Cost and yield constraints for products with large numbers of cores may create a requirement for masking manufacturing failures or in-field failures of on-die components."

Irregular Configurations

Cost and yield constraints for products with large numbers of cores may create a requirement for masking manufacturing failures or in-field failures of on-die components that in turn may result in configurations that deviate from the ideal topology of the on-chip interconnect. Another usage scenario that can create configurations that are less than ideal is an aggressive power-management strategy where certain segments of a chip are powered-down at low utilization. Such scenarios can be enabled only when the interconnect is capable of handling irregular topologies in a graceful manner.

Attributes of On-chip Interconnect

The on-chip interconnect for tera-scale architecture has to be designed keeping the usage scenarios just outlined in mind. Apart from the typical low-latency and high-bandwidth design goals, topology flexibility and predictable performance under a variety of loads are essential to enable the usage scenarios just described. We describe in this article the mechanisms to provide these features within the constraints of an on-chip interconnect for tera-scale architecture.

One of the main drivers for tera-scale architecture is the power and thermal constraints on a chip that have necessitated the shift from optimizing for scalar performance to optimizing for parallel performance. This shift has resulted in the integration of a higher number of relatively simpler cores on a chip instead of driving towards higher frequency and higher micro-architectural complexity. This trend also has implications for the on-chip interconnect. Because of the weak correlation of frequency with process technology generation, logic delay is becoming a diminishing component of the critical path, and wire delay is becoming the dominant component in the interconnect design. This implies that topologies that optimize for the reduction of wire delay will become preferred topologies for tera-scale architectures. Given the two-dimensional orientation of a chip, two-dimensional network topologies are obvious choices.

Another desirable attribute of an on-die interconnect is the ability to scale-up or scale-down by the addition or reduction of processor cores and other blocks in a cost-performance optimal and simple manner, thereby enabling the design to span several product segments. Based on the usage scenarios described at the beginning of this article, a latency optimized interconnect is critical for minimizing memory latency and ensuring good performance in a cache-coherent chip multi-processor (CMP). In addition, partitioning and isolation, as well as fault-tolerance require support, based on the envisaged usage scenarios. Because of all of these considerations, two-dimensional mesh, torus, and its variants are good contenders for tera-scale designs. (Figure 1) The design discussed in this article assumes mesh and torus as primary topologies and allows variations to enable implementation flexibility.

"Because of the weak correlation of frequency with process technology generation, logic delay is becoming a diminishing component of the critical path, and wire delay is becoming the dominant component in the interconnect design."

Even though on-chip wiring channels are abundant, shrinking core and memory structure dimensions and increasing numbers of agents on a die will put pressure on global wiring channels on the chip. This implies that wiring efficiency, i.e., the ability to effectively utilize a given number of global wires, will be an important characteristic of on-chip interconnect.

In order to support good overall throughput and latency characteristics with a modest buffering cost, our design assumes a buffered interconnect, based on wormhole switching [1] and virtual channel flow control [2]. It supports multiple virtual channels to allow different types of traffic to share the physical wires. The details of this design are discussed in the following section.

The organization of the rest of this article is as follows. In the next section we present the details of the on-chip interconnect design, primarily focusing on the micro-architecture of the router and highlighting the choices made to optimize for latency and energy efficiency. We then focus on the support for flexibility and adaptivity, discussing the options and our preferred direction. The performance results are then presented for different usage scenarios, indicating the benefit derived from the various features in the interconnect. Next we discuss some of the related work and conclude our article with a summary of our work and next steps.

On-chip Interconnect Design

Before getting into the specifics of the micro-architecture of this design, let us look at the topological aspects.

Supported Interconnect Topologies

The two dimensional on-die interconnect architecture we present is optimized for a CMP that is based on a tiled design supporting the mesh and torus topologies and a few variants. Some representative topologies are shown in Figure 1. Each processor tile may have a router that is connected to the local tile as well as to four other routers. Agents on the periphery can be connected to the local port of the router or directly to an unused router port of a neighboring tile. Other variations include two or more cores sharing a router, resulting in a concentrated topology that uses fewer routers for interconnecting the same number of cores. Another configuration is a torus with links that fold back along one or both of the horizontal and vertical directions. To balance wire delays in the torus topologies, links connect routers in alternate rows and/or columns.

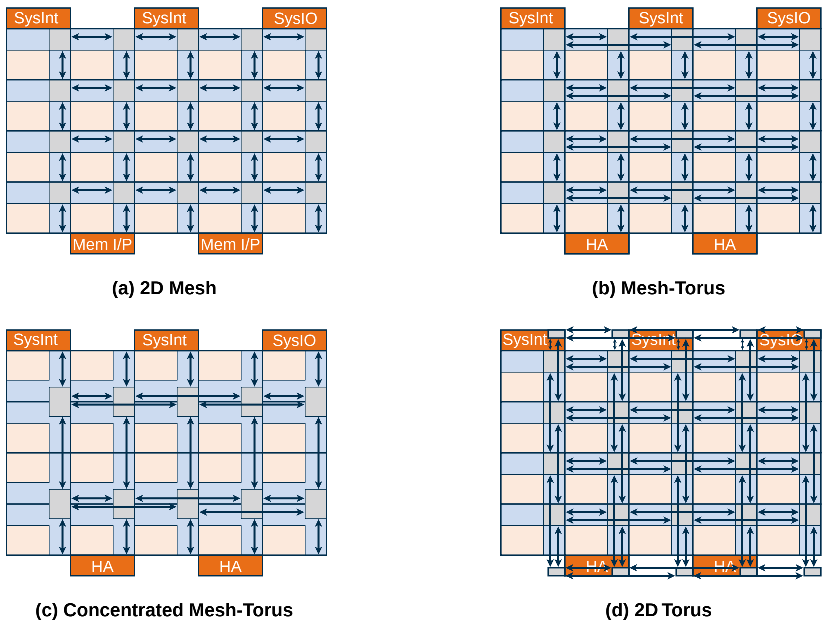


Figure 1: Examples of 2D Mesh and Torus Variants
Source: Intel Corporation, 2009

Router Micro-architecture

The principal component of the 2D interconnect fabric is a pipelined, low-latency router with programmable routing algorithms that include support for performance isolation, fault-tolerance, and dynamic adaptive routing based on network conditions.

"The principal component of the 2D interconnect fabric is a pipelined, low-latency router with programmable routing algorithms."

Figure 2 shows the stages of the router pipeline as well as the key functions performed in each stage. The functionality in our adaptive router includes most of the standard functionality that one can expect to see in a wormhole switched router [3].

Route Compute

This includes route computation for the header flit to determine the route, i.e., the output port of the router, which a packet must take towards its destination tile. In an adaptive routing scheme, this can imply that the packet may have a choice of more than one output port towards its destination (we support up to two output port choices). Our router architecture uses route pre-computation [4] for the route decision of neighboring routers, thereby removing route-computation from the critical path of the router pipeline.(Route pre-compute options and design tradeoffs are discussed in "Flexible and Adaptive On-chip Interconnect.").

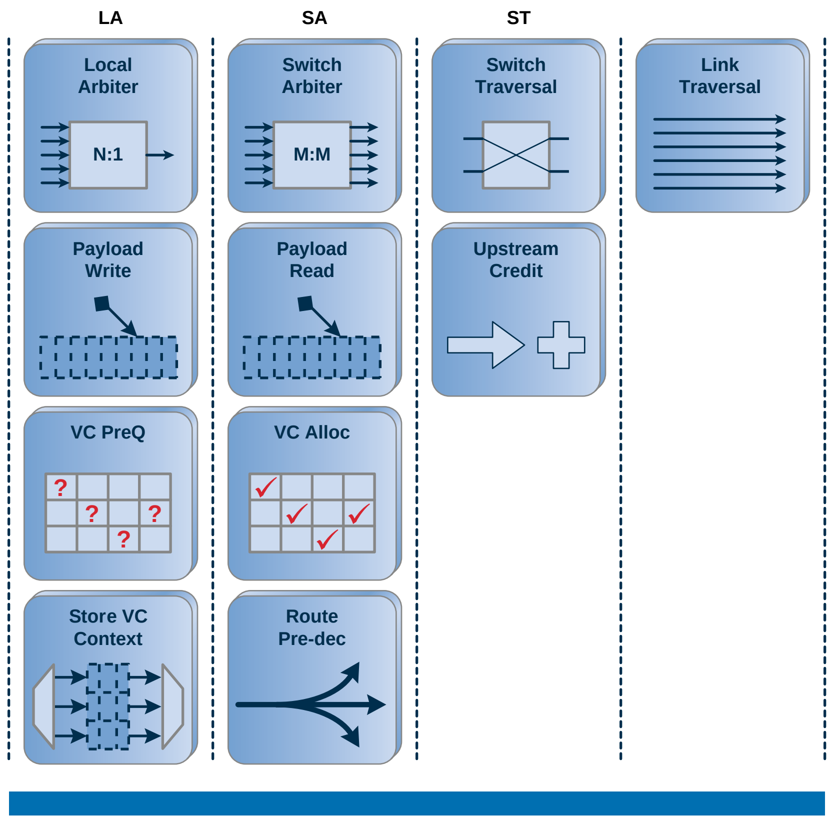


Figure 2: Adaptive Router Pipeline
Source: Intel Corporation, 2009

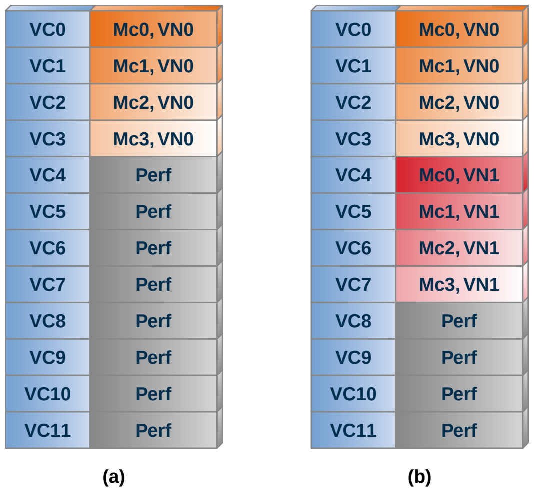


Figure 3: Mapping of Virtual Networks to Virtual Channels for a 2D Mesh and 2D Torus Topology
Source: Intel Corporation, 2009

Virtual Channels and Buffer Management

Our router architecture uses virtual channel flow control [21] both to improve performance and to enable support for deadlock-free routing in various flavors of deterministic, fault-tolerant and adaptive routing. The set of virtual channels (VCs) can be flexibly partitioned into two logical sets: routing VCs and performance VCs. Routing VCs are also logically grouped into virtual networks (VNs). VCs belonging to the same VN are used for message-class separation, but they use the same routing discipline. Routing VCs are used for satisfying deadlock-freedom requirements of a particular routing algorithm employed. Performance VCs belong to a common shared pool of VCs used for performance improvement (both for adaptive or deterministic routing schemes). Figure 3 shows an example of potential mappings of VCs to routing VCs belonging to specific message-classes and VNs, as well as the pool of performance VCs for 2D mesh and 2D torus topologies with minimal deadlock free XY routing algorithms for each network. The configuration is assumed to support 12 VCs and 4 message classes; the mesh requires a single VN (VN0) for deadlock freedom, whereas the torus requires 2VNs (VN0, VN1).

To support flexibility and optimal usage of packet buffering resources, our router supports a shared buffer at the input port that is used by all VNs in that port. The space in the buffer is dynamically managed by using a set of linked lists that track flits belonging to a given packet in a VC, as well as a list of free buffer slots for incoming flits. The operation of the flit-buffer is shown in Figure 4.

Depending on the size of the buffer, a banked implementation is used where each bank can be individually put into low-leakage states to save power at low levels of buffer utilization. Active power-management strategies to trade off performance and power can also be implemented.

Flow Control

The router supports credit-based flow control to manage the downstream buffering resources optimally. The flow control protocol is a typical scheme, except for the fact that it needs to handle the shared flit buffer resources and ensure reservation of at least 1-buffer slot resource for all routing VCs and also for each active performance VC. This is required to guarantee resource deadlock freedom with wormhole switching.

Router Pipeline and Arbitration Mechanisms

Our adaptive router pipeline is a high-performance pipeline with the bypass capability of one pipeline stage based on network conditions.

"Our adaptive router pipeline is a high-performance pipeline with the bypass capability of one pipeline stage based on network conditions."

A flit belonging to a packet enters the router at the input port. For a new packet, a message context is created to track relevant status information. The key functionality in the local arbitration stage is to determine a single packet at a given input that can contend for a router's output port with other potential candidates from other input ports.

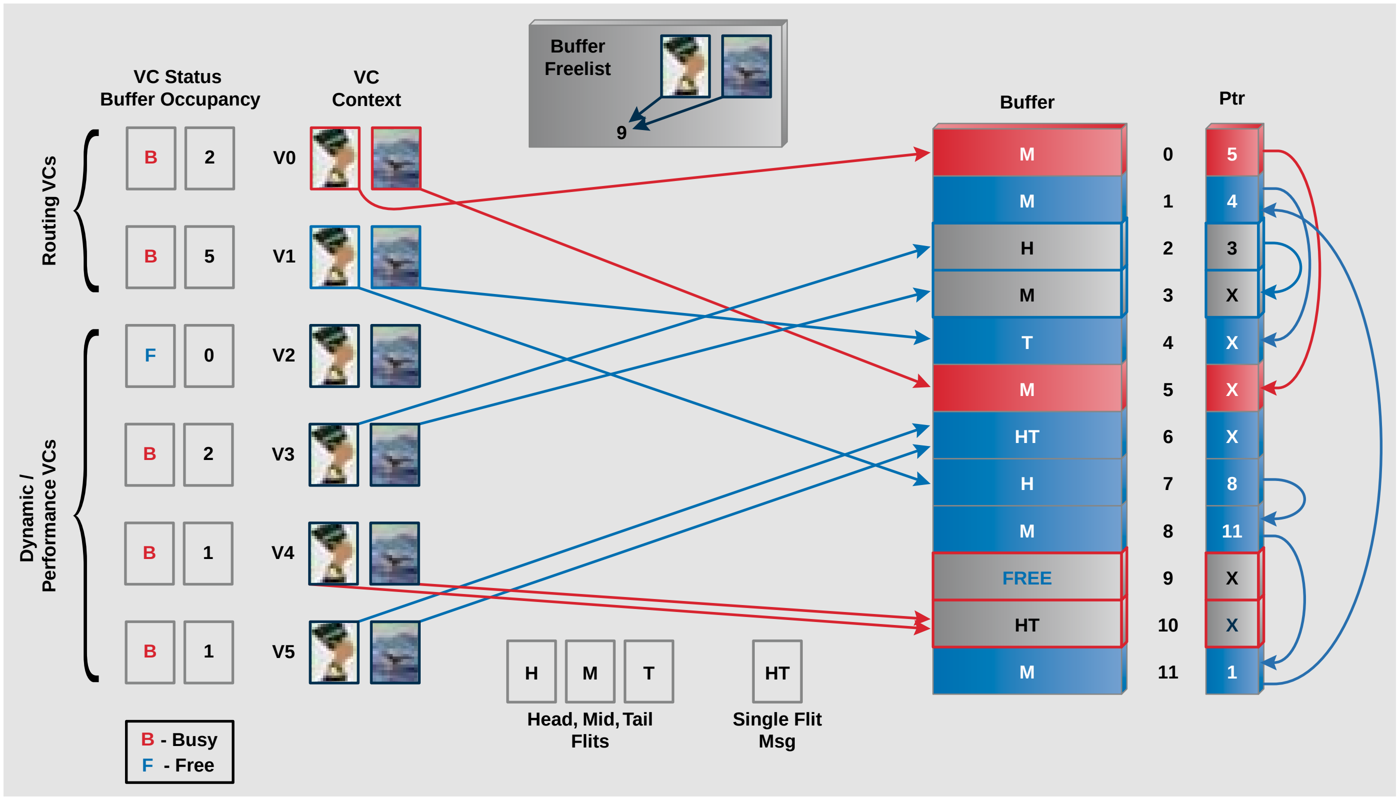


Figure 4: Shared Flit Buffer Management Across a Set of Virtual Channels
Source: Intel Corporation, 2009

Our router applies a filter on less viable packets to improve the performance of the local and global arbitration stages. Each packet may have up to two output-port candidates that may be used to route the packet to its desired destination. Both input and output side-arbiters make arbitration decisions, based on a round-robin priority.

Each output port selects a single winning input port from amongst one or more candidates from all input ports for the following cycle by using a rotating priority. A multi-flit packet may make a "hold" request, and the arbiter grants the output port to such an input port for an additional cycle.

VC allocation, and allocation of a downstream buffer slot for the granted flit, also occurs after switch arbitration. Route pre-computation or route table lookup for the next router downstream occurs concurrently with arbitration. The final stage of the router pipeline is switch traversal from inputs to outputs. Appropriate sideband information for credit management (and setup of information based on look-ahead routing for header flits) is done at this stage, and the flit flows out of the router over the interconnect links to the input port of a downstream router.

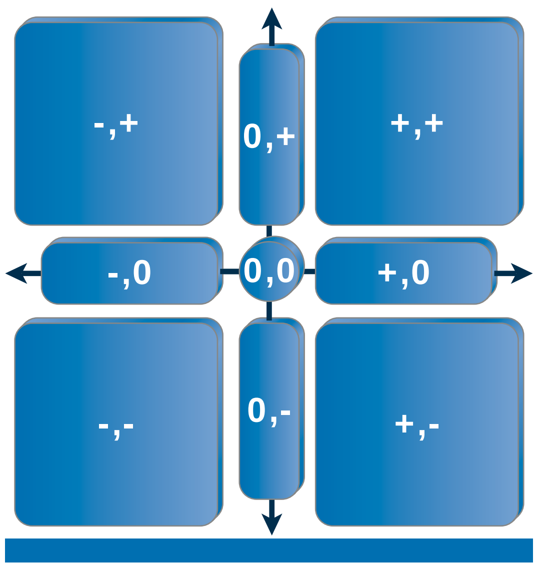


Figure 5: The Nine Regions a Destination Node can Belong to Relative to a Given Router
Source: Intel Corporation, 2009

Flexible and Adaptive On-chip Interconnect

In this section we describe the support for various routing algorithms to enable a flexible, configurable, and adaptive interconnect, and we discuss the design implications.

Support for Multiple Routing Algorithms

The router architecture supports distributed routing wherein the subsets of the routing decisions are made at each router along the path taken by a given packet.

In two-dimensional networks like mesh and torus, given a source node, the set of shortest paths to the destination fall into nine possible regions, as shown in Figure 5. The regions are labeled in terms of the minimal X and Y offset of shortest paths.

The baseline routing strategies supported by our router use this underlying classification of the destination for narrowing the set of paths to the destination [4, 5]. For minimal adaptive routing, up to two distinct directions may be permitted based on the region a destination node falls into. The routing decision (i.e., output ports and VN choices permitted) at each router is based on the current input port and VN a packet belongs to, as well as on the desired destination. For each VN, we can support the flexible algorithms with a small 9-entry table [4, 5] or with an even more economical storage of only a few bits per port [6].

With the baseline routing support alone, minimal path deterministic routing in mesh and torus and partially and fully adaptive minimal path routing algorithms, such as those based on the turn model [7], are supported by our implementation. Our adaptive router architecture also supports the Duato protocol [8] which reduces the VC resource requirements while providing full adaptivity. Table 1 shows a comparison of the minimum number of VCs required to implement a deadlock-free turn-model that is based on fully-adaptive routing versus that for a Duato-protocol-based implementation.

Topology	2 Msg Classes		4 Msg Classes	
	Turn Model	Duato Protocol	Turn Model	Duato Protocol
2D Mesh	4	3	8	5
2D Torus	6	5	12	9

Table 1: Minimum Virtual Channels Required for Fully-adaptive Routing (Turn Model v/s Duato Protocol)
Source: Intel Corporation, 2009

Baseline routing support also enables a deterministic fault-tolerant routing algorithm based on fault-region marking and fault-avoidance, such as in [9], as well as adaptive fault-tolerant routing algorithms [10]. Incomplete or irregular topologies caused by partial shutdown of the interconnect because of power-performance tradeoffs can be treated in a manner similar to a network with faults for routing re-configuration purposes.

Our router architecture also supports a novel two-phase routing protocol where a packet is routed to an intermediate destination and then to the final destination. This can be used to implement load-balancing and fault-tolerant routing algorithms, including non-minimal routing algorithms.

Finally, we also implement support for performance isolation amongst partitions on a mesh which may or may not have rectangular geometries. This is implemented through a hierarchical routing protocol that helps isolate communication of each partition including for non-rectangular partitions. This is purely a routing-based approach to providing isolation, and no additional resource management mechanisms, such as bandwidth reservation, are required with this approach.

Design Implications

Our flexible design had to have a minimal cost overhead in the definition phase. We outline next the various impacts of our design choices.

We maintain a high-performance router design by using a shared pool of VCs, as well as a shared buffer pool. The shared buffer pool reduces the overall buffer size and power requirements. Support for up to two output port choices with adaptive routing has little impact on the router performance. With the decoupled local and switch arbitration stages in the router pipeline (as described earlier), each packet arbitrates for a single output port candidate after applying the arbitration filter and path selection. The criteria used could be based on several congestion prediction heuristics that use locally available information, such as resource availability and/or resource usage history (see [4] for examples of such criteria). Path selection can be implemented without a significant impact on the arbitration stages.

"Our router architecture also supports a novel two-phase routing protocol where a packet is routed to an intermediate destination and then to the final destination."

Support for the adaptivity and flexible routing configuration requires the use of configurable route tables. While table storage requirements are small for the 9-entry table (approximately 3-4 percent per VN, when compared to the flit buffer storage requirement), these can be reduced by an order of magnitude by using the approaches in [6]. These table storage optimizations, that are independent of the network size, considerably reduce the area and power requirements for supporting flexibility features in the on-chip interconnect.

In the next section we describe some representative performance results for flexible interconnection network architecture.

Performance Results

We present the performance results for different configurations and traffic patterns to demonstrate the effectiveness of the adaptive routing scheme just described. However, before presenting the results on adaptive routing, we present the results to establish the optimal design parameters for typical operation.

"Table storage optimizations, that are independent of the network size, considerably reduce the area and power requirements."

Figure 6 shows the effect of buffer size and number of VNs on each input port on the network capacity for a uniform random traffic pattern on a 6x6, two-dimensional mesh network. This traffic pattern uses a mix of single flit and five-flit packets divided between two message classes, based on an expected distribution for cache coherency traffic. The target of each packet is randomly selected with equal probability, and the injection rate is increased until the network is close to saturation. For different buffer depths and VC settings, the network saturates at different points. Each plot in Figure 6 represents different buffer depths and shows the delivered throughput in terms of flits accepted per cycle per node for different numbers of VCs on the X-axis. The plots indicate that a larger number of buffers and VCs result in increased throughput; however, the improvement tapers off beyond sixteen buffers and ten virtual channels. These results were obtained by using a deterministic dimension-order routing scheme, since adaptive schemes are not beneficial for uniform random traffic patterns. We use sixteen buffers and twelve virtual channels as the baseline to evaluate the effectiveness of an adaptive routing scheme against that of a deterministic scheme.

To illustrate the effectiveness of an adaptive routing scheme, we use a traffic pattern that is adversarial to two-dimensional mesh topology. Adversarial traffic for a given topology and routing scheme illustrates a worst-case scenario, which shows the extent of degradation in network performance for some traffic patterns. A matrix transpose is a perfect example for a mesh with X-Y deterministic routing, since it results in an uneven traffic load across the links along the diagonal.

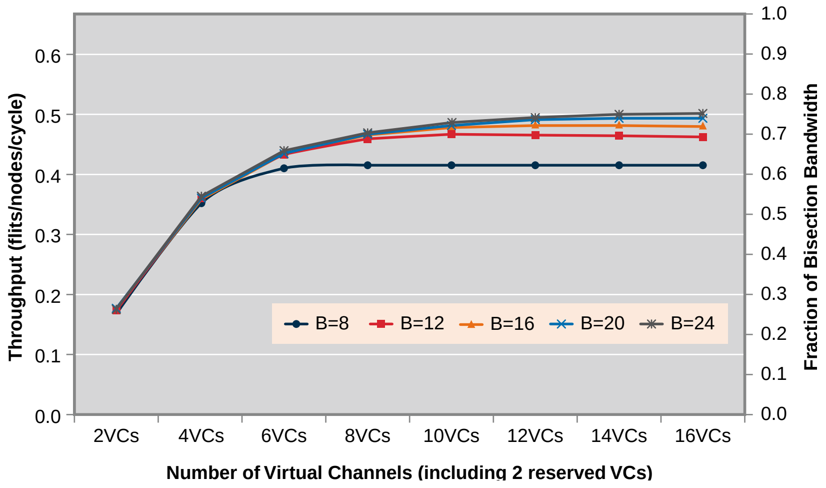


Figure 6: Effect of Number of Buffers and Virtual Channels on Delivered Throughput for a 6x6 Mesh
Source: Intel Corporation, 2009

Figure 7 illustrates the source-destination relationship for a transpose operation. For this traffic pattern, a node labeled (i,j) communicates with node (j,i) and vice-versa. This results in nodes in the upper triangle (highlighted in orange) communicating with nodes in the lower triangle (highlighted in pink). The nodes on the diagonal (highlighted in blue) do not generate any traffic. Figure 7 also illustrates the route taken when a deterministic XY routing scheme and some possible alternatives allowed by an adaptive routing scheme are used. Paths highlighted with thick lines do not follow the XY routing scheme. There are many other paths possible between these source and destination nodes that are not shown in the figure. A deterministic routing scheme tends to concentrate load on a few links in the network for this traffic pattern, and not utilize other alternative paths between source-destination pairs. An adaptive routing scheme that allows more flexibility in path selection among multiple alternatives avoids congested routes and improves the overall capacity of the network.

The effect of path diversity in increasing the overall network capacity is illustrated through the load-throughput plot in Figure 8 for transpose traffic by using different routing schemes. XY represents a deterministic routing scheme, whereas Adaptive is an implementation of a minimal fully-adaptive routing scheme using the Duato protocol. As shown in the plot, the network capacity is severely restricted for this pattern when a deterministic routing scheme is used. The adaptive routing scheme delivers much higher throughput for this traffic pattern.

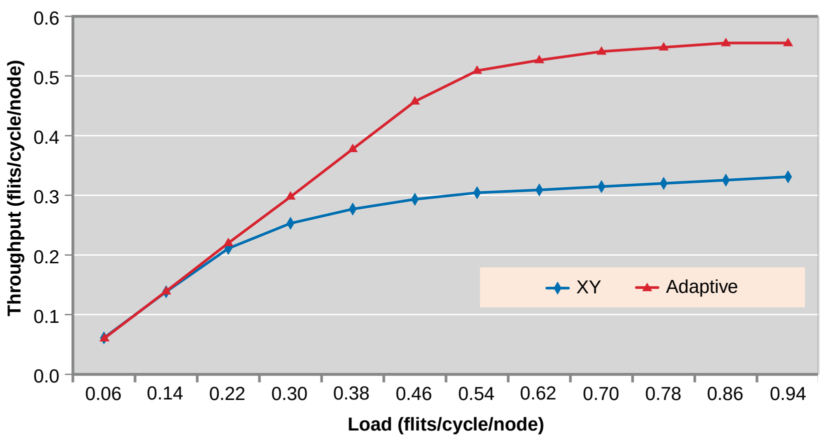


Figure 8: NetworkThroughput with a Deterministic and Adaptive Routing Scheme for Transpose Traffic on a 6x6 Mesh
Source: Intel Corporation, 2009

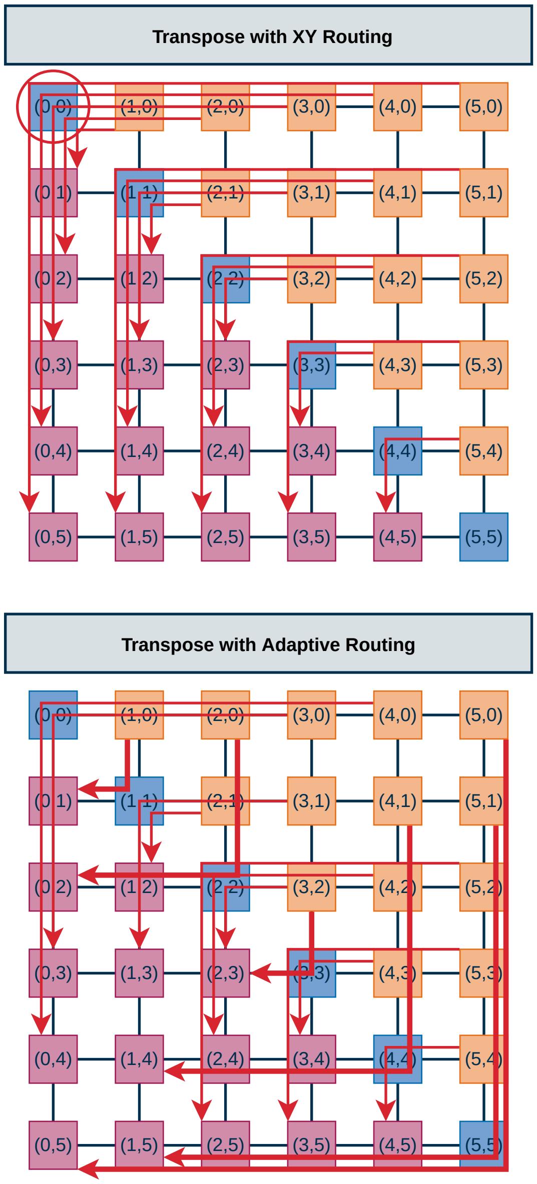


Figure 7: Traffic Pattern for a Transpose Operation with Deterministic and Adaptive Routing
Source: Intel Corporation, 2009

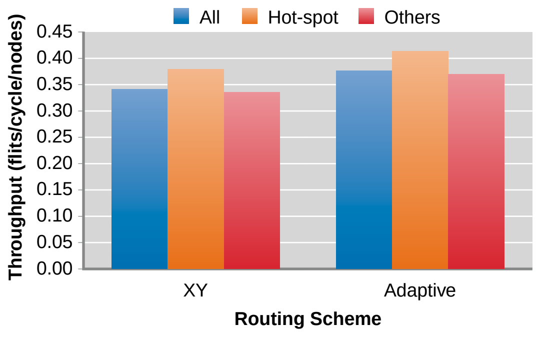


Figure 9: Network Capacity with and without Adaptive Routing in a Heterogeneous Environment
Source: Intel Corporation, 2009

When all the agents on the on-chip network are not of the same type, non-homogeneous traffic patterns can be created with the potential for transient hot-spots. One example of this is the traffic going to and generated from memory controllers, I/O interfaces, and system interconnect. Depending on the phases of execution or type of workload, some of these agents may be more heavily used than others and may become bottlenecks on the interconnect, thereby affecting other traffic that shares the path with congested traffic. Such scenarios can be crudely approximated with a mix of traffic patterns between different sets of agents. To illustrate this scenario, we set up an experiment with a 6x6 mesh network where six agents at the periphery (three in the top row and three in the bottom row) are considered hot-spots with 30 percent of the traffic targeting these nodes with equal probability and the rest of the traffic targeting the remaining nodes with equal probability. Throughput delivered to each node in terms of flits per cycle was measured for all nodes combined, for hot-spot nodes, and for nodes excluding hot-spot nodes. The result is illustrated in Figure 9, which shows that overall throughput as well as the throughput delivered to hot-spot nodes and other nodes excluding the hot-spot nodes have improved.

The interconnection network can be reconfigured in the presence of faults through adoption of fault-tolerant routing algorithms. In the absence of fault-tolerant routing, the network can become virtually unusable in the presence of high levels of identified and isolated faulty components. It may be possible to operate the network at a significantly diminished capacity by avoiding complete rows and/or columns of nodes containing a faulty node; i.e., use the nodes in that column or row as pass-through nodes. Here, we do not address a comparative performance benefit of fault-tolerant routing. However, in Figure 10 we show the fraction of usable nodes in the network and the resulting impact to the network latency as a sensitivity study to the number of faults in the network. The data are derived from an implementation of a fault-tolerant XY routing [9] which restricts the faulty regions to being rectangular in shape. The node failures in this context can be the result of core or router failure. Figure 10 (a) shows that when the number of randomly located faulty nodes is small, little or no additional nodes may be needed to be turned off to satisfy the rectangular shape constraint of faulty regions and to safely support the use of the rest of the network. As the number of faults increases, additional nodes, that are unsafe and prevent deadlock-free routing, need to be turned off. Figure 10 (b) shows that the increase in the average number of hops in the network is also small for a small number of faults. As the number of faults increases, the average number of unsafe nodes increases and then begins to drop off as the overall working cluster size (actual remaining functional system) diminishes, compared to the size of the full network. The actual region of interest is for a single to a small number of node failures for which fault-tolerant routing provides graceful degradation in performance.

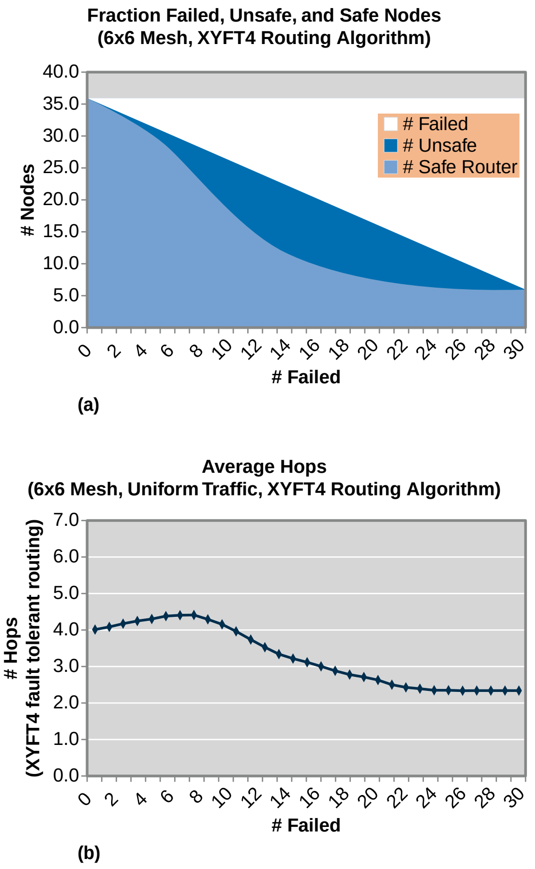


Figure 10. Fault-tolerant XY Routing: (a) Usable Nodes vs. Faults and (b) Average Hops vs. Faults
Source: Intel Corporation, 2009

Related Work

With increasing interest in CMP and system-on-chip (SoC) designs, on-chip networks are a very active area of research. The current generation of CMP and SoC designs have taken very different approaches: CMP designs favor customized, implementation-specific, on-chip networks that change from generation to generation, whereas SoC designs use standardized networks that allow quick integration of different design blocks. As the complexity and level of integration increases with CMPs, it is likely that CMP designs will also move towards standardized networks. Some of the research prototypes, such as Intel's tera-FLOP processor [11] and other recent designs from different vendors, have indicated a move towards this trend. The design presented in this article is a leap forward from the design used in [11]: it addresses some of the performance bottlenecks, such as the limited number of VCs, a more optimized design with lower latency, area and power, and advanced capabilities, such as flexible and adaptive routing.

"With increasing interest in CMP and system-on-chip (SoC) designs, on-chip networks are a very active area of research."

Research into flexible and adaptive networks has been ongoing for a long time, and the techniques developed have been applied in various networks in operation. The closest interconnect type to on-chip networks, where similar requirements and tradeoffs are at play, is in the area of multi-processor networks. However, on-chip networks have to be further optimized for latency and storage overheads, a factor that drove the decisions for the design presented in this article. For example, off-chip multi-processor networks have typically used virtual cut-through techniques [12, 13, 14], but these designs could result in larger buffering overheads, as the number of VCs increases. The design presented in this article uses wormhole switching [1], which does not require buffers to be allocated for the entire packet.

"The design presented in this article uses wormhole switching [1], which does not require buffers to be allocated for the entire packet."

The route specification mechanism used in this work relies on either a compressed table or on a LBDR [4, 5, 6] mechanism for look-ahead routing [4] such that route determination does not add to the pipeline delay at each router. Off-chip networks, such as the IBM Blue Gene/L* torus network [12] avoid using any table, and it relies on X, Y, and Z offsets to destination to make routing decisions. However, the Blue Gene/L network does not allow any topological irregularities and cannot tolerate link or router failures. Routing schemes used in the Cray T3E* [13] system allows more flexibility and can tolerate link and router failures, but adaptive routing is disabled when minimal paths from source to destination contain broken links. Cray T3E implementation requires a table as large as the system size (up to 544 entries) to determine the routing tags. As the number of agents grows on a chip, large table sizes result in area and power overheads.

Note that Cray T3E also uses an adaptive routing scheme by using escape VCs [8] similar to the scheme used in our implementation. However, the Cray T3E implementation has only one VC for adaptive routing usage, and it uses a virtual cut-through in the adaptive channels that require space for the entire packet. Such an implementation limits the benefit of adaptivity. Our implementation uses wormhole switching, even in the adaptive channels, and it allows multiple VCs to be used for packets, with the use of adaptive routing.

"As the number of agents grows on a chip, large table sizes result in area and power overheads."

Conclusions

We have presented the details of an on-die fabric with an adaptive router that supports various 2D interconnect topologies for tera-scale architectures. 2D mesh and torus topologies provide good latency and bandwidth scaling for tens to more than a hundred processor cores, as well as the additional connectivity that enables multiple paths between source-destination pairs that can be exploited by adaptive routing algorithms.

The architecture presented has an aggressive low-latency router pipeline. It can also provide high throughput in the presence of adversarial traffic patterns and hot-spots through the use of adaptive routing. The adaptive routing is supported without impacting the router pipeline performance, and it is supported through very economical and configurable routing table storage requirements. This router also supports very efficient use of resources by making use of shared (performance) VC and buffer pools.

The propsed fabric architecture and routing algorithms also support the ability to provide partitioning with performance isolation and the ability to tolerate several faults or irregularities in the topology, such as those caused by partial shutdown of processing cores and other components for power management.

References

[1] Dally, W. J. and Seitz, C. L. 1987. "Deadlock Free Message Routing in Multiprocessor Interconnection Networks." *IEEE Trans. Computers*, pages 547-553.

[2] Dally, W. J. 1992. "Virtual channel flow control." *IEEE Trans. Parallel and Distributed Systems*, pages 194-205.

[3] Dally, W. J. and Towles, B. 2004. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann.

[4] Vaidya, A., Sivasubramaniam A. and Das, C. R. 1999. "LAPSES: A recipe for high performance adaptive router design." In *Proceedings International Symposium on High Performance Computer Architecture (HPCA-5)*, pages 236-243.

[5] Flich, J., Mejia, A., Lopez, P. and Duato, J. "Region-Based Routing: An Efficient Routing Mechanism to Tackle Unreliable Hardware in Network on Chips." *International Symposium on Networks on Chip*, 2007.

[6] Flich, J., Rodrigo, S. and Duato, J. "An Efficient Implementation of Distributed Routing Algorithms for NoCs." *International Symposium on Networks on Chip*, 2008.

[7] Glass, C. J and Ni, L. M. "The Turn Model for Adaptive Routing." *Journal of the ACM*, pages 874-902, 1994.

[8] Duato, J. "A Necessary and Sufficient Condition for Deadlock-Free Adaptive Routing in Wormhole Networks." *IEEE Transactions on Parallel and Distributed Systems*, 1055-1067, 1995.

[9] Bopanna, R.V. and Chalasani, S. "Fault-Tolerant Wormhole Routing Algorithms for Mesh Networks." *IEEE Transactions on Computers*, pages 848–864, 1995.

[10] Duato, J. "A theory of fault-tolerant routing in wormhole networks." *IEEE Transactions on Parallel and Distributed Systems*, pages 600-607, 1994.

[11] Vangal, S., et al. "An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS." *IEEE International Solid-State Circuits Conference*, 2007.

[12] Adiga, N. R., et. al. "Blue Gene/L torus interconnection network." *IBM Journal on Research and Development*, 2005.

[13] Scott, S. and Thorson, G. "The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus." In *Proceedings on Hot Interconnects IV*, 147-156, 1996.

[14] Maddox, R. A., Singh, G., and Safranek, R. J. *Weaving High Performance Multiprocessor Fabric*. Intel Press, Beaverton, OR, 2009.

Acknowledgments

The work presented here has benefited from the contributions and insight of our colleagues Partha Kundu and Gaspar Mora Porta as well as our former colleagues Jay Jayasimha and the late David James.

We are also deeply thankful to our senior leaders Joe Schutz, Jim Held, Justin Rattner, and Andrew Chien for their continued support and encouragement for our work on scalable on-die interconnect fabrics through the tera-scale initiative.

Author Biographies

Mani Azimi is a Senior Principal Engineer and Director of the Platform Architecture Research team in the Microprocessor and Programming Research (MPR) group in Intel Labs. He received his PhD degree from Purdue University. He joined Intel in 1990 and has worked on a wide range of platform architecture topics including system protocol, processor interface, MP cache controller architecture, and performance modeling/analysis. He is currently focusing on tera-scale computer architecture challenges. His e-mail is mani.azimi at intel.com.

Donglai Dai is a Senior Research Scientist at Intel Labs. He joined Intel in 2007 and has worked on on-chip interconnect architecture and design for Intel's tera-scale computing initiative. Previously, he worked as an architect and lead designer of the threaded vector coprocessor, memory sub-system and coherence protocol, and interconnection network of high-performance computing (HPC) systems in Cray Inc. and Silicon Graphics Inc. He received a Ph.D. degree in Computer Science from Ohio State University. His e-mail is donglai.dai at intel.com.

Akhilesh Kumar is a Principal Engineer in Intel Labs. His research interests include cache organization, on–chip and off–chip interconnects, and interface protocols. He received his PhD degree in Computer Science from Texas A&M University. He joined Intel in 1996 and has contributed to the architecture and definition of system interfaces and cache hierarchies. His e-mail is akhilesh.kumar at intel.com.

Andres Mejia is a Senior Research Scientist at Intel Labs. His research interests include high-speed interconnects, cache organization, and performance modeling/analysis. He received his PhD degree in Computer Science from the Technical University of Valencia. His e-mail is andres.mejia at intel.com.

Dongkook Park is a Research Scientist in Intel Labs in Santa Clara, California. He received his PhD degree in Computer Science and Engineering from the Pennsylvania State University and he joined Intel in 2007. His current research interests include on-chip interconnects and the microarchitecture of on-chip routers for future many-core processors. His e-mail is dongkook.park at intel.com.

Roy Saharoy is a Research Scientist at Intel Labs in Santa Clara, California. His contributions at Intel include the development of micro-architecture performance models, performance analysis of interconnects, pre-silicon validation of platform architecture, performance projection, and performance tuning of workloads. His current focus is on performance modeling, simulation methodology, and workload-based characterization of scalable on-die interconnects. Roy has many years of work experience in diverse areas of software development. He holds an M.Tech degree from the Indian Institute of Technology (Kharagpur) India and conducts industry-sponsored research at the Electrical and Computer Engineering Department of the University of Illinois, Urbana-Champaign. His e-mail is roy.saharoy at intel.com.

Aniruddha S. Vaidya is a Research Scientist at Intel Labs in Santa Clara, California. His contributions at Intel include workload characterization, performance analysis, and architecture of server platforms. His current focus is on router and interconnection network architecture for Intel's tera-scale computing initiative. Ani has B.Tech and M.Sc. (Engg.) degrees from Banaras Hindu University and the Indian Institute of Science, and a PhD degree in Computer Science and Engineering from Pennsylvania State University. His e-mail is aniruddha.vaidya at intel.com.

Copyright

TERA-SCALE MEMORY CHALLENGES AND SOLUTIONS

Contributors

Dave Dunning
Intel Corporation

Randy Mooney
Intel Corporation

Pat Stolt
Intel Corporation

Bryan Casper
Intel Corporation

Index Words

DRAM
DDR
I/O Channels
Bandwidth
Power Efficiency

Abstract

Integrated circuit processing technology and computer architectures continue to mature. Single chip CPUs have been demonstrated to exceed one TeraFLOP [1]. This high level of computation, concentrated in a small area, creates many system design challenges. One of these major challenges is designing and building a memory sub-system that allows these CPUs to perform well while staying within reasonable system cost, volume, and power constraints.

This article describes the challenges that tera-scale computing presents to the memory sub-system, such as performance metrics including memory bandwidth capacity and latency, as well as the physical challenges of packaging and memory channel design. New technologies that need to be developed and matured for tera-scale memory sub-systems are also discussed.

Introduction

The architecture and construction of computers have gone through many changes. The first electronic digital computer was invented in 1939, and it is usually credited to John V. Atanasaff and Clifford Berry from Iowa State University. It consisted of vacuum tubes, capacitors, and a rotating drum memory. In 1945, the ENIAC was built: it weighed over 20 tons and filled a large room. In 1948, a team of engineers at Manchester University built a machine nicknamed "the baby." This was the first computer that was able to store its own programs, and it is usually considered to be the forerunner to the computers we use today. A type of altered cathode ray tube was used to store data.

The design constraints of these early electronic computers were the required computer room volume and the power consumed, primarily due to the use of thousands of vacuum tubes. In 1959, Jack Kilby, then at Texas Instruments, and Robert Noyce, then at Fairchild Semiconductor, invented the monolithic integrated circuit. The transition to integrated circuits was the start of an impressive treadmill for the computer industry where more and more transistors could be built and connected within a monolithic "chip." There have been varying different architectures, often categorized as mainframes and microprocessors (microcomputers). The level of integration for these computers has increased over time, but the basic architecture comprising a central processing unit (CPU) and memory has remained.

In the early years of electronic computers, drum memory was often used. This consisted of a rotating drum coated with ferromagnetic material and containing a row of read and write heads. Drum memory was followed by core memory that used magnetic rings to store information in the orientation of the magnetic field. As transistors matured, core memory was replaced with microchips, by using configurations of transistors and, often, capacitors to store digital information.

CPUs and memory have evolved to use high transistor count integrated circuits; they both use complementary metal oxide semiconductor (CMOS) technology. Although the components of CPUs and memory are built basically in the same way, memory and CPU continue to be partitioned into different sub-systems within a computer. Such a method has worked well up to this point: the transistor density (transistors per area of silicon substrate) for CPUs and the density of those used to create memory chips continue to follow Moore's Law [1]. These increases in integration have allowed for increased functionality and increased clock frequency, both leading to impressive improvements in system performance and reductions in system costs.

"The level of integration for computers has increased over time, but the basic architecture comprising a central processing unit (CPU) and memory has remained."

In the last decade, however, double data rate (DDR) memory has emerged as the dominant memory technology (in terms of number of units sold). Some of the key features that have made DDR memory appealing are the low cost per bit, sufficient bandwidth to supply instructions and data to the CPUs, and the shared bus aspect of the interfaces to the memory chips: more memory can be added to a memory channel interface if more capacity is needed, a form of "pay as you go" method of memory acquisition.

Recently, however, we are starting to see some strain in the ability of DDR-based memory to meet the needs of higher- and higher-performing tera-scale CPUs. The two constraints being felt most are the increasing power consumption that comes with the increased power density, as well as the electro-mechanical challenges (signal integrity) associated with exchanging data between two chips: i.e., not allowing bit rates per pin or trace to increase at a fast enough rate.

"Double data rate (DDR) memory has emerged as the dominant memory technology (in terms of number of units sold)."

Memory Fundamentals

In this article we focus on the memory in microprocessor-based systems. More precisely, we focus on memory that is outside of the CPU, that is, we do not focus on on-die caches, or on non-volatile storage. This memory outside of the CPU is circled in the simple drawing in Figure 1.

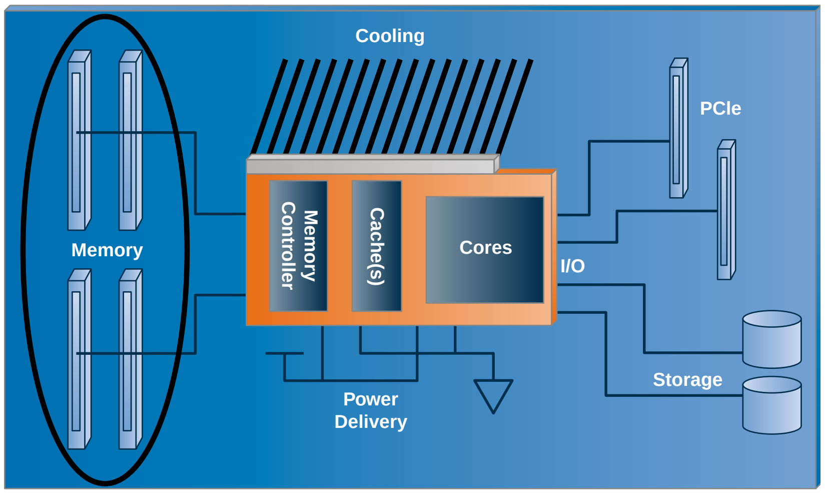


Figure 1: System Block Diagram
Source: Intel Corporation, 2009

Key Metrics

The key metrics for memory sub-systems are bandwidth, capacity, latency, power, system volume, and cost. We examine these next.

Bandwidth. Bandwidth is defined as bytes moved between the memory and the CPU's memory controller. Bandwidth is usually the most talked-about performance parameter. The bandwidth required for a system is usually market segment and application (working set size, code arrangement, and structure) dependent. Amdahl's rule of Thumb [3] suggests there should be one Bps for each instruction per second executed. This ratio is sometimes converted to one Bps, per floating point operation, per second. Designing and implementing that high a ratio of memory bandwidth to CPU performance has not been met for most general-purpose computer systems today.

"The focus is usually on read latency. Write latency can be posted (put in a queue) and is therefore considered less important."

Capacity. The capacity is the total number of bytes stored. A weak correlation exists between the capacity of memory to performance. There is a large range for memory capacity compared to bandwidth to memory. When comparing capacity (bytes) to bandwidth (Bps) in a ratio, the range goes from very small (0.10 to large (10).

Latency. This is the time it takes to read a word from memory. The focus is usually on read latency. Write latency can be posted (put in a queue) and is therefore considered less important. The latencies to DRAM devices have been reduced slowly over the last decade.

Power (energy per time). Power equals the energy consumed divided by the time in which that energy is consumed. Sometimes power is a more useful metric than energy per bit moved. In effect, to be accurate in assessing the power and energy efficiency of memory, both metrics are useful. Although the majority of the power consumed is often dependent on how much data are moved, power is also consumed that is not directly dependent on the data read and written. Lastly, the power and energy per bit moved for memory are also dependent on how the DRAM devices are physically connected to the memory controller.

"Volume reductions must be accompanied with commensurate power reduction so that power densities are not increased."

System volume. This is the space taken up by the DRAM chips (often called DRAM devices). It is common for DRAM devices to be mounted on a small printed circuit board, called dual inline memory modules (DIMMs). Therefore, system volume is sometimes measured in mm^3. Board area (mm^2) is often considered, and although technically not a volume, board area is often lumped into the "volume" category. Volume reductions must be accompanied with commensurate power reduction so that power densities are not increased.

Cost. The cost refers to the money it costs to implement the memory sub-system of choice.

Memory Sub-system Scaling

Most DRAM processes are very efficient at implementing small-sized trench capacitors in their substrate, as well as low leakage transistors. These processes tend to favor fast n-channel transistors used as charge switches. DRAMs often implement the p-channel transistors to be slow (longer switching time) relative to the n-channel transistors. Therefore, a DRAM process is not good for general logic functions when compared to the processes used to implement CPUs. Because of these choices, the evolution of DRAM has seen a high percentage of logic functions managed in the DRAM controller, not the DRAM itself. These choices, coupled with process shrinks, have resulted in an impressive reduction in per-memory bit cost.

"The evolution of DRAM has seen a high percentage of logic functions managed in the DRAM controller, not the DRAM itself."

As CPU performance has increased, the bit rate per pin or bit rate per bump (depending on the CPU and packaging technology) has also increased. Unfortunately for system designers, the ability to increase performance within a chip has increased much faster than the ability to increase the bit rate per pin, bit rate per bump, and bit rate per trace. The result has been a reduction in the number of DIMMs that can be attached to a memory channel while still maintaining good signal integrity. Although these challenges and limitations may be well understood, they have proven to be difficult to solve within the current cost constraints of most systems.

"Volume microprocessor-based systems have demanded an increase of two times per two years in off-chip bandwidth."

To summarize, memory sub-systems are mostly constrained by four main factors: bandwidth, power, capacity, and cost. System main memory is typically implemented with commodity DRAM. The bandwidth per pin is not increasing as quickly as the compute capability of CPUs: this imbalance causes an increase in CPU pin count, and it either adds cost, or reduces the bandwidth per compute operation, and decreases the system performance. The capacity-dependent power for DRAM, and the energy per bit which is read and written for DRAM, are not decreasing as quickly as the compute capability of the CPUs is increasing. Tera-scale CPUs increase performance by allowing many threads to run simultaneously. Concurrently running threads are expected to maintain or increase the demand for bandwidth beyond the traditional twice-every-two-year curve [4].

Evolutionary Memory Scaling

The interface to dual inline memory modules and the architecture of double data rate memory do not scale well for future tera-scale CPUs. The following sections address the areas where scaling to the future is problematic.

Bandwidth

Traditionally, volume microprocessor-based systems have demanded an increase of two times per two years in off-chip bandwidth. This trend of increased bandwidth demand is based on the bandwidth demand of a single microprocessor core. As we add cores to the microprocessor, and increase the parallel applications running on those cores, the bandwidth demand will increase beyond this traditional trend. The increased demand has historically been met by using commodity DRAM (DDRx) technology, and in some high-end applications, by using specialized memory such as graphics DRAM (GDDRx). The bandwidth supplied by these devices is limited by the width of the external interface, the pin speed of that interface, and the cycle time and banking of the DRAM itself.

The cycle time of the DRAM is limited by both the DRAM process technology and by the architecture of the chip, i.e., the size of the physical arrays and the parasitics associated with them. DRAM technology has been optimized for capacity, not for bandwidth. The transistors are designed for low-leakage power, and the metal stack is composed of two to three layers that are optimized for density. The cycle time for large arrays is thus limited by metal parasitics and by the transistors (optimized for DRAM) that are used for control logic and the datapath. Decreasing the sub array size to reduce these parasitics, as is typically done for GDDRx, means replicating peripheral circuits, such as decoders and sense amps, and increasing the wiring required to get the data from the increased number of sub-arrays off of the chip. These factors decrease the memory density, thereby increasing the cost per bit, while also increasing power consumption.

"DRAM technology has been optimized for capacity, not for bandwidth."

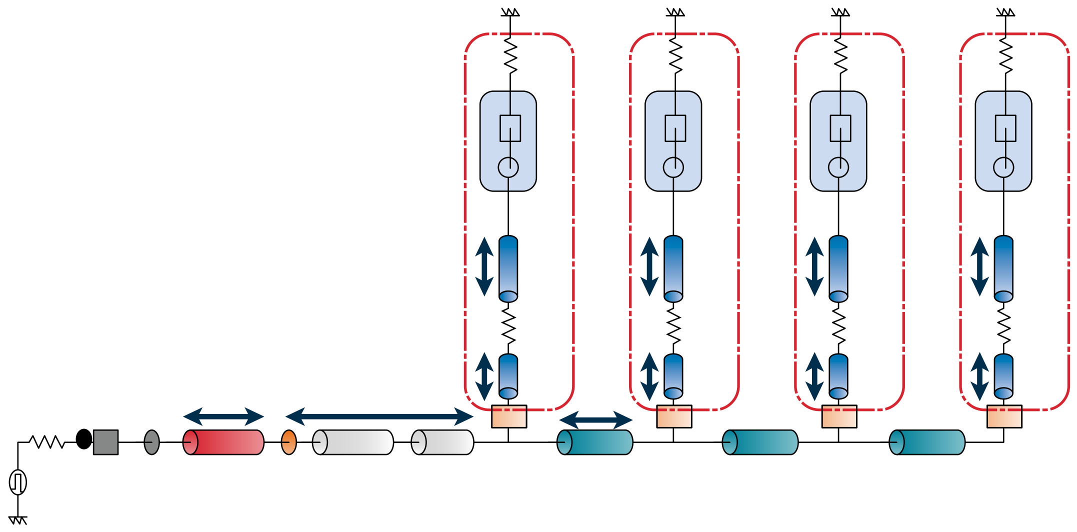


Figure 2: DDR2 Four-DIMM Topology
Source: Intel Corporation, 2009

The pin speed of the external interface is limited by the performance of the low-leakage transistors available as well as by the configuration of the physical components in the path between the memory controller and memory packages, connectors, sockets, wires, etc. The width of the interface is limited by package size and cost, as well as by the interconnect components, such as module connectors. A major limitation in pin speed for DDRx solutions is the physical configuration. Packaged DRAM devices are placed on DIMMs. Multiple DIMMs are then connected to a common set of wires going to a memory controller. This common set of wires forms a DRAM "channel." DIMMs are connected to this channel by inserting the DIMMs into edge connectors on a motherboard. This "multi-drop" style of connection creates an electrical discontinuity, or impedance change, at each DIMM connector on the common motherboard wires. Such a design limits the overall speed of the wires. An electrical illustration of this type of connection is shown for a DDR2 four-DIMM topology in Figure 2.

Figure 3 shows a "pulse response," or representation, of the distortion introduced by propagating a voltage pulse representing one bit of data across this interconnect.

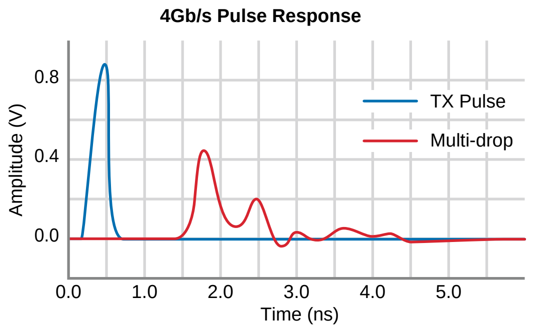


Figure 3: Pulse Response
Source: Intel Corporation, 2009

In order to increase the pin speed, the number of DIMMs connected to the channel must decrease in each generation. DDR2 has four DIMMs while DDR3 data rates often allow just two: a future DDR4 will likely be limited to one DIMM per channel, and eventually only have a single "rank" of memory on that DIMM, meaning there will be a DRAM device on only one side of the DIMM. Once we move to one DIMM per channel in DDR4, the pin speed will be capped at 2.5-3.5 Gigabits per second (Gbps) per pin, due to the signaling scheme used and the physical channel components. At that point, DDRx bandwidth scaling via pin speed will cease. This will correspond to a bandwidth of 24 Gigabytes per second (GBps) at a 3 Gbps pin speed for a 64-bit-wide interface. Adding more bandwidth then means adding memory pins and channels to the system that will be limited by component package sizes, pin counts, and DIMM connector pin counts.

"DDR2 has four DIMMs while DDR3 data rates often allow just two: a future DDR4 will likely be limited to one DIMM per channel."

In order to get more bandwidth without adding channels, we then must move to GDDRx-type topologies. In this technology design, the memory device is soldered to the motherboard, and there is one chip or set of chips per channel, thus removing the DIMM-related discontinuity. However, this severely limits capacity, since the multiple DIMM, multiple ranks per DIMM scheme, is now gone. To improve the cycle time of the memory, the sub-array size is decreased, again trading reduced capacity for increased bandwidth. This design will provide a solution for a pin speed increase of up to 5 – 6 Gbps, at which time DRAM process technology and package parasitics will prevent further increases. Exotic solutions such as the so-called XDR scheme proposed by RAMBUS Corporation would involve a two-pin per signal, or differential scheme, to increase the pin speed further. To justify the extra pins, pins whose speed is twice that of GDDRx would be necessary. Such a design will severely push the capability of DRAM process technology and result in unacceptable increases in power and perhaps cost.

Power

DRAM power is composed of three main components: power consumed by the storage array, power consumed by the peripheral circuits, and power consumed by the datapath from the array to the I/O pins. Approximately 50 percent of the power is in the datapath, with the other 50 percent split between peripheral circuits and the array. Historical power trends for DDRx-based systems show overall power to be between 40 – 200 milliwatts (mW) per Gbps. This number is dependent on both the particular type of memory used, DDRx or GDDRx, for example; the power supply; process technology nodes; the physical interconnect channel configuration; and the memory usage model. Smaller sub-arrays are more power efficient at the expense of capacity. This makes GDDRx, for example, two to three times more power efficient than DDRx, but with a substantial density penalty.

"To keep the datapath relatively narrow and save external pins, the datapath becomes the highest speed portion of the design, thus consuming 50 percent of the device power."

As mentioned previously, DRAM process technology is not amenable to high-speed functions. In order to keep the datapath relatively narrow and save external pins, the datapath becomes the highest speed portion of the design, thus consuming 50 percent of the device power.

The physical channel for GDDRx, which connects to one device per channel, is different than the channel for fully buffered DIMMS, which have added active devices on the DIMM to form connections to other DIMMs in a "point-to-point" fashion. This design configuration is the least power efficient of all configurations, due to the added power of the added active device.

Usage policies such as "page open" or "page closed" have a large impact on power efficiency, depending on the "activation efficiency" of the device, or on the amount of data actually used when a row of the memory array is "activated," or read to the sense amplifiers in preparation for making the data available to send to the external memory pins.

As a quick case study, assume that a future high-end system needs a terabyte per second (TBps) of external bandwidth. This corresponds to 8 terabits per second (Tbps) of bandwidth. At 40 mW per Gbps this memory sub-system then consumes 320 Watts. At 200 mW per Gbps, it consumes 1600 Watts. These efficiencies will scale somewhat with further decreases in DRAM power supply, but beyond 1.2 Volts (currently at 1.5 Volts), this will become very difficult to reduce further, due to process technology and circuit constraints. Clearly, an evolutionary solution to providing high bandwidth will become a show-stopper due to power.

Capacity

We have already discussed the tradeoff for bandwidth versus capacity in traditional memory sub-systems. One solution is fully buffered DIMMs, which, as previously discussed, will be limited by power. Interim solutions, such as a "buffer on board," will help in the short term by placing a single buffering component on the motherboard rather than on the DIMMS, but will still be limited, due to both expandability and power constraints.

"When measured in number of processor cycles, the latency has been increasing."

Latency

Latency is trending down somewhat in absolute terms with process technology improvements, but when measured in number of processor cycles, the latency has been increasing. Latency improvements (reductions) would come from reducing the size of the memory sub-array to limit on-chip wire parasitics, and from moving the memory as close to the memory controller as possible to limit external wire length. Both of these solutions are limited in effectiveness and/or trade capacity for latency. The memory controller, due to its complex association with rows, columns, pages, and ranks of DDR devices has the largest impact on latency in the system. If the architecture of DRAM devices does not change, it will be very difficult to reduce the complexity of the memory controllers. Therefore, it will be very difficult to reduce the latency to read from and write to memory while maintaining the existing cost per bit and existing bandwidths.

Evolutionary Summary

In summary, the key trends for evolutionary memory sub-system scaling are these:

- Bandwidth scaling for traditional DDRx-based systems will end at about 24 GBps for a single channel.
- To get this bandwidth, capacity per channel will be limited to one DIMM without extra components, such as buffer on board (motherboard).
- GDDRx gives increased bandwidth but at the cost of capacity. Pin speed will be limited to 5 – 6 Gbps for GDDR channels being constructed today.
- Power in the memory sub-system varies from 40 – 200 mW per Gbps, translating to hundreds of Watts for a TBps of bandwidth.
- Adding capacity to evolutionary memory sub-systems is limited to adding channels, fully buffered DIMMs, or putting a buffer on the motherboard. All of these add cost and power to the system. Fully buffered DIMM memory is the least power efficient of all DRAM memory technologies.
- Latency improvements for evolutionary systems will be minimal.

Tera-scale Memory Challenges

"DRAM technology will continue to be the solution of choice in these applications for the foreseeable future."

Tera-scale CPUs put additional stress on the memory sub-system and the technologies used to implement them. In the following section, we describe some of those challenges.

Memory Technology

As we look forward to the era of tera-scale computing, the first question we need to ask is which memory technology(s) will fill the needs of these systems. DRAM technology has long dominated the market for off-chip memory bandwidth solutions in computing systems. While non-volatile memory technologies such as NAND Flash and Phase Change Memory are vying for a share of this market, they are at a disadvantage with respect to bandwidth, latency, and power. Given this, DRAM technology will continue to be the solution of choice in these applications for the foreseeable future. We, therefore, discuss future DRAM memories.

We discussed basic DRAM technology earlier in this article. Since the technology is fundamentally dependent on a fixed amount of energy storage on a capacitor, density scaling will become problematic in DRAM technology. At that point, 3-D technology will become a viable path for further scaling.

Key Metric Equals Bytes per FLOP

The key metric for the performance of memory sub-systems is bandwidth (Bps) per computational performance. Performance is usually measured as floating point operations per second (FLOPs per second), sometimes instructions executed per second, and [rarely] clock frequency (cycles per second). Typically, the seconds unit in both the denominator and the numerator are removed, and the metric is expressed as bytes per FLOP. Available bandwidth within a platform has increased twice every two years for many years. Multi-core technology continues to increase the bandwidth demand of CPUs, while at the same time, the available bandwidth as well as the capability of external memory sub-systems are increasing at a lower rate, due to increased DRAM limitations described earlier.

"Available bandwidth within a platform has increased twice every two years for many years."

Future Memory Sub-systems

Changing or fixing one of the problems with bandwidth will not alter the fundamental challenges we face with memory sub-systems. We need a holistic approach to achieve the required results. The main factors that will need to be addressed to achieve the optimal solution for increased bandwidth and lower energy per bit of future tera-scale memory sub-systems are the channel materials, the IO density, the memory density, and the memory device architecture. We examine the changes required in each of these areas.

Channel Materials

First we look at the materials that could be used to construct channels between CPUs and memory modules.

Typically, in order to increase rates as much as possible at the longer lengths, complexity is added to the I/O solution in the form of additional equalization, more complex clocking circuits, and possibly, data coding. These added features increase the power consumed by the I/O solution. More complex interconnects, such as flex cabling, improved board materials, such as Rogers or high-density interconnect (HDI), and eventually, optical solutions, must be considered. These features can increase the cost of the I/O solution. To reduce the impact of the increased cost of the channel, the size of the channel should be reduced, and this has additional electrical signaling benefits. In order to minimize both power and cost, locality in the data movement should be exploited to the greatest possible extent. Exploiting locality of data movement leads to the next desired characteristic in our future memory solutions: I/O density.

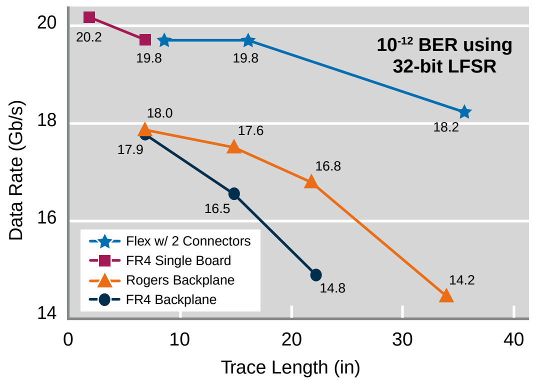


Figure 4: Data Rate vs. Trace Length for Different Materials
Source: Intel Corporation, 2009

I/O Density

In looking at the desired characteristics for future memory sub-systems that require very high bandwidth, it is instructive to divide the challenge of building a high I/O density solution into two parts: I/O and memory core components. We first look at the I/O solution. In Figure 5, we plot the power efficiency in mW per Gbps of various I/O options, including DDR3, GDDR5, on-package I/O, two experimental Intel interfaces, and an interface that may be enabled through short, dense interconnects.

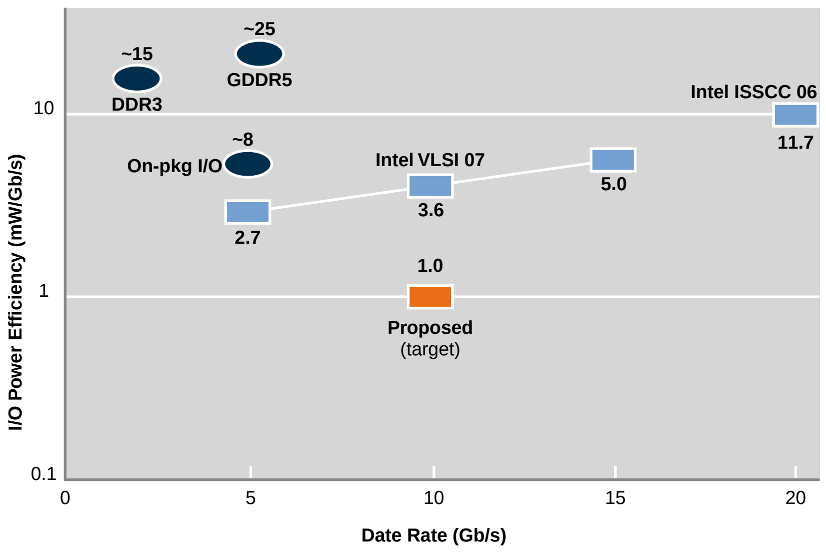


Figure 5: Power Efficiency of Various I/O Options Plotted in mW per Gbps
Source: Intel Corporation, 2009

Note that the traditional memory interfaces achieve power efficiencies that vary from 15 mW/Gbps to 20mW/Gbps. These numbers are dependent on the physical channel between the CPU's memory controller and the memory, the style of the I/O, and the rate at which the I/O is running. Typically, the faster the I/O, the less power efficient it will be, as the process technology is pushed harder and features are added to the I/O solution to increase speed. The Intel research presented at the International Solid State Circuits Conference in 2006 [5] is an interface built to achieve 20 Gbps on 90 nm CMOS. The interface labeled "Intel VLSI 07" [6] in Figure 5, on the other hand, was built to operate across a wide performance range with optimum power efficiency at all performance points. Note that, for that interface, the power efficiency does indeed improve at lower data rates, 2.7 mW per Gbps at 5 Gbps, versus 15 mW per Gbps at 15 Gbps. This improvement is achieved by optimizing both the I/O circuits and the power supply for each data rate to achieve the desired bit error rate on the interface, given the non-linear characteristic of reduced power efficiency as the bit rate increases. By utilizing shorter, more optimized interconnects, we can potentially further improve this style of interface to achieve approximately 1 mW per Gbps at 10 Gbps. Note that this is approximately an order of magnitude better than the I/O solution on traditional memory technologies.

From the data given, we can draw the conclusion that, in order to minimize the power consumed, any data movement should be across as wide of an interface as possible. For a given aggregate desired bandwidth, this means that we can operate each I/O line at the lowest possible rate, thereby improving the power efficiency. In order to improve power efficiency without increasing the size or form factor of the system or the associated distances that data must move, we need to increase the density of the interconnect of the channel between the memory controller and the memory.

The density of traditional chip-to-chip interconnects is limited by several factors, chief of which, is, of course, the cost of the selected interconnect. Traditional printed circuit board technology is, for example, much less dense than microprocessor package interconnect density as measured by the number of wires per cross-sectional area. Given this knowledge, one obvious way to increase the interconnect density is to mount the memory on the microprocessor package as shown in Figure 6.

Using this configuration, no connections between the CPU and the memory go to the motherboard, and the improved channel material yields the density increase. This configuration helps to solve the problem of moving bits from the microprocessor to the edge of the memory die; however, we also need to explore more efficient methods of moving those bits in and out of the memory array.

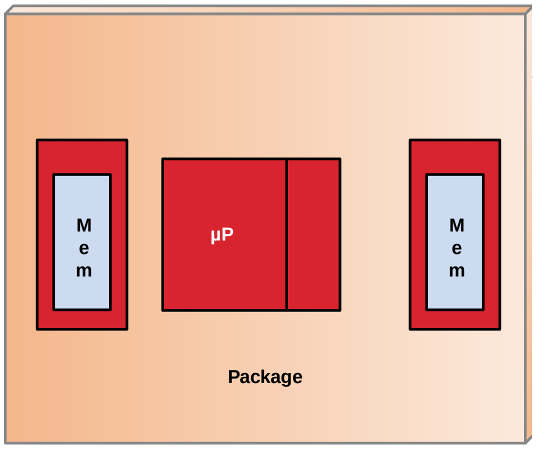


Figure 6: Memory Mounted on Microprocessor Package
Source: Intel Corporation, 2009

Memory Density

Earlier in this article, we discussed the limitations that DRAM technology imposes on high bandwidth solutions, as well as the fact that density scaling may become an issue at some point in the future. We need a technology that will solve both of these issues. 3-D technology, based on through silicon vias (TSVs), offers one such possible solution [7, 8, 9]. 3-D stacked memory will provide an increase in memory density through stacking, and it will enable a wide datapath from the memory to the external pins, relaxing the per-pin bandwidth requirement in the memory array as shown in Figure 7.

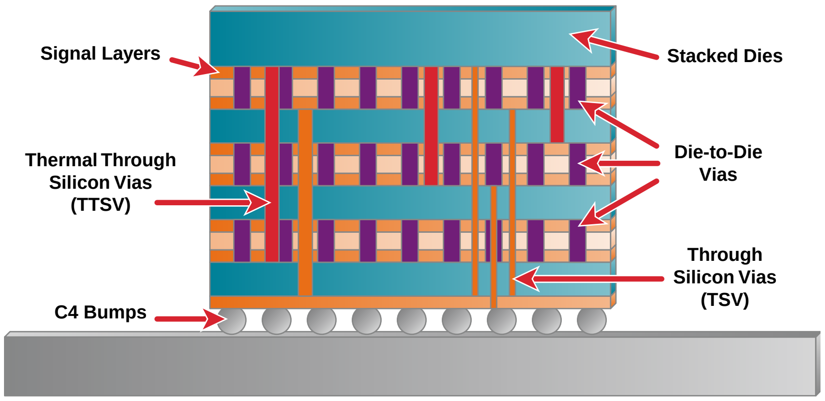

Signal Layers
Stacked Dies
Thermal Through Silicon Vias (TTSV)
Die-to-Die Vias
Through Silicon Vias (TSV)
C4 Bumps

Figure 7: 3-D Stacked Memory Module [7]
Source: Intel Corporation, 2009

This design achieves six objectives:

- We provide a method for further scaling of DRAM density.
- We enable a relatively wide datapath from the memory array to the memory pins, relaxing the speed constraints on the DRAM technology.
- We maintain a high density connection from the memory module to the memory controller, which makes for more efficient use of power. We never connect to low-density motherboards, sockets, or connectors.
- We eliminate many of the traditional interconnect components from the electrical path, including memory controller package vertical path, socket, and memory edge connectors.
- We can separate the high bandwidth I/O solution from the microprocessor and memory controller power delivery path, since we are using the top of the package rather than the bottom to deliver bandwidth.
- The increased density eliminates the need for the electrically-challenged and energy-inefficient, multi-drop DIMM bus. The new stacked memory will be seen as a single load device.

Finally, we need a way to move the data from the wide datapath from the memory array to the memory device pins. There are several possible ways to move the data: the general characteristics necessary for an optimal solution are the ability to efficiently multiplex the data at a rate that matches the data rate of the increased device pins (Gbps), rather than a rate that matches the slower, wider memory datapath, at an efficient energy level (picojoules per bit) that closely matches the characteristics of the CPU generating the memory requests.

Now that we have an efficient method of moving the bits, and a means to scale device capacity, we look at what we can do with the memory device architecture.

"RAS/CAS provides a good set of choices when low-cost per DRAM bit is the highest priority."

Memory Device Architecture

DDR-based DRAM devices use an architecture of rows of bits that are activated via a row address strobe (RAS). This causes small charges stored on rows of capacitors to pull up or down a bit bus that is attached to sense amplifiers. These sense amplifiers amplify the small voltage from the memory bit storage capacitor into an externally readable voltage and they buffer them in temporary storage. The relatively few bits that are being read or written within that row are then selected via a column address strobe (CAS). This architecture provides a good set of choices when low-cost per DRAM bit is the highest priority. This architecture also allows for low pin-count devices and minimizes the area required for logic and data paths within the die. A simplified drawing of this architecture is shown in Figure 8.

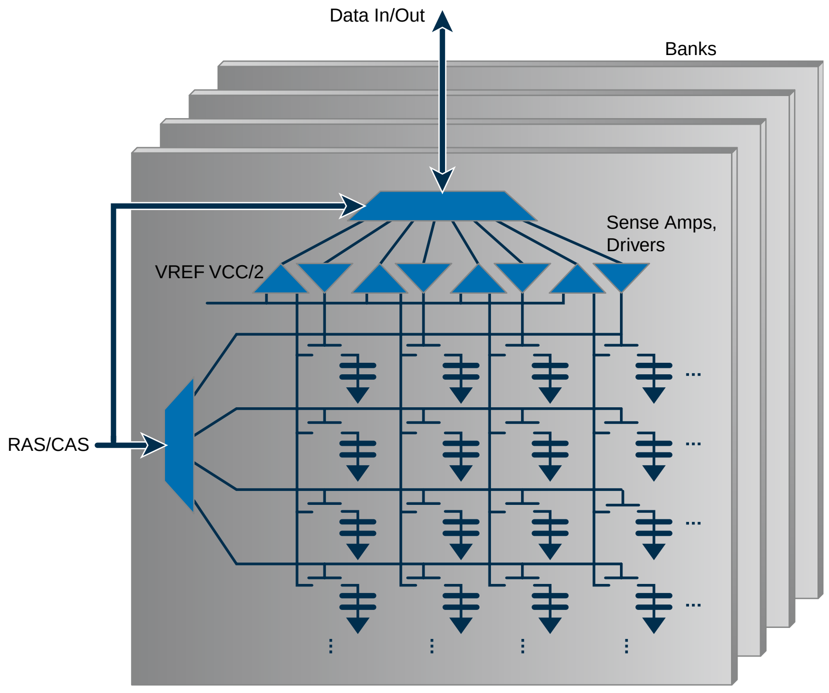


Figure 8: Simplified DRAM Architecture
Source: Intel Corporation, 2009

Unfortunately, this architecture is not conducive to low energy per bit read or written, since energy is used to amplify all of the bits in the entire row even though only a few are read. Therefore energy is wasted on the unused bits. Row access sizes, called pages, are generally in the 1-Kilobyte to 2-Kilobyte range, while the memory requests that activated the page are commonly only 8 bytes, a 128:1 or 256:1 ratio, respectively. Although operating systems and memory controllers try to optimize data placement and access patterns to maximize the reuse of an activated page, most of the data in an activated page are not used. To make matters worse, as the number of cores in CPUs increases and more threads are executing concurrently, the address locality of data accesses, and therefore the ability to maximize activated page reuse, is decreasing. This will lead to further wasted energy in the memory sub-system.

As DRAM technology has advanced and capacity per device has increased over the last 15 years, the page size and therefore activation energy has not been reduced as quickly as the number of bits has increased. Adding banks enables multiple rows to be activated and held open while other banks are being accessed. This feature has an additional benefit of reducing the need for consecutive spatial locality of accesses to reuse an activated page, but it does little to address the overall ratio of activated energy to request size.

"As the number of cores.. increases.. and more threads are executing concurrently, the address locality… is decreasing."

Reducing the energy of the memory device architecture will require a reduction in the size of the pages to reduce the activation energy per request. Various page sizes must be investigated against many workloads. To further mitigate the impact of the loss of spatial locality, due to advanced multi-threaded, multi-core, tera-scale CPUs, the number of banks can be greatly increased, which should also simplify the routing of the bit lines and sense amplifiers in the memory device [8].

TSVs, as discussed previously, can be used to route many pages from a large number of banks within the DRAM memory array to the device pins and connect them to the improved channel between the DRAM memory module and the tera-scale CPU. A discussion of how proposed memory array architecture is implemented and routed with TSVs to the device pins is beyond the scope of this article, and, moreover, is dependent on the individual technology developed by the memory array manufacturers; however, the individual technologies discussed in this article can be combined to achieve an overall improved memory device architecture.

Memory Hierarchy

"A new memory hierarchy could provide increased performance, low power, or both increased performance and lower power, within a set cost constraint."

Given a memory of the type we describe, we must also examine the entire memory hierarchy. For example, it may be advantageous to add a level of memory to the hierarchy. We may have some amount of high-bandwidth memory, while the rest of the memory capacity has a low-bandwidth requirement. There will be cost, performance, and power tradeoffs when deciding how the high-bandwidth memory is connected to the CPU. The same is true for the lower bandwidth, higher capacity of the slower "bulk" memory. Since the majority of the memory is low bandwidth, we have several options for connecting it to the CPU, such as connecting the memory to the printed circuit board and using copper traces.

The reasons for inserting an additional level in the memory hierarchy are the same as those for looking into new memory technologies: performance, power, and cost. More precisely, a new memory hierarchy could provide increased performance, low power, or both increased performance and lower power, within a set cost constraint. Equally appealing is reducing cost but maintaining the same performance and power consumption levels, something that may also be possible with the addition of another level of memory hierarchy.

Analyzing different memory hierarchies is a huge challenge. All the metrics mentioned previously need to be evaluated in the context of the applications of interest (see "Key Metrics"). Adding to the challenge is anticipating the effect tera-scale CPUs will have on the memory traffic to and from memory.

The techniques to estimate the performance of memory are often broken into cache management and memory management. For comparison purposes, we choose the following starting point.

- Caches are on the same CPU die as the processing cores.
- The capacity of the caches is in the range of 1 – 4 Megabytes per core for the last level of cache (if there is more than one level of cache present).
- The unloaded latency to read from a cache is in the range of 10–30 nanoseconds (ns).
- The replacement policy of the cache is managed by an algorithm managed by hardware in the CPU. Cache lines are the unit of transfer (size of data that is placed and replaced). The cache line size (number of bytes) is dependent on the architecture of the core. Most current CPUs usually have two or more levels of cache as well as a (very low-latency) register file per core. In a thorough analysis of memory sub-system performance, these features would have to be considered.
- The main memory is DRAM, and it is on separate chips from the CPU.
- The range of the capacity of main memory varied greatly from system to system, with the average range within an order of magnitude of 1 Gigabyte per core.
- The unloaded latency to read from main memory is on the order of magnitude of 50 ns.
- The "replacement" policy is pages, managed by the operating system (OS). The size of pages varies; it is commonly 4 Kilobytes or larger.

When considering additional levels of the memory hierarchy, the key decisions are where to add a level or levels in the memory hierarchy and how the levels of memory are managed.

"The lower the latency to memory, the higher the energy per bit read and written."

Memory Hierarchy — Where to Add Memory

When designing a memory hierarchy, we use the following guidelines. Looking out from a CPU, the closer (lower) level of memory must have the lowest latency and the highest bandwidth, and it can have the lowest capacity. As the levels of memory increase, the latency increases; the bandwidth decreases while the capacity increases. Hidden within these guidelines is the fact that the lower the latency to the memory, the higher the cost per bit. Also, the lower the latency to the memory, the higher the energy per bit read and written.

Earlier, we concluded that to meet the needs of tera-scale systems, designers should investigate new architectures and manufacturing techniques for DRAM, with an emphasis on 3-D stacking with TSVs. We are confident that these techniques will lead to improved DRAM products, while maintaining a low cost per bit stored. We also realize that when the new technologies are introduced, it will take time for the price per bit to drop. Therefore, early use of 3-D stacked memory as near memory, backed up by DDR-based DRAM or other low cost per bit memory technologies, may be an appealing and cost-effective choice for designers.

Memory Hierarchy — How Layers are Managed

Usage and management policies for near memory and where these policies are implemented are difficult research questions to answer. We chose to separate these questions into two procedures: manage the near memory as a cache or present the memory to the OS as two regions of memory with different performance and energy characteristics.

"Using policies implemented in hardware to manage the near memory as a cache is appealing, because this kind of management will cause the minimum disruption to application and OS software."

Using policies implemented in hardware to manage the near memory as a cache is appealing, because this kind of management will cause the minimum disruption to application and OS software. The effectiveness of this management scheme will depend on how well the replacement policy leads to a high hit rate into the near memory. A high hit rate is essential to make this memory hierarchy meet the performance-to-cost goal. Target application data set size and memory access patterns, and how those interact with the replacement policy, will have a large impact on the hit rate. Other factors that will come into play are how the near memory is arranged: associatively and number of ways (if more than one way in the cache). Adding to the challenge is that the size of the unit of data that is replaced also affects the performance and complexity (a form of cost) of the memory sub-system. As the associativity increases, the size of the tags increase. As the size of the replacement unit decreases, the hit rate should increase, but the number of tags also increases.

The other extreme for a near and far memory management choice is to provide mechanisms in hardware to move data to and from either range of memory, and to leave the replacement policy to software. Due to the design complexity, we expect that the majority of people writing application code and OS code will not have enough incentive to tackle this problem for many years. Tera-scale computing will provide an increasing number of threads that are available for running applications. Effectively using and coordinating those threads will require that applications and the OS's that those application run on evolve further. Code that is specialized for a lightweight OS and written to specific hardware will be the only code that can take advantage of two-level memory sub-systems managed by software.

Power and Energy Considerations

As the hierarchy of the memory sub-system is examined, more emphasis must be placed on the power (energy used per time) that is consumed to run the applications. The simple statement that data movement must be minimized will take on additional importance as tera-scale CPUs are built. For example, if a memory sub-system is built with near and far memory, the near memory will consume a low amount of energy to read and write bits, as well as lower latency, higher bandwidth, and less capacity than those required by far memory. The bandwidth to and from the near memory must be considerably higher than the bandwidth to and from the far memory, but the cost per bit of the far memory must be lower than the cost per bit of the near memory. This creates an additional incentive for designers to choose replacement policies for the near memory, such that the hit rate is high and the eviction rate is low.

Summary and Conclusions

Tera-scale CPUs will demand a large amount of bandwidth from their memory sub-systems. Their demand for memory bandwidth will exceed one TBps. Current DRAM architectures and the DDR-based interfaces to those chips will not meet the needs of tera-scale CPUs. There is no simple solution to the challenges associated with bandwidth to memory; many aspects of the memory sub-system will require further research. Three key metrics have been identified that need improvement: bandwidth, power, and cost. There are strong dependencies between these metrics.

Increasing the bit rate per pin conflicts with lowering the power of the memory sub-system. I/O circuits have demonstrated impressively high bit rates. However, I/O circuits achieve higher power efficiency (lower mW per Gbps) at slower data rates. Therefore, to achieve high-bandwidth interfaces that are power efficient, wide slow interfaces need to be developed. To support those interfaces, new packaging and new materials will need to be used. In order to keep costs down, the use of these new technologies needs to be localized to small areas.

The use of TSVs as a third dimension of connection between DRAM chips is a promising technology to increase the capacity of DRAM products and reduce power consumption. TSVs effectively provide many more I/O connection points into and out of the DRAM chips. These connection points allow for different ways to access and manage the bit storage arrays that can lead to more accesses in parallel, thereby improving performance. TSVs also allow for different access groupings within the DRAM chips that can reduce the power consumed.

The introduction of different architectures of DRAM chips creates questions of what memory sub-systems will look like in the future. As new types of memory are built, the memory sub-systems will evolve to use the new memory products in ways that best meet the market segment's performance, power, and cost requirements. Not all memory sub-systems will look the same.

"Tera-scale CPUs will demand a large amount of bandwidth from their memory sub-systems. Their demand for memory bandwidth will exceed one TBps. Current DRAM architectures and the DDR-based interfaces to those chips will not meet the needs of tera-scale CPUs."

References

[1] Vangal, Sriram et al. "An 80-Tile 1.28 TFLOPs Network-on-Chip in 65nm CMOS." In *IEEE International Solid-State Circuits Digital Technology Papers,* February 2007, page 98.

[2] Moore, G. "Cramming more components onto integrated circuits." *Electronics Magazine*, 19 April 1965.

[3] Gene Amdahl. "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities." In *Proceedings of AFIPS Conference*, (30), pages 483-485, 1967.

[4] Brian Rogers, Anil Krishna, Gordon Bell, Ken Vu, Xiaowei Jiangy, Yan Solihin. "Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling," *ISCA'09*, June 20–24, 2009.

[5] B. Casper, et al. "A 20Gb/s forwarded clock transceiver in 90nm CMOS." *ISSCC 2006 Digest of Technology Papers,* pages 90-91, February 2006.

[6] G. Balamurugan, J. Kennedy , G. Banerjee, J. Jaussi, M. Mansuri, F. O'Mahony, B. Casper and R. Mooney. "A scalable 5–15 Gbps, 14–75 mW low power I/O transceiver in 65 nm CMOS." *Symposium VLSI Circuits Digital Technology Papers,* 2007, page 270. Also published in *IEEE Journal of Solid-State Circuits,* vol. 43, pages 1010-1019, April 2008.

[7] Deshpande, A., Natarajan, V. and Methani, J. "Pareto-Optimal Orientations for 3-D Stacking of Identical Dies." *10th Electronics Packaging Technology Conference, IEEE,* 2008, pages 193–199.

[8] Loh, G. "3D-Stacked Memory Architectures for Multi-core Processors." *35th ACM International Symposium on Computer Architecture (ISCA)*, pages 453-464. June 21-25, 2008. Beijing, China.

[9] Black, B., Annavaram, M., Brekelbaum, N., DeVale, J., Jiang, L., Loh, G., McCauley, D., Morrow, P., Nelson, D., Pantuso, D., Reed, P., Rupley, J., Shankar, S., Shen, J. and Webb, C. "Die Stacking (3D) Microarchitecture." In *39th IEEE/ACM International Symposium On Microarchitecture,* (Micro). pages 469-479.

[10] D. Burger, J. R. Goodman, and A. Kagi. "Memory bandwidth limitations of future microprocessors." In *ISCA '96: 23rd Annual International Symposium on Computer Architecture,* pages 78-89, New York, NY, USA, 1996. ACM.

[11] R. Kumar, D. Tullsen, N. Jouppi, and P. Ranganathan. "Heterogeneous chip multiprocessors." *Computer,* 38(11):32(38), 2005.

[12] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. "IBM POWER6 microarchitecture." *IBM Journal of Research and Development*.51(6):639-662, 2007.

[13] M. K. Qureshi, M. A. Suleman, and Y. N. Patt. "Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines." In *HPCA '07: 2007 IEEE 13th International Symposium on High Performance Computer Architecture,* pages 250-259, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Y. Solihin, F. Guo, T. R. Puzak, and P. G. Emma. "Practical Cache Performance Modeling for Computer Architects." In *Tutorial with HPCA-13,* 2007.

[15] J. Tseng, H. Yu, S. Nagar, N. Dubey, H. Franke, and P. Pattnaik. "Performance Studies of Commercial Workloads on a Multi-core System." *IEEE 10th International Symposium on Workload Characterization,* 2007. IISWC 2007, pages 57-65.

[16] Polka, L., Kalyanam, H., Hu, G. and Krishnamoorthy, S. "Package Technology to Address the Memory Bandwidth Challenge for Tera-Scale Computing." *Intel Technology Journal, Vol. 11, no.3,* 2007, pages 196-205.

[17] P. G. Emma and E. Kursun. "Is 3D Chip Technology the Next Growth Engine for Performance Improvement?" *IBM Journal of Research & Development.* 52(6):541-552, 2008.

[18] A. Hartstein, V. Srinivasan, T. Puzak, and P. Emma. "On the Nature of Cache Miss Behavior: Is It p2?" In *The Journal of Instruction-Level Parallelism,* volume 10, 2008.

[19] M. D. Hill and M. R. Marty. "Amdahl's law in the multicore era." *IEEE Computer,* 41(7):33-38, 2008.

Acknowledgement

We thank James Jaussi for his contributions to the data provided in this paper.

Authors' Biographies

Dave Dunning is a Senior Principal Engineer within the Systems and Circuits Research Lab within Intel Labs. Dave's career has ranged from parallel supercomputer system architecture to design of full custom CMOS circuits and chips. Dave has participated in high-speed, serial-based, inter-chip designs and high-speed clocking structures. Dave is currently concentrating on energy-efficient memory and memory controllers. He holds a BA degree in Physics from Grinnell College, Grinnell, Iowa, a BS degree in Electrical Engineering from Washington University, St, Louis, Mo. and an MS degree in Electrical Engineering from Portland State University, Portland, Or. Dave joined Intel in 1988. His e-mail is dave.dunning at intel.com.

Randy Mooney is an Intel Fellow and director of I/O research in Intel's Circuits and Systems Research Lab, part of Intel Labs. He joined Intel in 1992 and is responsible for circuit and interconnect research, focusing on solutions for multi-Gbps, chip-to-chip connections on microprocessor platforms. Another current research focus is memory bandwidth solutions for these platforms. He has authored several technical papers and is listed as an inventor or co-inventor on more than 60 patents. Randy previously worked in Intel's former Supercomputer Systems Division, developing interconnect components for parallel processor communications, as well as a method of bidirectional signaling that was used for the interconnect in Intel's teraFLOPs supercomputer. Prior to joining Intel, Randy developed products in bipolar, CMOS, and BiCMOS technologies for Signetics Corporation. He received an MSEE degree from Brigham Young University. His e-mail is randy.mooney at intel.com.

Patrick Stolt joined Intel in 1991. He is a Senior Technology Strategist in Intel Labs focused on Memory, IO, System On a Chip (SOC) and emulation technologies. Over the last 18 years, he has worked as a board and chip design engineer, silicon and system architect, and technology and product strategist in many areas, including in-circuit emulators, server memory chip sets, server memory architectures, front-side bus architecture, small business network Application Specific Standard Part (ASSP) SOC chips, Internet tablets, video processing, and SOC for digital TV and video analytics, cognitive radios and tera-scale memories. Prior to Intel Patrick worked in the Aerospace and Test & Measurement industries. Patrick has five patents issued and several pending. He holds a BSEE degree from the University of Wisconsin-Madison. His e-mail is patrick.stolt at intel.com.

Bryan Casper received his MS degree in Electrical Engineering from Brigham Young University, Provo, UT. He is currently leading the high-speed signaling team of Intel's Circuit Research Lab, based in Hillsboro, Oregon. In 1998, he joined the Performance Microprocessor Division of Intel Corporation and contributed to the development of the Intel® Pentium® 4 and Xeon® processors. Since 2000, he has been a circuit researcher at the Intel Circuit Research Lab, responsible for the research, design, validation, and characterization of high-speed mixed signal circuits and I/O systems. During his time in the Circuit Research Lab he has developed analytical signaling analysis methods, high-speed I/O circuits and architectures, and on-die oscilloscope technology. His e-mail is bryan.k.casper at intel.com.

Copyright

ULTRA-LOW VOLTAGE TECHNOLOGIES FOR ENERGY-EFFICIENT SPECIAL-PURPOSE HARDWARE ACCELERATORS

Contributors

Ram K. Krishnamurthy
Intel Corporation

Himanshu Kaul
Intel Corporation

Index Words

Motion Estimation
Sum of Absolute Difference (SAD)
Ultra-low Voltage
Variation Compensation
Video Encoding

Abstract

Key enablers to achieving tera-scale processor performance under a constant power envelope will be the addition of special-purpose hardware accelerators and their ability to operate at ultra-low supply voltages. Special-purpose hardware accelerators can improve energy efficiency by an order of magnitude, compared to general-purpose cores, for key compute-intensive applications. Performance per Watt increases as the supply voltage is reduced, but the degraded transistor on/off current ratios at the lower supply voltages can limit the minimum operational supply voltage. Circuit solutions for robust operation at ultra-low supply voltages will also need to minimize any performance impact on the nominal supply operation for a highly scalable design.

This article describes ultra-low voltage design techniques and learnings from a video motion estimation engine fabricated in 65 nm CMOS technology. This chip is targeted for special-purpose, on-die acceleration of SAD computation in real-time video encoding workloads on power-constrained mobile microprocessors. Various datapath circuit innovations within the accelerator improve energy efficiency for SAD calculations at nominal supplies, while ultra-low voltage optimizations enable robust circuit operation for further efficiency gains at ultra-low supply voltages, with minimal impact on nominal supply performance. Silicon measurements of the accelerator demonstrate performance of 2.055 GHz at the nominal supply voltage of 1.2 V, with scalable performance of up to 2.4 GHz at 1.4 V, 50° C. Robust, ultra-low voltage, optimized circuits enable operation measured down to 230 mV (sub-threshold). Across this wide range of operational supplies, maximum energy efficiency of 411 GOPS/W or 12.8 macro-block SADs/nJ is achieved by operating the accelerator at a near-threshold voltage of 320 mV, for 23 MHz frequency and 56 μW power consumption. This represents a 9.6X higher efficiency than at the nominal 1.2 V operation.

Introduction

As transistor integration density continues to increase, the number of cores in a microprocessor will also increase to handle higher performance demands. Improvements in future microprocessor energy efficiency will necessitate that not all the additional cores be targeted for general-purpose processing. For specific applications, such as video processing, graphics, encryption, and communication, dedicated hardware accelerators can provide 10 – 100x higher energy efficiency or performance per Watt. Future multi-core processors will need to be heterogeneous so that key applications, that are computationally intensive or inefficiently handled by general-purpose cores, can be processed by the dedicated accelerators to improve the overall energy efficiency of the processor. The range of applications that can be mapped to a particular accelerator is determined by the degree of flexibility or reconfigurability of the accelerator, which also provides a design tradeoff against the achievable energy efficiency. Figure 1 shows the conceptual organization of such a future microprocessor to achieve tera-scale performance within a constant power envelope.

"The degree of flexibility or reconfigurability of the accelerator provides a design tradeoff against the achievable energy efficiency."

The increasing number of video-capture applications in mobile devices motivates the need for energy-efficient compression of raw video data to meet the storage and transmission constraints of mobile platforms. Motion estimation (ME) is the most performance- and power-critical operation in video encoding, and it can benefit from accelerator circuits to increase throughput in a power-efficient manner. Furthermore, encoder specifications have a wide range of requirements for throughput and power to handle a variety of video resolution, frame rate, and application specifications [1-6], resulting in the need for tunable power and performance capabilities for such accelerators. ME algorithms remove inter-frame redundancies to achieve video compression. Similarities between consecutive frames are used to locate movement of pixel blocks in the current frame with respect to the reference frames. Motion vectors for block translations are used to encode the frames, which can be reconstructed by the decoder from the reference frames. The error metric, used for the purposes of block matching by the algorithms, is the SAD between the respective pixel blocks of the two frames.

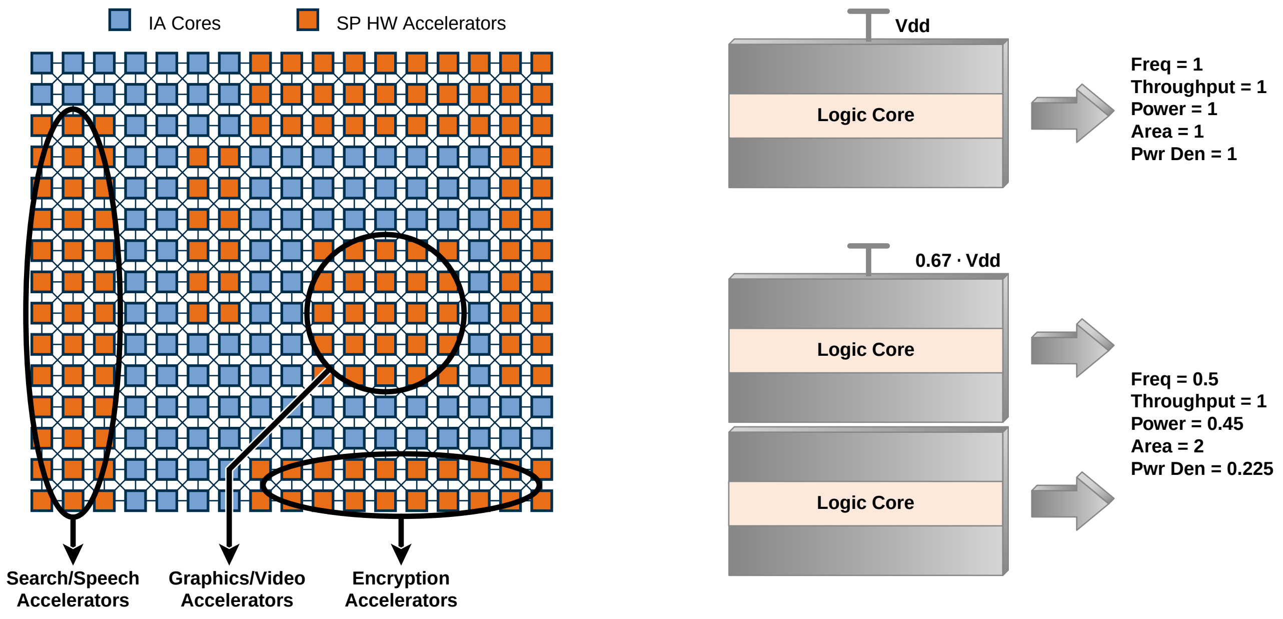


Figure 1: Conceptual Organization of a Tera-scale Processor and Supply Voltage Scaling Impact
Source: Intel Corporation, 2009

"Power consumption decreases by a greater percentage than the corresponding decrease in performance."

Further improvements in energy efficiency can be achieved by lowering the supply voltage of a special-purpose accelerator or core, as power consumption decreases by a greater percentage than the corresponding decrease in performance. Various signal-processing and compute-intensive applications have a high degree of parallelism. Increased hardware-level parallelism for these applications can enable the same throughput at the lower supply voltage, while taking advantage of the increased energy efficiency for lowering overall power consumption. This effect is illustrated in Figure 1: the supply voltage is reduced for a processing core by one-third, for a 50 percent reduction in the normalized frequency. Two cores at the lower supply voltage operate at half the frequency, but they maintain the same throughput as that of a single core at the nominal supply, while consuming less than half the total power. The power-density, which determines thermal hot-spots on the die, is also reduced in this example by more than 75 percent. Further gains in energy efficiency can be obtained by operating at aggressively lower supply voltages.

Processors operating at ultra-low and sub-threshold voltages have been demonstrated for niche applications with low throughput requirements [7-10]. However, the robustness of circuit operation decreases at lower supply voltages, and design optimizations that enable robust functionality at ultra-low supply voltage can severely degrade performance at the nominal voltage. ME accelerators, targeted for integration in high-performance microprocessors, require both high performance at the nominal supply voltages and energy-efficient robust performance in the presence of increased variations at ultra-low supply voltages. Though body biasing and circuit sizing have been used to address variations at sub-threshold supplies [8-10], these techniques can incur considerable area and power penalties. Circuit operation at ultra-low supply voltages also requires level-shifting circuits for interfacing with other processing or memory units that operate at higher supply voltages. These level shifters add power and performance overhead that must be reduced to maximize the energy-efficiency gains from low-voltage operation.

"Design optimizations that enable robust functionality at ultra-low supply voltage can severely degrade performance at the nominal voltage."

A ME engine, targeted for special-purpose, on-die acceleration of SAD computation [11] in real-time video encoding workloads on power-constrained microprocessors, is fabricated in 65 nm CMOS [12]. Intensive SAD computations required for ME block searches benefit from SAD datapath optimizations that improve energy efficiency. The accelerator increases ME energy efficiency with circuit techniques: among these are speculative difference computation with parallel sign generation for SADs, optimal reuse of sum XOR min-terms in static 4:2 compressor carry gates, and distributed accumulation of input carries for efficient negation. Robust ultra-low voltage optimized circuits result in a wide operational range of supply voltages (1.4 V – 230 mV) with minimal performance and power overhead at nominal operation, enabling the accelerator to achieve an optimal energy-delay tradeoff, by dynamically varying operating conditions based on target performance demands and power budgets. Nominal performance (measured at 1.2 V, 50° C) for the accelerator is 2.055 GHz, with 48 mW of power consumption. Performance is scalable up to 2.4 GHz, 82 mW (measured at 1.4 V, 50° C), while the lowest power consumption of 14.4 μW (4.3 MHz performance) is achieved in deep-sub-threshold operations at 230 mV. Ultra-low supply operation enables increased energy efficiency, with maximum energy efficiency of 411 GOPS per Watt (measured at 320 mV, 50° C), for 23 MHz of performance and 56 μW of power consumption. Two-stage cascaded, split-output-level shifter circuits provide energy and area-optimized up-conversion of ultra-low voltage signals to the nominal supply. The accelerator can tolerate up to ±2x process and temperature-induced performance variations at ultra-low supplies by using supply voltage compensation of ±50 mV.

"The accelerator can achieve an optimal energy-delay tradeoff, by dynamically varying operating conditions based on target performance demands and power budgets."

For the remainder of this article, we first describe the accelerator organization and SAD8 circuits; we then describe the circuit designs and optimizations that enable ultra-low voltage operation for achieving higher energy efficiency. We detail the cascaded split-output-level shifter circuit next, and then we provide power, performance, and energy-efficiency measurements on silicon. We end our discussion with measurement results that demonstrate the use of supply compensation to counter large performance variations at ultra-low voltages.

Motion Estimation Accelerator Organization

The motion estimation accelerator (Figure 2) uses SAD calculations to locate a block in the current frame that is minimally different from a block in a reference frame. Pixels from the reference and current frames, stored in local memory, are provided as inputs to the SAD8 unit, which computes the SAD for eight pairs of 8b pixels in a single cycle. The SAD8 unit operates at a lower supply voltage to improve the energy efficiency of the critical computation. This operation is highly parallel for motion estimation, and the SAD8 unit can be scaled accordingly for iterative inter-frame SAD calculations. The output of the SAD8 unit is up-converted to the higher supply by a level shifter circuit. Supply compensation is used to maintain constant performance for the SAD8 unit under increased temperature and process-induced variations at low supply voltages.

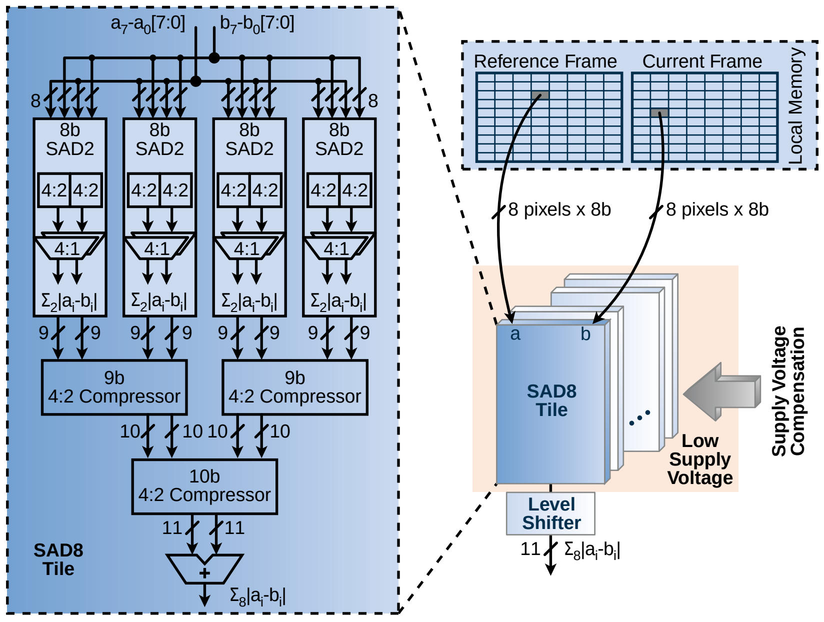


Figure 2: Motion Estimation Accelerator Organization
Source: Intel Corporation, 2009

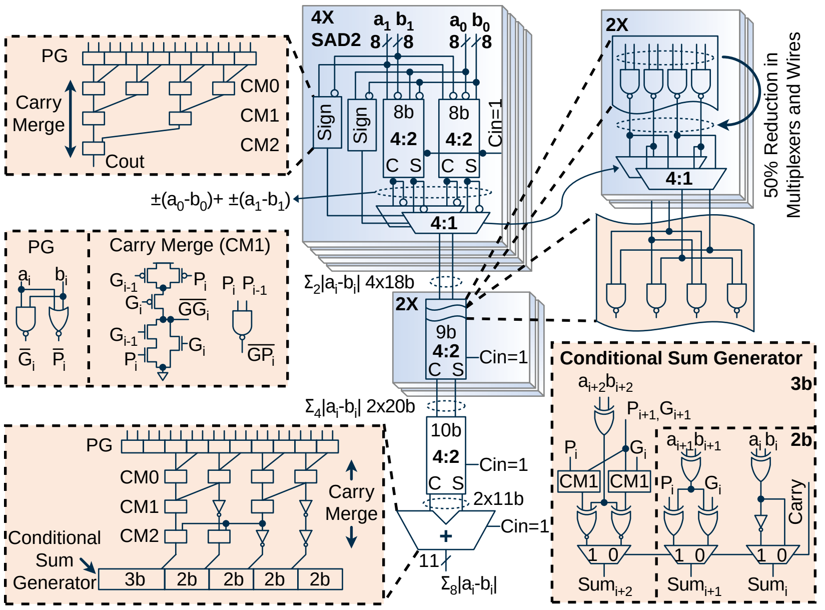


Figure 3: SAD8 Organization and Circuits
Source: Intel Corporation, 2009

Within the SAD8, absolute differences between pixel pairs are computed and summed. Each pixel pair contains one pixel from the reference frame and one pixel from the current frame. The SAD2 unit computes the SAD for two pixel pairs, followed by two stages of compressor units, to provide SAD computation for eight pixel pairs in carry-save format. Absolute difference calculations, as well as summations, are accomplished with 4:2 compressors to increase energy efficiency. A carry propagation adder converts the carry-save SAD output to a normal 11b unsigned binary number, providing the metric for block matching, required by the ME algorithms. The accelerator architecture can also be extended to handle wider SAD computations, e.g., SAD16, SAD32, etc., by increasing the number of compressor stages before the final carry propagating adder.

SAD8 Circuits

Several circuit optimizations improve the performance and energy efficiency of the SAD8 unit (Figure 3). We discuss some of these here.

Speculative SAD2

The SAD is composed of the operations of computing the difference, the absolute value, and the summation of the absolute values, in sequential order. In a conventional SAD2 circuit, the difference between each pixel pair is computed and, based on the sign of this difference, either the true or negated output is selected to obtain the non-negative (absolute) difference. Following this, a summation stage adds the two absolute differences. The requirement that the sign of the difference must be resolved prior to summation results in a dependency that prevents further computation from being completed before the 2:1 multiplexer selects between the positive or negative difference values.

The speculative SAD2 circuit (Figure 3) enables more computation ahead of the sign and before the multiplexer stage. The difference and sum stages are combined to produce the sums of differences, by using 4:2 compressors. Speculation on the signs of differences requires all four possible sums of differences, ±(a0 - b0) + ±(a1 - b1), to be computed before the late-arriving signs, by negating the appropriate inputs. Two of these sums are computed by using a pair of 4:2 compressors, while compressor symmetry is used to obtain the other two sums of differences by inverting the outputs of these 4:2 compressors. The late-arriving signs of the differences select the correct SAD with the 4:1 multiplexer. Summation of differences prior to sign computation and four-way speculation, using only two 4:2 compressors, provides a performance advantage that can be translated to a 30 percent energy reduction compared to a conventional non-speculative design for the same delay; thus, energy efficiency is improved for SAD2 computation.

Efficient Carry Insertion for 2's Complement

The SAD operation negates one input in each pixel pair for the difference computation. Negating a number in 2's complement arithmetic requires a bit-wise inversion of the input, followed by the addition of 1 at the least significant bit (LSB). Irrespective of whether the true or negative output of the difference is selected to obtain the absolute difference, the addition of a 1 is required for every pixel pair, as exactly one input is negated. In conventional designs, this is implemented by setting the carry-in (Cin) for the difference to 1 and selecting the true output of the difference. Selection of the inverted output is the equivalent of inverting all the inputs with Cin=0. In this case, a 1 is added after the multiplexer. Conditional negation to obtain the non-negative difference requires the use of a half adder stage after the multiplexer to insert a conditional carry for every pair, adding to the critical path of the conventional design.

The speculative SAD circuit removes the half adder stage from the critical path. Selection of the inverted output of any of the 4:2 compressors to obtain the SAD in the SAD2 circuit is equivalent to inverting all the inputs and setting the LSB Cin of the compressor to 0. A carry is conditionally inserted in the LSB, after the 4:1 multiplexer, by setting the output carry c[0] to 1 with the use of a parallel 2:1 multiplexer. In a conventional compressor tree this is an unused signal, set to 0. This speculative SAD2 circuit accommodates the addition of a single 1 for every two pixel-pairs and only half the negation carries required for all eight pairs. Carry insertion for the other four pixel-pairs is accomplished by adding a constant 4 in the compressor tree in a distributed manner: this is done by setting the LSB Cin of the three 4:2 compressor units and the final adder to 1. Optimized distribution of all the carries required for negation results in a zero-delay penalty; consequently, the critical path of the SAD8 unit is reduced.

"Optimized distribution of all the carries required for negation results in a zero-delay penalty."

Sign Generation

The computation to determine the sign of the difference is part of the critical path in the SAD2 circuit. The sign-generation circuit uses a propagate-generate (PG) stage followed by a 3-stage, radix-2, logarithmic carry tree for delay reduction. Since the most significant bit (MSB) carry provides the sign, the carry tree is optimized to compute only the carry-out of the 8b difference. The signs of differences are decoded and buffered to provide the 4:1 multiplexer selects within the SAD2. Sign computation of the difference is carried out in parallel with the 4:2 summation, with the path from SAD2 inputs to multiplexer selects being more critical than the 4:2 compressor, by one gate stage.

Compressor

An energy-efficient 4:2 compressor circuit (Figure 4) is a key building block for the SAD8 design, as it is used throughout the datapath for difference calculations and summations. The first stage of sum XORs in the compressor is expanded to two stages of NAND gates, by using differential inputs, to enable reuse of XOR min-terms in the carry logic. The remaining XORs and multiplexer are implemented without buffering or series connected transmission gates, and they provide differential outputs. The differential outputs are directly compatible with next-stage compressor inputs and also provide inverted outputs without any delay penalty. This property enables generation of both true and inverted outputs of the 4:2 compressor pair in the SAD2 circuit, without affecting the critical path. Sharing of min-terms by using the first stage of NAND gates, between the XOR and carry logic, allows the 4:2 compressor design to be optimized further within the SAD8, by relocating these NAND gates in front of the inter-compressor wires. This not only results in improved drive strength for the long wires, as they are driven by NAND gates rather than the transmission gates of the compressor, but also eliminates long differential interconnects, thereby reducing wiring and associated energy by 50 percent. Without this optimization, extra inverters would be required in the critical path to achieve the same result. Also, the first stage of NAND gates of each 4:2 compressor in the second summation stage of the SAD8 is moved behind the multiplexers in each pair of SAD2 circuits (Figure 3). This provides a 50 percent reduction in the number of multiplexers, in addition to the wire reduction benefit.

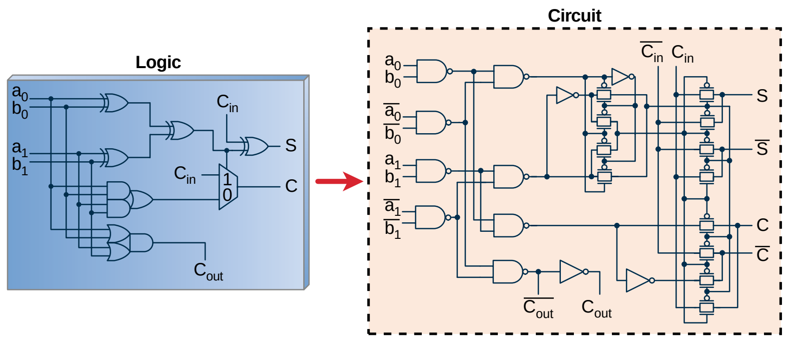


Figure 4: 4:2 Compressor Logic and Circuit Design
Source: Intel Corporation, 2009

Final Adder

Following the compressor tree in the SAD8 circuit, a final 11b adder (Figure 3) converts the SAD from carry-save format to a normal binary number. The adder design uses a radix-2 sparse carry-merge tree that generates carries at alternate bit positions only. The sums are pre-computed by the conditional 2b and 3b sum generators, with the correct sum being selected by the late-arriving carry from the carry-merge tree. This design results in a 50 percent wiring reduction within the carry tree compared to that of a conventional Kogge-Stone adder, while maintaining the same number of gate stages in the critical path.

"This design results in a 50 percent wiring reduction within the carry tree compared to that of a conventional Kogge-Stone adder."

The combined result of the circuit optimizations just described translates into a 20 percent reduction in delay for the SAD8 unit compared to a conventional non-speculative carry-propagating, adder-based implementation. This performance advantage translates to a 24 percent energy-reduction at iso performance.

Ultra-Low Voltage Circuit Optimizations

Though the performance-per-Watt can be significantly improved for parallelizable workloads, such as ME, by lowering the supply voltage, transistor on/off current ratios degrade considerably at ultra-low supply voltages. As a result, DC node voltages drift away from full-rail values, thereby affecting noise margins and reliability across process skews. Circuits such as flip-flops, wide multiplexers, deep-stack logic, and series connected transmission gates have nodes with weak on-current paths and large off-current paths that are more vulnerable to this effect, unless optimized for reliable low-supply operation [7].

The storage nodes in flip-flops have weak keepers and large transmission gates. When the transmission gate for the slave stage of a conventional master-slave flip-flop circuit (Figure 5) is turned off ($\Phi=0$), the weak on-current from the slave-keeper contends with the large off-current through the transmission gate. This causes the node voltage to droop, affecting the stability of the storage node. Ultra-low voltage reliability of the flip-flops can be improved by the use of non-minimum channel length devices in the transmission gates to reduce off-currents exponentially. Furthermore, upsized keepers improve on-currents that restore the charge that is lost due to leakage at this node. The write operation remains unaffected since the keepers can be interrupted. At reduced supply voltages, the worst-case static droop (Figure 5) for the conventional flip-flop increases considerably to more than 40 percent of the supply voltage at 200 mV, severely affecting functionality. The circuit modifications just described reduce the worst-case droop by 4X in the ultra-low voltage optimized design.

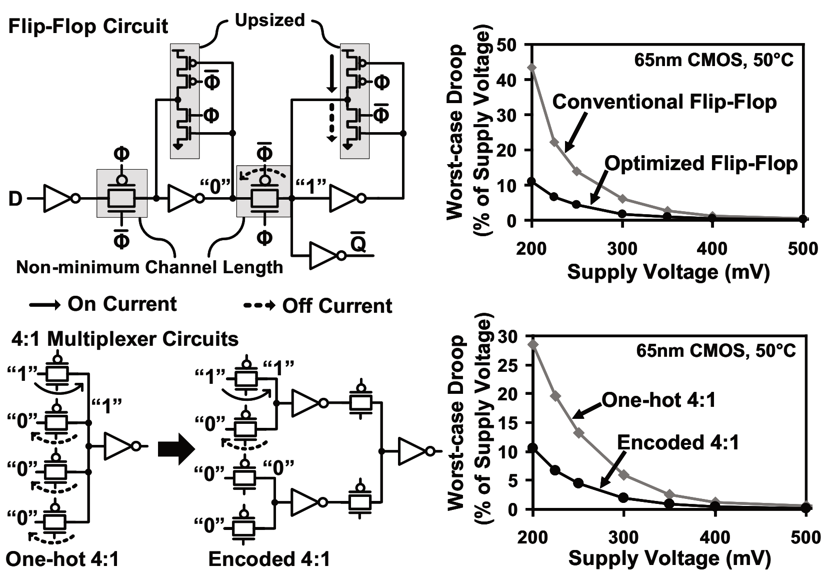


Figure 5: Flip-flop and Multiplexer Optimizations for Ultra-low Voltage Operation
Source: Intel Corporation, 2009

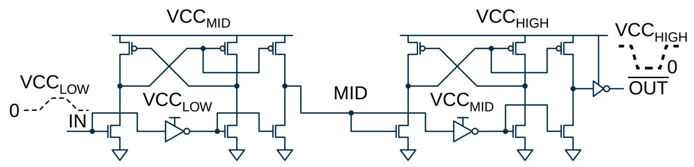
Wide multiplexers are also prone to static droops on nodes shared by transmission gates at ultra-low supplies. Such structures are typical for one-hot multiplexers, where the on-current of one of the selected inputs contends with the off-current of the remaining unselected inputs. To avoid this situation, wide multiplexers have been remapped in the datapath by the use of 2:1 multiplexers, thereby reducing worst-case, off-current contention. Remapping a one-hot 4:1 multiplexer to an encoded 4:1 multiplexer composed of 2:1 multiplexers results in up to a 3X reduction in worst-case static droop (Figure 5).

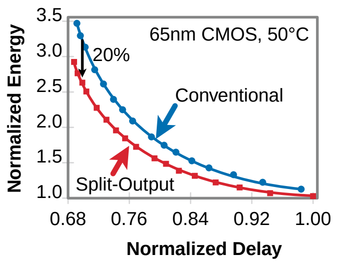
Other optimizations in the datapath include remapping deep stacked combinational logic and series connected transmission gates to a maximum of three transistor stacks. All the circuit optimizations to enable reliable ultra-low voltage operation result in a 1 percent area, a 1 percent power, and a 2 percent performance penalty for the accelerator at the nominal 1.2 V supply, representing a favorable tradeoff for enabling ultra-low voltage operation.

"Optimizations in the datapath include remapping deep stacked combinational logic and series connected transmission gates to a maximum of three transistor stacks."

Level Shifter Circuits

The use of multiple supply voltage domains results in the need for level shifter circuits at the low-to-high voltage domain boundaries. Conventional level shifters use a cascode voltage switch logic (CVSL) stage to provide the up-conversion functionality, with the associated contention currents contributing to a significant portion of power of the level shifter. Driving the output load directly with the CVSL stage increases its size, while use of additional gain stages at the output of the level shifter, to reduce CVSL stage loading, results in increased delay. The low-voltage output of the ME accelerator is up-converted to the nominal voltage level by using a two-stage cascaded split-output level shifter (Figure 6). An intermediate supply voltage for up-conversion over such a large voltage range limits the maximum current ratio between the higher-supply PMOS pull-up and lower-supply NMOS pull-down devices for correct CVSL stage functionality. Energy-efficient, up-conversion from sub-threshold voltage levels to nominal supply outputs is achieved by decoupling the CVSL stage of this level shifter from the output, thereby enabling a downsized CVSL stage for the same load without the need for extra gates in the critical path. Reduced contention currents in a downsized CVSL stage enable the split-output design to achieve up to a 20 percent energy reduction for equal fan-out and delay (Figure 6). Furthermore, simultaneous reductions of 11 percent in level shifter area and 14 percent fan-in loading (Table below) are achieved with the split-output design.





Normalized Comparison	Conventional CVSL	Split-Output CVSL
Fan-in Capacitance	1.0	0.86
Total Area	1.0	0.89
CVSL-stage Contention Energy	1.0	0.46

Figure 6: Two-stage Cascaded Split-output Level Shifter and Comparisons to Conventional CVSL Level Shifter
Source: Intel Corporation, 2009

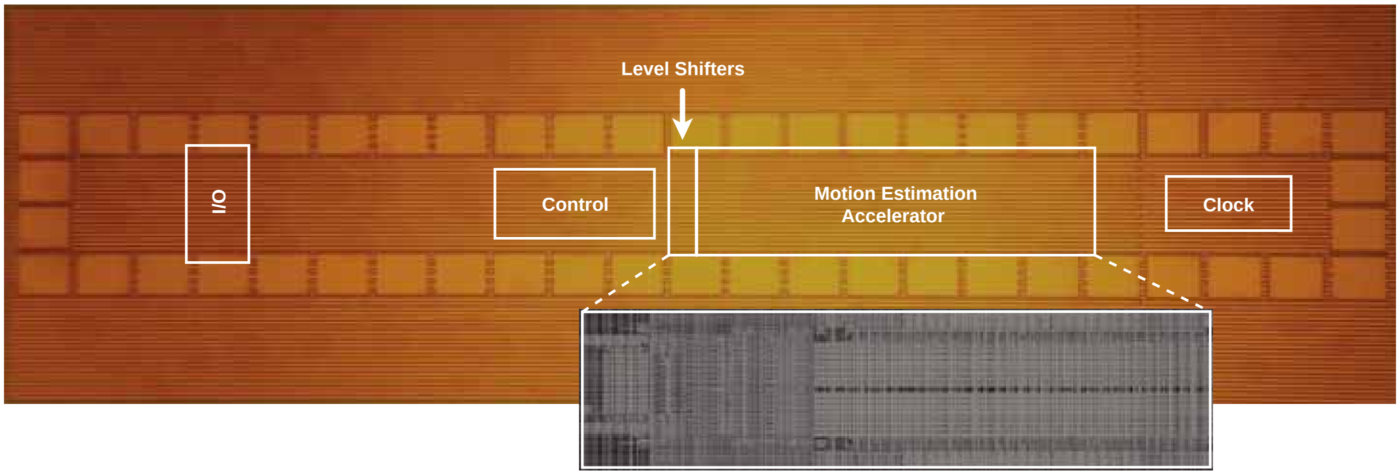


Figure 7: ME Accelerator Die Photograph
Source: Intel Corporation, 2009

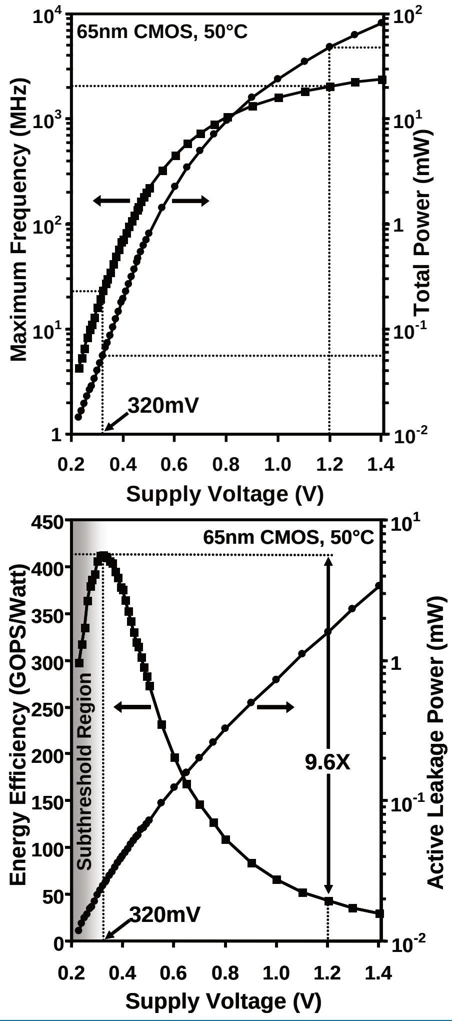


Figure 8: Maximum Frequency, Total Power, Energy Efficiency and Active Leakage Power Measurements Versus Supply Voltage
Source: Intel Corporation, 2009

Measurement Results

The accelerator operates at a nominal supply voltage of 1.2 V and is implemented in a 65 nm CMOS technology, with one poly and eight layer copper interconnects [12]. Figure 7 shows the die micrograph of the chip, with the ME accelerator occupying an area of 0.089 mm^2 (Table 1). The total die area is 0.96 mm^2, with a pad count of 50. The total number of transistors in the ME accelerator and test circuits is 70,000.

Process	65 nm CMOS
Nominal Supply	1.2 V
Interconnect	1 poly, 8 metal Cu
Accelerator Area	0.089 mm^2
Die Area	0.960 mm^2
Number of Transistors	70 K
Pad Count	50

Table 1: Implementation Details
Source: Intel Corporation, 2009

Frequency and power measurements (Figure 8) of the ME accelerator were obtained by sweeping the supply voltage in a temperature-stabilized environment of 50° C. The accelerator is fully functional over a wide operating range of 1.4 V to 230 mV (sub-threshold region). At the nominal supply of 1.2 V, the ME accelerator operates at a maximum frequency of 2.055 GHz, consuming a total power of 48 mW. This represents an energy-efficiency metric of 43 GOPS/Watt, where one operation is a complete SAD of eight pixel pairs. Performance can be scaled up to 2.4 GHz at 1.4 V, with a total power consumption of 82 mW, for purposes such as accelerating high-resolution video streams.

Figure 8 also shows the energy-efficiency and leakage-power measurements over the same range of supply voltages at a temperature of 50° C. The total active leakage power component is 1.6 mW (3 percent of total power) at the nominal 1.2 V supply, increasing to 3.4 mW at 1.4 V. As the supply voltage is reduced, the power consumption drops at a faster rate than the maximum frequency, resulting in improved energy efficiency. The ME algorithms have a high degree of parallelism, as well as a wide range of performance requirements, to take advantage of the increased SAD unit energy efficiency at the ultra-low voltage supplies. Ultra-low voltage circuit optimizations enable reliable operation at deep sub-threshold supply voltages as low as 230 mV, with a frequency of 4.3 MHz and a total power consumption of 14.4 μW. However, accelerator performance in the deep sub-threshold region degrades at a higher rate than total power, resulting in sub-optimal energy efficiency. Peak energy efficiency of 411 GOPS/Watt is measured at a near-threshold supply voltage of 320 mV, with a maximum frequency of 23 MHz and total power of 56 μW (9.6X higher energy efficiency compared to nominal voltage operation at 1.2 V). Processing a 16x16-pixel macro-block requires 32 SAD8 operations, resulting in a peak SAD efficiency of 12.8 macro-block SADs/nJ. Although absolute leakage power scales with supply voltage (Figure 8), the leakage component of total power increases to 44 percent at the energy-optimal 320 mV supply. ME accelerator performance and energy-efficiency measurements are summarized in Table 2.

Worst-case Power	48 mW at 2.055 GHz, 1.2 V, 50˚ C (nominal)
Active leakage power	1.6 mW at 1.2 V, 50˚ C (3% of total power)
Nominal performance	2.055 GHz, 43 GOPS/Watt at 1.2 V, 50˚ C
Peak performance	2.4 GHz, 82 mW at 1.4 V, 50˚ C
Ultra-low voltage mode total power	56 μW at 23 MHz, 320 mV, 50˚C
Ultra-low voltage energy-efficiency	411 GOPS/Watt at 320 mV, 50˚ C (9.6X higher than nominal)
Minimum supply voltage operation	4.3 MHz, 14.4 μW at 230 mV, 50˚ C

Table 2: Measured Performance and Power Summary
Source: Intel Corporation, 2009

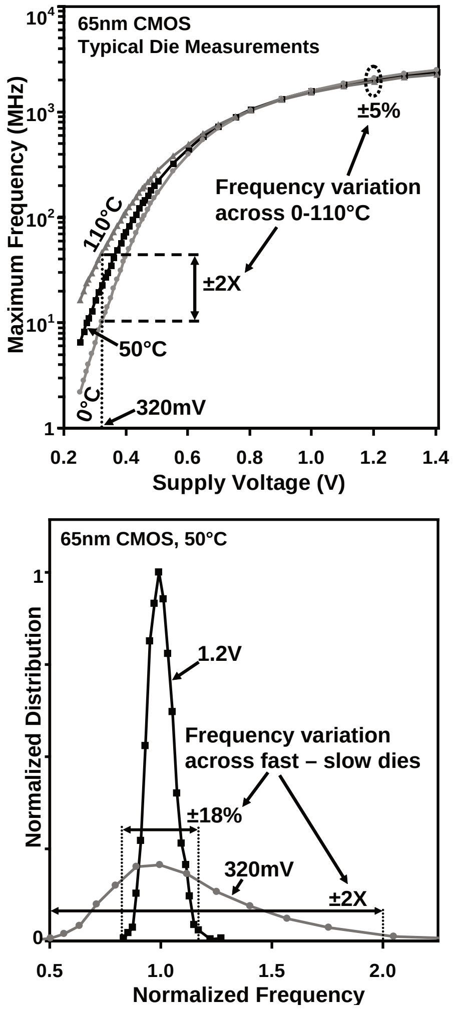


Figure 9: Frequency Variations Across 0 – 110° C Temperature and Fast-slow Process Skews
Source: Intel Corporation, 2009

Supply Compensation

On-currents are exponentially related to process parameters and temperature at the ultra-low supply voltages, which increases the process and temperature-induced performance variations. Frequency measurements of the accelerator (typical die) over a supply voltage range of 1.4 V to 0.25 V for constant temperatures of 0, 50, and 110°C (Figure 9), show a temperature-based frequency variation of ±5 percent around the 50° C temperature point, at the 1.2 V nominal supply. This variation increases to ±2x at the peak efficiency supply voltage of 320 mV. Figure 9 also shows the normalized performance distributions due to process variations for the accelerator at the 1.2 V and 320 mV supplies. These distribution curves are obtained from Monte-Carlo-based variation analysis and simulations, with the performance for the two supplies normalized to their respective medians. Frequency spread between fast and slow skews increases from ±18 percent at 1.2 V, to ±2x at the 320 mV supply. While circuit monitoring and compensation techniques have been demonstrated for nominal operation [13, 14], ultra-low voltage operation requires compensation solutions to address a much larger performance spread.

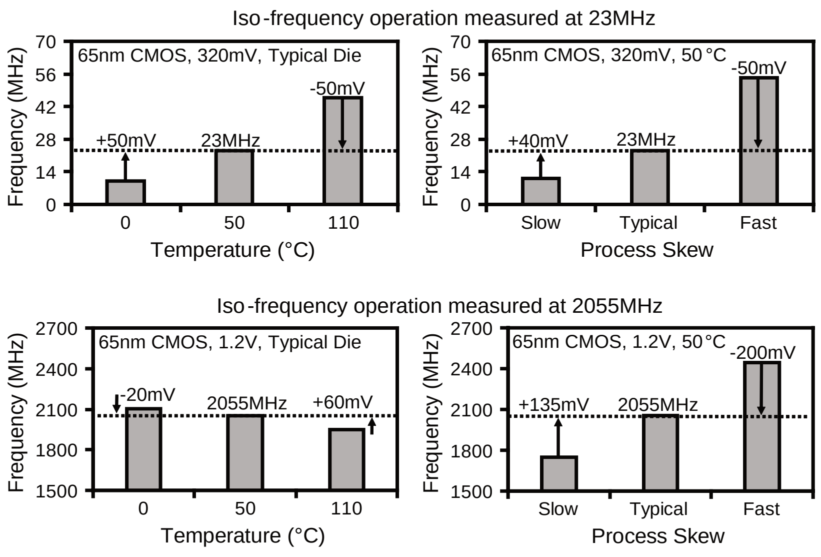


Figure 10. Supply Voltage Compensation Measurements for Temperature and Process Variation at 320 mV and 1.2 V Operation
Source: Intel Corporation, 2009

The exponential relationship between on-currents and supply voltage in the ultra-low voltage region enables the use of supply voltage as a powerful knob to compensate for the increased process and temperature-induced variations. Compensation for the ±2x performance difference across the extreme temperature range of 0-110°C, for a typical die operating at 320 mV, is provided by increasing the supply voltage at the lower temperature by 50 mV and decreasing the supply voltage at the higher temperature by 50 mV to maintain the nominal performance of 23 MHz (Figure 10). Compensation across process skews at 320 mV, at a constant 50°C temperature, requires a supply voltage increase of 40 mV at the slow process skew, and a decrease of 50 mV for the fast process skew is required to maintain the nominal performance. Thus, a measured range of ±50 mV above/below 320 mV is adequate to compensate for the larger performance variations across a wide range of process or temperature corners. In comparison, at the nominal supply of 1.2 V, supply voltage adjustments of -20 mV and +60 mV at 0 and 110° C, respectively, are required to address the ±5 percent performance variation below/above 50° C to maintain the nominal 2.055 GHz performance (Figure 10). To address the ±18 percent performance variations due to process skew at 1.2 V, the accelerator at the slow/fast process skews requires supply compensation of +135/-200 mV to maintain iso-frequency. The larger range of supply voltage compensation for process skews at nominal supplies is due to the lower sensitivity of performance to supply voltage in this region.

"Special-purpose hardware accelerators and robust ultra-low voltage operation are key enablers for improved energy efficiency in multi-core processors with tera-scale performance."

Summary and Conclusions

Special-purpose hardware accelerators and robust ultra-low voltage operation are key enablers for improved energy efficiency in multi-core processors with tera-scale performance. We demonstrate these two technologies in this article by describing the design, challenges, solutions, and silicon measurements for an ultra-low voltage ME accelerator, fabricated in 65 nm CMOS. Energy-efficient circuits enable computation of eight pixel-pair SADs with an energy efficiency of 411 GOPS/Watt at 320 mV, consuming 56 μW of power for 23 MHz operation. Ultra-low voltage circuit optimizations enable robust operation over a wide range of supply voltages from 1.4 V down to a sub-threshold operation at 230 mV. At the nominal supply of 1.2 V, the accelerator operates at 2.055 GHz consuming 48 mW at 50° C, and it scales to 2.4 GHz operation at 1.4 V. Supply adjustments of ±50 mV compensate for ±2x process and temperature-induced variations at the ultra-low supply voltage of 320 mV, enabling the accelerator to achieve high energy efficiency at the ultra-low supplies with constant performance.

"Energy-efficient circuits enable computation of eight pixel-pair SADs with an energy efficiency of 411 GOPS/Watt at 320 mV, consuming 56 μW of power for 23 MHz operation."

References

[1] H.-C. Chang et al. "A 7mW-to-183mW Dynamic Quality-Scalable H.264 Video Encoder Chip." *ISSCC Digest of Technical Papers,* pages 280-281, Feb. 2007.

[2] C.-P. Lin et al. "A 5mW MPEG4 SP Encoder with 2D Bandwidth-Sharing Motion Estimation for Mobile Applications." *ISSCC Digest of Technical Papers,* pages 412-413, Feb. 2006.

[3] Y.-K. Lin et al. "A 242mW 10mm^2 1080p H.264/AVC High-Profile Encoder Chip." *ISSCC Digest of Technical Papers,* pages 314-315, Feb. 2008.

[4] H. Yamauchi et al. "An 81MHz, 1280 x 720pixels x 30frames/s MPEG-4 Video/Audio Codec Processor." *ISSCC Digest of Technical Papers,* pages 130-131, Feb. 2005.

[5] Y.-W. Huang et al. "A 1.3TOPS H.264/AVC Single-Chip Encoder for HDTV Applications." *ISSCC Digest of Technical Papers,* pages 128-129, Feb. 2005.

[6] T. Fujiyoshi et al. "An H.264/MPEG-4 Audio/Visual Codec LSI with Module-Wise Dynamic Voltage/Frequency Scaling." *ISSCC Digest of Technical Papers,* pages 132-133, Feb. 2005.

[7] A. Wang and A. Chandrakasan. "A 180-mV Subthreshold FFT Processor Using a Minimum Energy Design Methodology." *IEEE Journal Solid-State Circuits,* pages 310-319, Jan. 2005.

[8] J. Kwong et al. "A 65nm Sub-Vt Microcontroller with Integrated SRAM and Switched-Capacitor DC-DC Converter." *ISSCC Digest of Technical Papers,* pages 318-319, Feb. 2008.

[9] S. Hanson et al. "Exploring Variability and Performance in a Sub-200-mV Processor." *IEEE Journal Solid-State Circuits,* pages 881-891, April 2008.

[10] M.-E. Hwang et al. "A 85mV 40nW Process-Tolerant Subthreshold 8x8 FIR Filter in 130nm Technology." *Symposium on VLSI Circuits*, pages 154-155, June 2007.

[11] H. Kaul et al. "A 320mV 56μW 411GOPS/Watt Ultra-Low Voltage Motion Estimation Accelerator in 65nm CMOS." *IEEE Journal Solid-State Circuits,* pages 107-114, Jan. 2009.

[12] P. Bai et al. "A 65nm Logic Technology Featuring 35nm Gate Lengths, Enhanced Channel Strain, 8 Cu Interconnect Layers, Low-k ILD and 0.57μm^2 SRAM Cell." *IEDM Technical Digest,* pages 657-660, Dec. 2004.

[13] R. McGowen et al. "Power and Temperature Control on a 90-nm Itanium Family Processor." *IEEE Journal Solid-State Circuits,* pages 229-237, Jan. 2006.

[14] C. Kim et al. "An On-Die CMOS Leakage Current Sensor for Measuring Process Variation in Sub-90nm Generations." *Symposium on VLSI Circuits,* pages 250-251, June 2004.

Acknowledgments

We thank the Pyramid Probe Card Division of Cascade Microtech Inc. for high bandwidth membrane probe solutions; and M. Haycock, J. Schutz, S. Borkar, G. Taylor, S. Pawlowski, A. Chien, C. Riley, E. Hannah, Prof. V. Oklobdzija, Prof. K. Roy, and Prof. A. Chandrakasan for encouragement and discussions.

Authors' Biographies

Ram K. Krishnamurthy received a BE degree in Electrical Engineering from the Regional Engineering College, Trichy, India, in 1993, and a PhD degree in Electrical and Computer Engineering from Carnegie Mellon University, Pittsburgh, PA, in 1998. Since 1998, he has been with Intel Corporation's Circuits Research Labs in Hillsboro, Oregon, where he is currently a Senior Principal Engineer and heads the high-performance and low-voltage circuits research group. He holds 80 issued patents and has published over 75 conference/journal papers. He served as the Technical Program Chair/General Chair for the 2005/2006 IEEE International Systems-on-Chip Conference. He has received two Intel Achievement Awards, in 2004 and 2008, for the development of novel arithmetic circuit technologies and hardware encryption accelerators. His email is ram.krishnamurthy at intel.com.

Himanshu Kaul received a B.Eng. (Hons.) degree in Electrical and Electronics Engineering from the Birla Institute of Technology and Science, Pilani, India, in 2000, and an MS and PhD degree in Electrical Engineering from the University of Michigan, Ann Arbor, in 2002 and 2005, respectively. Since 2004, he has been with Intel Corporation's Circuits Research Labs in Hillsboro, OR, where he is currently a Research Engineer in the High-Performance and Low Voltage Circuits Research Group. His research interests include on-chip signaling techniques and low-power and high-performance circuit design. His e-mail is himanshu.kaul at intel.com.

Copyright

LESSONS LEARNED FROM THE 80-CORE TERA-SCALE RESEARCH PROCESSOR

Contributors

Saurabh Dighe
Intel Corporation

Sriram Vangal
Intel Corporation

Nitin Borkar
Intel Corporation

Shekhar Borkar
Intel Corporation

Index Words

TeraFLOPS
Many-core
80-tile
Network-on-chip
2D Mesh Network

Abstract

Sustained tera-scale-level performance within an affordable power envelope is made possible by an energy-efficient, power-managed simple core, and by a packet-switched, two-dimensional mesh network on a chip. From our research, we learned that (1) the network consumes almost a third of the total power, clearly indicating the need for a new approach, (2) fine-grained power management and low-power design techniques enable peak energy efficiency of 19.4 GFLOPS/Watt and a 2X reduction in standby leakage power, and (3) the tiled design methodology quadruples design productivity without compromising design quality.

Introduction

Intel's Tera-scale Research Computing Program [1] lays out a vision for future computing platforms and underscores the need for tera-scale performance. We envision hundreds of networked cores running complex parallel applications under a highly constrained energy budget. Consequently, one of the important research areas in this initiative is to develop a scalable tera-scale processor architecture that can address the needs of our future platforms. The Teraflops Research Processor is a key first step in this direction. Focusing on some of the vital ingredients of a tera-scale architecture: a power optimized core, a scalable on-chip interconnect, and a modular global clocking solution, we established the following research goals for our project:

- Achieve teraFLOPS performance under 100 W.
- Prototype a high-performance and scalable on-chip interconnect.
- Demonstrate an energy-efficient architecture with fine-grained power management.
- Develop design methodologies for network-on-chip architectures (NoC).

Our intent in this article is to focus on key lessons we learned from the research prototype. We present our findings in a structured format. We first provide an overview of the chip and briefly describe key building blocks. We then highlight the novel design techniques implemented on the chip and the tiled design approach. Next, we summarize measured silicon results. Finally, we discuss the pros and cons of certain design decisions, including our recommendations for future tera-scale platforms.

Architecture Overview

Rapid advancement in semiconductor process technology and a quest for increased energy efficiency have fueled the popularity of multi-core and NoC architectures [2]. The teraFLOPS research processor contains 80 tiles arranged as an 8 x 10, 2-D mesh network, shown in Figure 1. Each tile consists of a processing engine (PE) connected to a 5-port router with mesochronous interfaces (MSINT), which forwards packets between the tiles. More detailed information on the chip architecture and interconnect can be found in [3, 4].

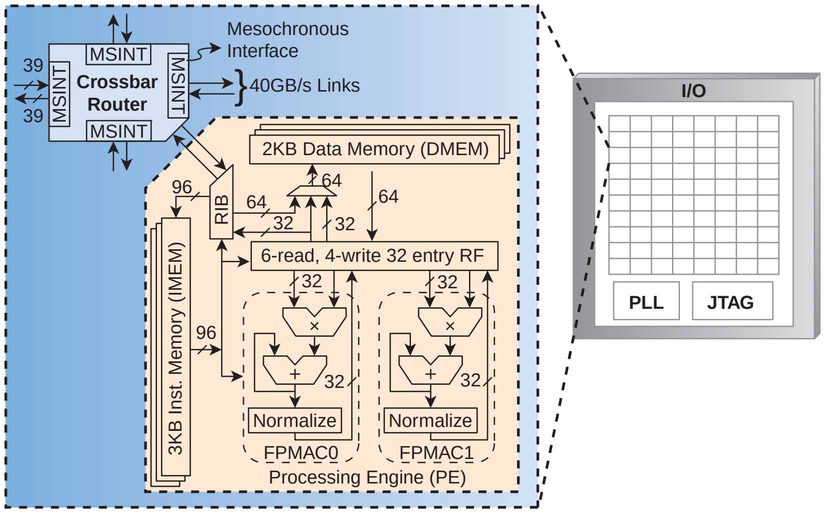


Figure 1: 80-core Processor Tile Architecture
Source: Intel Corporation, 2009

Processing Engine

The PE contains two independent fully-pipelined, single-precision, floating-point multiply-accumulator (FPMAC) units capable of providing an aggregate performance of 20 GFLOPS. The key to achieving this high performance is a fast, single cycle, accumulation algorithm [5], developed by analyzing each of the critical operations involved in conventional floating point units (FPUs) with the intent of eliminating, reducing, or deferring the amount of logic operations inside the accumulate loop.

We came up with the following three optimizations. First, the accumulator retains the multiplier output in carry-save format and uses an array of 4-2 carry-save adders to accumulate the results in an intermediate format. This removes the need for a carry-propagate adder in the critical path. Second, accumulation is performed in base 32, converting expensive variable shifters in the accumulate loop to constant shifters. Third, we moved the costly normalization step outside the accumulate loop, where the accumulation result in carry-save is added, and the sum is normalized and converted back to base 2. These optimizations allow accumulation to be implemented in just fifteen FO4 stages. This approach also reduces the latency of dependent FPMAC instructions and enables a sustained multiply-add result (2FLOPS) every cycle. Moving to 64-bit arithmetic results in wider mantissa for increased throughput.

"The PE contains two independent fully-pipelined, single-precision units capable of providing an aggregate performance of 20 GFLOPS."

"The 80-tile, on-chip network is a 2D mesh that provides a bisection bandwidth of 2 Terabits/s."

The PE includes a 3-KB, single-cycle, instruction memory (IMEM) and 2KB of data memory (DMEM). This amounts to a total distributed on-die memory of 400 KB. The capacity of the local memory was enough to support blocked execution of a select few LAPACK kernels. With a 10-port (6-read, 4-write) register file, we allow scheduling to both FPMACs, simultaneous DMEM load/store, and packet send/receive from the mesh network. A router interface block (RIB) handles packet encapsulation between the PE and router.

On-chip Interconnect

The 80-tile, on-chip network is a 2D mesh that provides a bisection bandwidth of 2 Terabits/s. The key communication block for the NoC is a 5-port, pipelined, packet-switched router with two virtual lanes (see Figure 2) capable of operating at 5 GHz [6] at a nominal supply of 1.2 V. It has a 6-cycle latency or 1.2 ns/hop at 5 GHz. It connects to each of its neighbors and the PE by using phase-tolerant mesochronous links that can deliver data at 20 GBytes/sec. The network uses a source-directed routing scheme, based on wormhole switching, that has two virtual lanes for dead-lock free routing and an on-off scheme, by using almost-full signals for flow control. The width of the links and router frequency were chosen to transfer a single precision FPU operand at high speed and approximately 1 ns/hop latency.

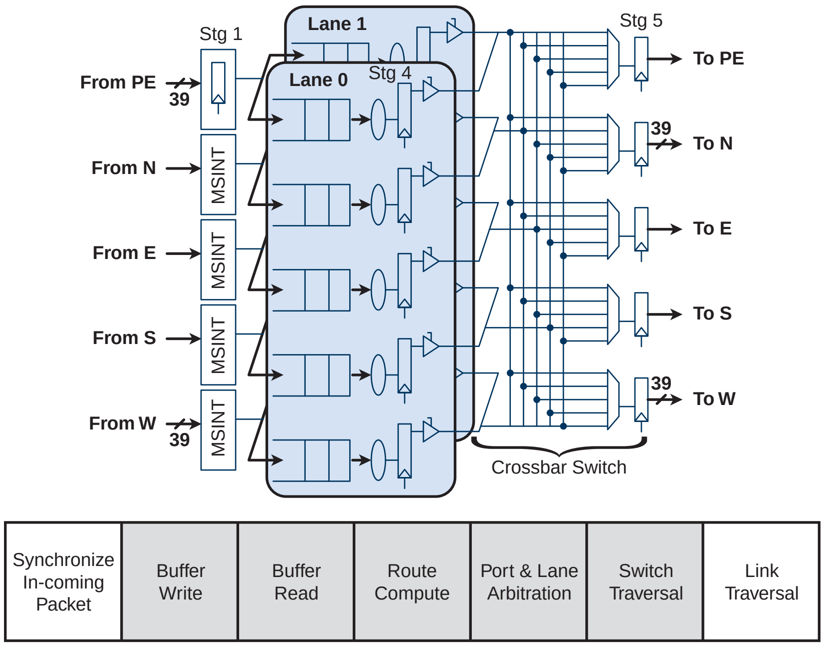


Figure 2: 5-port Two-lane Shared Crossbar Router Architecture
Source: Intel Corporation, 2009

Instruction Set and Programming Model

We define a 96-bit Very Long Instruction Word (VLIW) that allows a maximum of up to eight operations to be issued every cycle. The instructions fall into one of five categories: instruction issue to both floating-point units, simultaneous data memory load and stores, packet send/receive via the on-die mesh network, program control that uses jump and branch instructions, and synchronization primitives for data transfer between PEs. With the exception of FPU instructions, which have a pipeline latency of nine cycles, most other instructions execute in one to two cycles.

"We define a 96-bit Very Long Instruction Word (VLIW) that allows a maximum of up to eight operations to be issued every cycle."

To aid with power management, the instruction set includes support for dynamic sleep and wakeup of each floating-point unit. The architecture allows any PE to issue sleep packets to any other tile or to wake it up for processing tasks.

The architecture supports a message-passing programming model by providing special instructions to exchange messages to coordinate execution and share data [7]. The fully symmetric architecture allows any PE to send or receive instructions and data packets to or from any other tile.

Novel Circuit and Design Techniques

We used several circuit techniques to achieve high performance, low power, and a short design cycle. The fifteen FO4 design uses a balanced core and router pipeline, with critical stages employing performance-setting, semi-dynamic flip-flops. In addition, a robust scalable mesochronous clock distribution is employed in a 65-nanometer, 8-metal CMOS process that enables high integration and single-chip realization of the teraFLOP processor.

"With the exception of FPU instructions, which have a pipeline latency of nine cycles, most other instructions execute in one to two cycles."

Circuit Design Style

To enable a 5-GHz operation, we designed the entire core by using hand-optimized datapath macros. For quick turnaround we used CMOS static gates to implement most of the logic. However, critical registers in the FPMAC and at the router crossbar output utilize implicit-pulsed, semi-dynamic flip-flops (SDFF) [8, 9], which have a dynamic master stage coupled with a pseudostatic slave stage. When compared to a conventional static, master-slave flip-flop, SDFF provides both shorter latency and the capability of incorporating logic functions, with minimum delay penalty, each of which are desirable properties in high-performance digital designs. However, pulsed flip-flops have several important disadvantages. The worst-case hold time of this flip-flop can exceed clock-to-output delay because of pulse width variations across process, voltage, and temperature conditions. Therefore, pulsed flip flops must be carefully designed to avoid failures due to min-delay violations.

"To enable a 5-GHz operation, we designed the entire core by using hand-optimized datapath macros."

"We used fine-grained clock gating and sleep transistor circuits to reduce active and standby leakage power, which are controlled at full-chip, tile-slice, and individual tile levels, based on workload."

Fine grain Power Management

To achieve the goal of demonstrating teraFLOPS performance below 100 watts of power, we had to adopt and combine various power-saving features and use innovative power-management technologies. To this end, we used fine-grained clock gating and sleep transistor circuits [10] to reduce active and standby leakage power, which are controlled at full-chip, tile-slice, and individual tile levels, based on workload. Figure 3 shows clock and power gating in the FPMAC, router, and instruction/data memories. Approximately 90 percent of FPU logic and 74 percent of each PE is sleep enabled. Each tile is partitioned into twenty-one smaller sleep regions, and dynamic control of individual blocks is based on instruction type. Each FPMAC can be controlled through NAP/WAKE instructions. The router is partitioned into ten smaller sleep regions, and control of individual router ports depends on network traffic patterns. We inserted sleep transistors in the register file cells without impacting area too much. An additional track had to be used to route the sleep signal. Special attention was paid to sleep-non-sleep interfaces, and intelligent data gating at flip-flop boundaries ensured additional firewall circuits were not required.

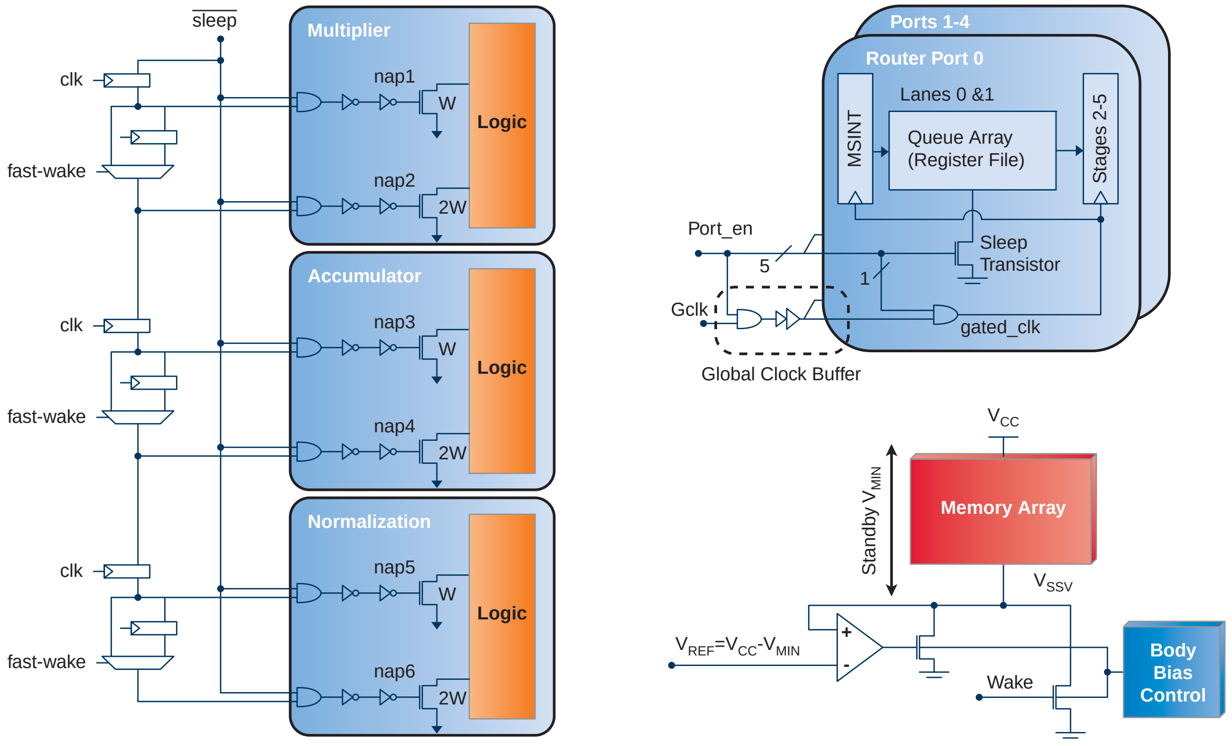


Figure 3: Fine-grain Power Management in the Tile
Source: Intel Corporation, 2009

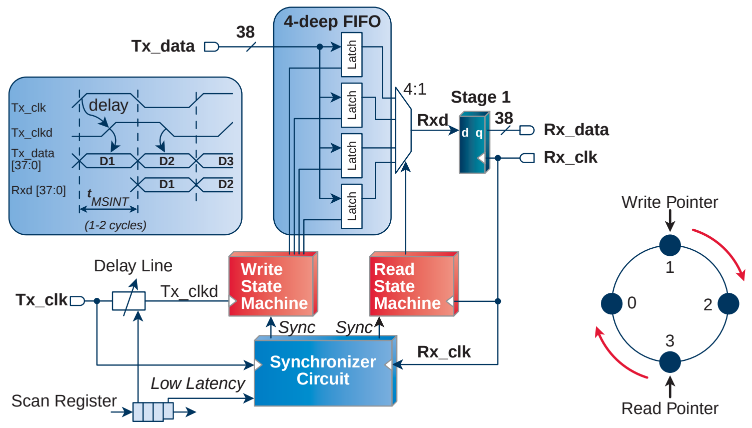


Figure 4: Phase-tolerant Mesochronous Interface
Source: Intel Corporation, 2009

Mesochronous Clocking

The chip uses a scalable global mesochronous clocking technique, that allows for clock-phase-insensitive communication across tiles and for synchronous operation within each tile. The on-chip PLL output is routed by using horizontal (Metal-8) and vertical (Metal-7) spines. Each spine consists of differential clocks for low duty-cycle variation along the worst-case clock route of 26 mm. An op-amp at each tile converts the differential clock inputs to a single-ended clock with a 50 percent duty cycle, prior to distribution, by using an H-tree. The 2-mm long point-to-point, unidirectional router links implement a phase-tolerant, mesochronous interface as shown in Figure 4. This allows clock-phase-insensitive communication across tiles and enables a scalable, on-die communication fabric that simplifies global clock distribution.

Double-pumped Crossbar

The crossbar switch area increases as a square function $O(n^2)$ of the total number of I/O ports and the number of bits per port. Consequently, the crossbar can dominate a large percentage of the area. To alleviate this problem, we double pump the crossbar data buses by interleaving alternate data bits as shown in Figure 5. We use dual-edge triggered flip-flops to do this; thereby, effectively reducing by half the crossbar hardware cost.

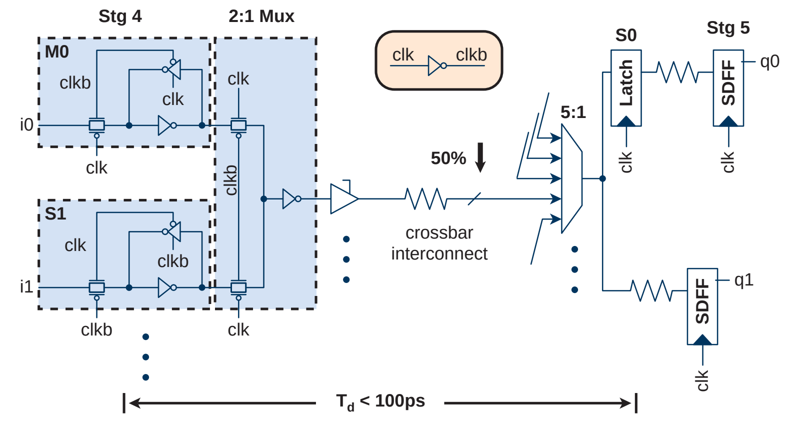


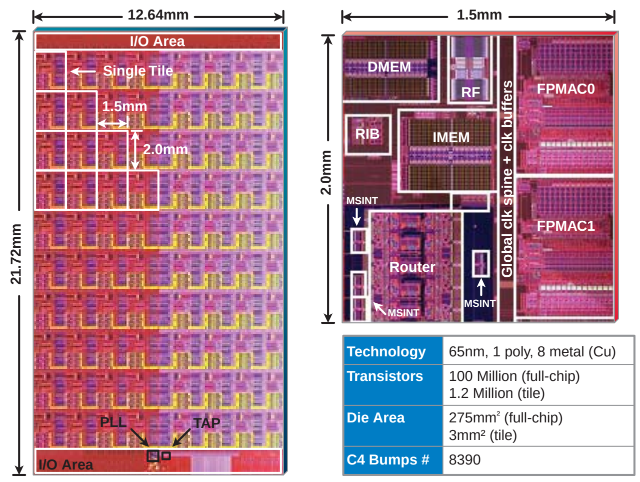
Figure 5: Double-pumped Crossbar
Source: Intel Corporation, 2009

"Each tile is completely self-contained, including power bumps, power tracks, and global clock routing."

Tiled-design Methodology

While implementing the teraFLOPS processor, we followed a "tiled design methodology" where each tile is completely self-contained, including power bumps, power tracks, and global clock routing. This design enabled us to seamlessly array all tiles at the top level, by simply using abutment. This methodology enabled rapid completion of a fully custom design with less than 400 person-months of effort.

Results



Technology	65nm, 1 poly, 8 metal (Cu)
Transistors	100 Million (full-chip) 1.2 Million (tile)
Die Area	275mm² (full-chip) 3mm² (tile)
C4 Bumps #	8390

Figure 6: Full-chip and Tile Micrograph and Characteristics
Source: Intel Corporation, 2009

We fabricated the teraFLOPS processor in 65-nm process technology. The die photographs in Figure 6 identify the chip's functional blocks and individual tiles. The 275-mm^2, fully custom design contains 100 million transistors. The chip supports a wide dynamic range of operation; namely, 1 GHz at 670 mV up to 5.67 GHz at 1.35 V, as shown in Figure 7. Increased performance with higher voltage and frequency can be achieved at the cost of power. As we scale Vcc/frequency, power consumption ranges from 15.6 W to 230 W as shown in Figure 8. Fine-grain sleep transistors limit the leakage power from 9.6 percent to 15.6 percent of the total power. With all 80 tiles actively performing single-precision, block-matrix operations, the chip achieves a peak performance of 1.0 TFLOPS at 3.16 GHz while dissipating 97 W. By reducing voltage, and by operating close to the threshold voltage of the transistor, energy efficiency for the stencil application can be improved from 5.8 GFLOPS per Watt to a maximum of 19.4 GFLOPS per Watt as shown in Figure 9.

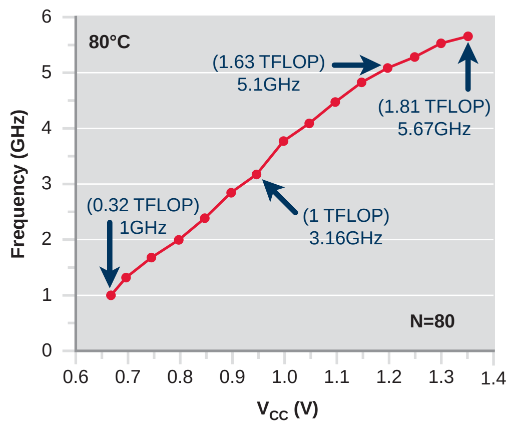


Figure 7: Measured Chip Fmax and Peak Performance
Source: Intel Corporation, 2009

The measured global clock distribution power is 2 W at 1.2 V and 5.1 GHz operation, and it accounts for just 1.3 percent of the total chip power. At the tile level, power breakdown shows that the dual FPMACs account for 36 percent of total power, the router and links account for 28 percent, the IMEM and DMEM account for 21 percent, the tile-level synchronous clock distribution accounts for 11 percent, and the multi-ported register file accounts for 4 percent. In sleep mode, the nMOS sleep transistors are turned off, reducing chip leakage by 2X, while preserving the logic state in all memory arrays. Total network power per tile can be lowered from a maximum of 924 mW with all router ports active to 126 mW, resulting in a 7.3X reduction. The network leakage power per tile when all ports and global clock buffers to the router are disabled is 126 mW. This number includes power dissipated in the router, MSINT, and in the links.

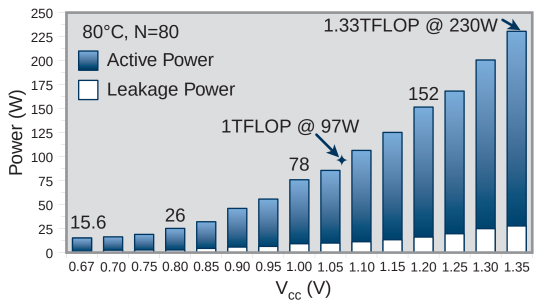


Figure 8: Measured Power Versus Vcc for Stencil Application
Source: Intel Corporation, 2009

Discussion and Tradeoffs

The goal of achieving teraFLOPS performance under 100 W entails studying the traditional tradeoffs between performance, power, and die size, but equally important are looking at issues such as multi-generation scalability, modular design/validation, and support for parallel programming models.

Today's general-purpose cores are capable of performance in the order of tens of GFLOPS. However, achieving teraFLOPS performance with these cores on the current process technology is prohibitive, from an area and power perspective. Our work corroborates that a computational fabric built by using programmable, special-purpose cores provides high levels of performance in an energy-efficient manner. Power-optimized fast computation hardware, simple decoded VLIW instruction words, and low-power memories ensure that a large percentage of the energy consumed goes towards computing FLOPS. While architecting the core we were aware of the importance of balancing data memory bandwidth with compute/communication bandwidth, which entailed adding a single cycle, 6-read, 4-write register file. As data transfer on chip costs significant energy, larger caches will be required to keep the data local. Maintaining coherency across many cores is a significant challenge as well. Hardware costs and increased coherency traffic on the mesh will pose hurdles for completely hardware-based coherent systems. Instead, future tera-scale processors will explore message-passing architectures. Special on-die, message-passing hardware is very efficient for core-to-core communication, making software-based coherency with hardware assists a viable solution for the future. In addition to support for message passing, another enhancement that proved important is the ability to overlap compute and communication. A core can directly transfer instructions/data into the local memory of another core without interrupting the other core. This resulted in improved FPMAC utilization with fewer idle cycles and enabled performance numbers that were close to the maximum achievable.

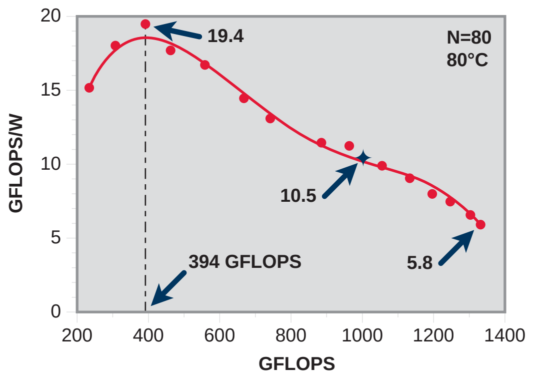


Figure 9: Measured Chip Energy Efficiency for Stencil Application
Source: Intel Corporation, 2009

With increasing demand for interconnect bandwidth, on-chip networks are taking up a substantial portion of the system's power budget. The router on our teraFLOPS processor consumes 28 percent of tile power. Our goal for a compelling solution is to use compact low-power routers that consume less than 10 percent of the chip power and die budget. At the same time they must deliver high, on-die bisection bandwidth and low latency. Techniques, such as speculation and bypass, are well known, but they add to the power consumption and are therefore undesirable. Future routers would also need to incorporate extensive fine-grained, power-management techniques to enable dynamic operation that adapts to differing traffic patterns. Heterogeneous NoCs [11], that allocate resources as needed, and circuit-switched networks [12, 13] are promising approaches. Traffic patterns and bandwidth requirements are going to dictate on-die network architectures for the future. Hybrid approaches to on-die networks can save communication power by utilizing fewer fully-connected crossbar routers at the expense of reduced bandwidth. Instead of one router per core in each tile, we could amortize the power/area of the router by having two or more cores on a shared bus connected to the local port of the router in each tile.

"The router on our teraFLOPS processor consumes 28 percent of tile power. Our goal for a compelling solution is to use compact low-power routers that consume less than 10 percent of the chip power."

The two popular clocking techniques for on-die networks are 1) a completely synchronous system with closely matched skews, and 2) a globally asynchronous, locally synchronous system with handshaking signals for data transfer (GALS). Synchronous systems are the simplest to implement and are well understood, but they can consume significant power for high-frequency clock distribution. With increased with-in die variation, matching skews across large dies is becoming difficult, which also results in excessive timing guard bands. GALS suffers from area overhead, due to additional hand-shaking circuits, lack of mature design tools, and increased design complexity. The mesochronous clocking scheme tries to address these problems by distributing a single frequency clock without the overhead of matching clock skews. This causes phase differences between clocks, distributed to individual routers that need to be accounted for by synchronization circuitry in the data paths. This technique scales well as tiles are added or removed. Multiple cycles are required for the global clock to propagate to all 80 tiles; this systematic skew inherent in the distribution helps spread peak currents because of simultaneous clock switching. To support mesochronous or phase-tolerant communication across tiles, we pay a synchronization latency penalty for the benefit of a lightweight global clock distribution. The area and power overhead of the synchronizers can be significant for wide links. It is important to understand these tradeoffs before abandoning a synchronous implementation in favor of mesochronous clocking.

Tera-scale computing platforms need to be efficient to meet the energy constraints of future data centers. We employed per-tile, fine-grained power management with clock and power gating. By exposing WAKE/NAP instructions to software, we could put FPMACs to sleep during idle windows. This enabled us to reach an energy efficiency of 19.4 GFLOPS/Watt. In stark contrast, a 3-GHz, general-purpose CPU provides an energy efficiency of 0.07 GFLOPS/Watt. As different applications with different compute/communication profiles and performance requirements are invoked over time, the optimal number of cores and Vcc/frequency to achieve maximum energy efficiency varies. Hence, to further improve workload efficiency, we recommend dynamic voltage frequency scaling with independent voltage and frequency islands for future tera-scale processors.

"We reached an energy efficiency of 19.4 GFLOPS/Watt. In stark contrast, a 3-GHz, general-purpose CPU provides an energy efficiency of 0.07 GFLOPS/Watt."

To operate across a wide dynamic voltage range it is important to implement circuits with robust static CMOS logic that operate at low voltages. Operating close to threshold voltage of the transistor increases energy efficiency; however, contention circuits in register files and small signal arrays typically limit the lowest operating voltage (Vccmin) of a processor. It is critical for tera-scale processors to operate at the lowest energy point, and this makes research in Vccmin-lowering techniques a vital part of the tera-scale research agenda. Designs should also be optimized for power with extensive usage of low-leakage transistors and selective usage of nominal transistors in critical paths. It is important to strike a balance between delay penalty and leakage savings during device-type selection for sleep transistors. We chose to utilize nominal devices for 5-GHz operation with a 2X leakage savings.

As we integrate more cores on a single die, adopting a scalable design methodology is critical for design convergence, validation, product segmentation, and time-to-market. The proposed tiled design methodology enabled faster convergence in timing verification and physical design. Global wires that do not scale well with technology could be avoided. We ensured the tiles were small, completely self-contained, and could be assembled by abutment. This also ensures uniform metal/via density that helps in manufacturability and yield. Consequently, we achieved high levels of integration with a small design team and low overhead. Pre/post-silicon debug effort was greatly reduced with first silicon stepping fully functional. In addition, a standardized communication fabric with a predefined interface combined with a tiled design approach provides the flexibility of integrating any number of homogenous or heterogeneous cores and facilitates product segmentation.

"The tiles were small, completely self-contained, and could be assembled by abutment."

Conclusion

"Tera-flop performance is possible within a mainstream power envelope."

Tera-flop performance is possible within a mainstream power envelope. Careful co-design at architecture, logic, circuit, and physical design levels pays off, with silicon achieving an average performance of 1 TFLOP at 97 W and a peak power efficiency of 19.4 GFLOPS/Watt. Tile-based methodology fulfilled its promise, and the design was done with half the team in half the time. Communication power accounts for almost one-third of the total power, highlighting the need for further research in low-power, scalable networks that can satisfy the requirements of a tera-scale platform. On a final note, to be able to successfully exploit the computing capability of a tera-scale processor, research into parallel programming is vital.

References

[1] J. Held, J. Bautista, and S. Koehl. "From a few core to many: A tera-scale computing research overview." 2006. Available at ***http://www.intel.com***

[2] W.J. Dally and B. Towles. "Route Packets, Not Wires: On-Chip Interconnection Net-works." In *Proceedings 38th Design Automation Conference* (DAC 01), ACM Press, 2001, pages 681-689.

[3] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, C. Roberts, V. Erraguntla, Y. Hoskote, N. Borkar, and S. Borkar. "An 80-Tile Sub-100W TeraFLOPS Processor in 65-nm CMOS." *IEEE Journal of Solid-State Circuits,* vol. 43, pages 29–41, January 2008.

[4] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar. "A 5GHz Mesh Interconnect for a Teraflops Processor." *IEEE Micro,* Vol. 27, pages 51-61, 2007.

[5] S. Vangal, Y. Hoskote, N. Borkar and A. Alvandpour. "A 6.2-GFlops Floating-Point Multiply-Accumulator with Conditional Normalization." *IEEE Journal of Solid-State Circuits,* pages 2314–2323, Oct., 2006.

[6] S. Vangal, A. Singh, J. Howard, S. Dighe, N. Borkar, A. Alvandpour. "A 5.1GHz 0.34mm^2 Router for Network-on-Chip Applications." *IEEE Symposium on VLSI Circuits,* 2007. 14-16, June 2007.

[7] T. Mattson, R. Van der Wijngaart, M. Frumkin. "Programming the Intel 80-core network-on-a-chip terascale processor." In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing,* November 15-21, 2008, Austin, Texas.

[8] F. Klass. "Semi-Dynamic and Dynamic Flip-Flops with Embedded Logic." *1998 Symposium on VLSI Circuits, Digest of Technical Papers,* pages 108–109, 1998.

[9] J. Tschanz, S. Narendra, Z. Chen, S. Borkar, M. Sachdev, V. De. "Comparative Delay and Energy of Single Edge-Triggered & Dual Edge-Triggered Pulsed Flip-Flops for High-Performance Microprocessors." *ISLPED,* pages 147-151, 2001.

[10] J. Tschanz, S. Narendra, Y. Ye, B. Bloechel, S. Borkar and V. De. "Dynamic sleep transistor and body bias for active leakage power control of microprocessors." *IEEE Journal of Solid-State Circuits,* pages 1838–1845, Nov. 2003.

[11] K. Rijpkema et al. "Trade-Offs in the Design of a Router with Both Guaranteed and Best-Effort Services for Networks on Chip." In *IEEE Proceedings On Computers and Digital Techniques,* vol. 150, no. 5, 2003, pages 294-302.

[12] P. Wolkette et al. " An Energy-Efficient Reconfigurable Circuit-Switched Network-on-Chip." In *Proceedings IEEE International Parallel and Distributed Symposium, (IPDS 05), IEEE CS Press,* 2005, pages 155a.

[13] M. Anders, H. Kaul, M. Hansson, R. Krishnamurthy, S. Borkar. "A 2.9Tb/s 8W 64-core circuit-switched network-on-chip in 45nm CMOS." *34th European Solid-State Circuits Conference,* 2008. *ESSCIRC,* pages 182-185, 15-19 Sept. 2008.

Acknowledgements

We thank the entire Advanced Microprocessor Research team at Intel for flawless execution of the chip, we thank Tim Mattson and Rob Van Der Wijngaart for workloads, and Joe Schutz, Greg Taylor, Matt Haycock, and Justin Rattner for guidance and encouragement.

More information, including current and past issues of Intel Technology Journal, can be found at:
http://developer.intel.com/technology/itj/index.htm

ISBN 978-1-934053-24-9

9 781934 053249

7 35858 21144 4

$49.95 US