Multithreaded Processors.

Article *in* The Computer Journal · March 2002

Source: DBLP

CITATIONS
42

READS
845

3 authors, including:

Borut Robic
University of Ljubljana
112 PUBLICATIONS **1,155** CITATIONS
SEE PROFILE

Jurij Silc
Jožef Stefan Institute
118 PUBLICATIONS **1,122** CITATIONS
SEE PROFILE

Multithreaded Processors

THEO UNGERER[1], BORUT ROBIČ[2] AND JURIJ ŠILC[3]

[1]*University of Augsburg, Department of Computer Science, Augsburg, Germany*
[2]*University of Ljubljana, Faculty of Computer and Information Science, Ljubljana, Slovenia*
[3]*Jožef Stefan Institute, Computer Systems Department, Ljubljana, Slovenia*
Email: Theo.Ungerer@informatik.uni-augsburg.de

The instruction-level parallelism found in a conventional instruction stream is limited. Studies have shown the limits of processor utilization even for today's superscalar microprocessors. One solution is the additional utilization of more coarse-grained parallelism. The main approaches are the (single) chip multiprocessor and the multithreaded processor which optimize the throughput of multiprogramming workloads rather than single-thread performance. The chip multiprocessor integrates two or more complete processors on a single chip. Every unit of a processor is duplicated and used independently of its copies on the chip. In contrast, the multithreaded processor is able to pursue two or more threads of control in parallel within the processor pipeline. Unused instruction slots, which arise from pipelined execution of single-threaded programs by a contemporary microprocessor, are filled by instructions of other threads within a multithreaded processor. The execution units are multiplexed between the threads in the register sets. Underutilization of a superscalar processor due to missing instruction-level parallelism can be overcome by simultaneous multithreading, where a processor can issue multiple instructions from multiple threads each cycle. Simultaneous multithreaded processors combine the multithreading technique with a wide-issue superscalar processor such that the full issue bandwidth is utilized by potentially issuing instructions from different threads simultaneously. This survey paper explains and classifies the various multithreading techniques in research and in commercial microprocessors and compares multithreaded processors with chip multiprocessors.

Received 11 May 2001; revised 20 December 2001

1. INTRODUCTION

VLSI technology will allow future microprocessors to have an issue bandwidth of 8–32 instructions per cycle [1, 2]. As the issue rate of future microprocessors increases, the compiler or the hardware will have to extract more *instruction-level parallelism* (ILP) from a sequential program. However, ILP found in a conventional instruction stream is limited. ILP studies which allow branch speculation for a single control flow have reported parallelism of around 7 instructions per cycle (IPC) with infinite resources [3, 4] and around 4 IPC with large sets of resources (e.g. 8 to 16 execution units) [5]. Contemporary high-performance microprocessors therefore exploit speculative parallelism by dynamic branch prediction and speculative execution of the predicted branch path to increase single-thread performance.

Research into future microarchitectures—exemplified by the proposal of a *superspeculative microprocessor* [6]—has additionally looked at the prediction of data dependences, source operand values, value strides, address aliases and load values with speculative execution applying the predicted values [7, 8, 9, 10]. The superspeculative microarchitecture technique is applied to increase the performance of a single program thread by means of branch and value speculation techniques. Only instructions of a single thread of control are in execution.

Multithreading pursues a different set of solutions by utilizing coarse-grained parallelism [11, 12, 13]. A multithreaded processor is able to concurrently execute instructions of different threads of control within a single pipeline. Depending on the architectural approach, multithreading is applied either to increase performance of a single program thread by implicitly utilizing parallelism which is more coarse-grained than ILP (so-called implicit multithreading) or to increase performance of a multiprogramming or multithreaded workload (so-called explicit multithreading).

1.1. Notion of a thread

The notion of a thread in the context of multithreaded processors differs from the notion of software threads in multithreaded operating systems. In the case of a multithreaded processor a thread is always viewed as a hardware-supported thread which can be—depending on the specific form of multithreaded processor—a full program (single-threaded UNIX process), a light-weight process (e.g. a POSIX thread) or a compiler- or hardware-generated thread (subordinate microthread, microthread, nanothread etc.). Consequences for multithreaded processor design are as follows.

- The most common coarse-grained thread-level parallelism is to execute multiple processes in parallel. This implies that different logical address spaces have to be

maintained for the different instruction streams that are in execution.

- The parallel execution of multiple threads from a single application usually implies a common address space for all threads. Here threads of control are identical with the threads (light-weight processes) of a multithreaded operating system such as Sun Solaris, IBM AIX and Windows NT, used by today's symmetric multiprocessor workstations and servers. This approach has several architectural advantages for multithreaded processors as well as for chip multiprocessors (CMPs). Cache organization is simplified when a single logical address space is shared. Moreover, thread synchronization, as well as exchange of global variables between threads, can be made very efficient by providing common on-chip memory structures (shared caches or even shared registers). If the threads shared a single program multiple data (SPMD) program structure, a single multiported instruction cache (I-cache) might be considered. However, although most desktop applications like Acrobat, Netscape, Photoshop, Powerpoint and Winword today use three to eight threads, most thread activity in these program systems is restricted to a single main thread [14]. This drawback may be alleviated by parallelizing compilers in conjunction with regularly structured application problems such as, for example, numerical problems or multimedia applications.
- Execution of a sequential application may be accelerated by extracting threads of control either statically by the compiler or dynamically by hardware from a single instruction stream (exemplified by the implicit multithreaded architectures, see below).

1.2. Implicit multithreading

One set of solutions for increasing the performance of sequential programs is to apply an even higher degree of speculation in combination with a functional partitioning of the processor. Here thread-level parallelism is utilized, typically in combination with thread-level speculation [15]. A thread in such architectures refers to any contiguous region of the static or dynamic instruction sequence. The term *implicit multithreaded* architecture refers to any architecture that can concurrently execute several threads from a single sequential program. The threads may be obtained with or without the help of the compiler.

Examples of such architectural approaches are the multiscalar [16, 17, 18, 19], the trace processor [20, 21, 22], the single-program speculative multithreading architecture [23], the superthreaded architecture [24, 25], the dynamic multithreading processor [26] and the speculative multithreaded processor [27]. Some of these approaches may rather be viewed as very closely coupled CMPs, because multiple subordinate processing units execute different threads under the control of a single sequencer unit, whereas a multithreaded processor may be characterized by a single processing unit with a single or multiple-issue pipeline able to process instructions of different threads concurrently.

A number of research projects have surveyed *eager execution*—dual path execution of branches. They extend either superscalar or simultaneous multithreaded processors. All need some kind of architecture that is able to pursue two threads in parallel typically by assuming multithreaded processor hardware.

Closely related to implicit multithreading are architectural proposals that only slightly enhance a superscalar processor by the ability to pursue two or more threads only for a short time. In principle, *predication* is the first step in this direction. An enhanced form of predication is able to issue and execute a predicated instruction even if the predicate is not yet solved. A further step is dynamic predication [28] as applied for the Polypath architecture [29] that is a superscalar enhanced to handle multiple threads internally. Another step to multithreading is simultaneous subordinate microthreading [30] which is a modification of superscalars to run threads at a microprogram level concurrently.

1.3. Explicit multithreading

The solution surveyed in this paper is the utilization of coarser-grained parallelism by *(single) CMPs* and *multithreaded processors*. A CMP (sometimes called a *multiprocessor chip*) integrates two or more complete processors on a single chip. Therefore, every unit of a processor is duplicated and used independently of its copies on the chip.

In contrast, a multithreaded processor interleaves the execution of instructions of different threads of control in the same pipeline. Therefore, multiple program counters are available in the fetch unit and the multiple contexts are often stored in different register sets on the chip. The execution units are multiplexed between the thread contexts that are loaded in the register sets. The latencies that arise in the computation of a single instruction stream are filled by computations of another thread. This ability is in contrast to reduced instruction set computer (RISC) processors or today's superscalar processors, which use busy waiting or a time-consuming, operating system-based thread switch. Multithreaded processors tolerate memory latencies by overlapping the long-latency operations of one thread with the execution of other threads—in contrast to the CMP approach.

Depending on the specific multithreaded processor design, either a single-issue instruction pipeline (as in scalar RISC processors) is used or instructions from different instruction streams are issued simultaneously. The latter are called simultaneous multithreaded (SMT) processors and combine the multithreading technique with a wide-issue superscalar processor such that the full issue bandwidth is utilized by potentially issuing instructions from different threads simultaneously.

The focus of this survey paper is, in particular, on multithreaded processors that are designed to simultaneously execute threads of the same or of different processes.

We call such processors *explicit multithreaded* in contrast to the implicit multithreaded processors mentioned earlier. Explicit multithreaded processors are able to increase the performance of a multiprogramming workload. However, single-thread performance may slightly decrease when compared to a single-threaded processor. Note that explicit multithreaded processors aim at a low execution time of a multithreaded workload, while superscalar and implicit multithreaded processors aim at a low execution time of a single program.

1.4. The origins of multithreading

The first multithreaded processors in the 1970s and 1980s [11] applied multithreading at the user thread level to solve the memory access latency problem that arises for each memory access after a cache miss—in particular, when a processor of a shared-memory multiprocessor accesses a shared-memory variable located in a remote-memory module. To perform such a remote-memory access in a distributed shared-memory (DSM) multiprocessor, the processor issues a request message to the communication network that couples the processor–memory nodes. The request message traverses the network to the memory module. The memory module reads the value, respectively the cache line, and sends a result message back to the requesting processor. Depending on the coherence scheme, further actions may be necessary to guarantee memory consistency or cache coherence before the requested value or cache line is sent back [31]. The interval between the sending of the request message until the return of the result message is called *(remote) memory access latency* or often just the *latency*. Latencies that arise in a pipeline are defined with a wider scope—for example, covering also long-latency operations like `div` or latencies due to branch interlocking. The latency becomes a problem if the processor spends a large fraction of its time sitting idle and waiting for remote accesses to complete.

Load access latencies measured on an Alpha Server 4100 SMP with four 300 MHz Alpha 21164 processors are [32]: seven cycles for a primary cache miss which hits in the on-chip secondary cache of the 21164 processor, 21 cycles for a secondary cache miss which hits in the tertiary (board-level) cache, 80 cycles for a miss that is served by the memory and 125 cycles for a dirty miss, i.e. a miss that has to be served from another processor's cache memory. For DSM multiprocessors supporting up to 1024 processors we can expect latencies of 200 cycles or more. Furthermore, memory access latencies are expected to increase over time as on-chip speeds are increased more quickly than off-chip speeds.

Older multithreaded processor approaches from the 1980s usually extend scalar RISC processors by a multithreading technique and focus at effectively bridging very long remote memory access latencies. Such processors will only be useful as processor nodes in DSM multiprocessors. However, developing a processor that is specifically designed for DSM multiprocessors is commonly regarded as too expensive. Multiprocessors today comprise standard off-the-shelf microprocessors and almost never specifically designed processors (with the exception of Cray MTA [33]). Therefore, newer multithreaded processor approaches also strive for tolerating smaller latencies that arise from primary cache misses that hit in secondary cache, from long-latency operations or even from unpredictable branches.

Another root of multithreading comes from dataflow architectures. Viewed from a dataflow perspective a *single-threaded* architecture is characterized by the computation that conceptually moves forward one step at a time through a sequence of states, each step corresponding to the execution of one enabled instruction. The state of a single-threaded machine consists of the *memory state* (program memory, data memory, stack) and the *processor state* which consists of the *continuation* or *activity specifier* (program counter, stack pointer) and the *register context* (a set of register contents). The processor state is also called the *context* of a thread. Today most processors are of a single-threaded processor architecture.

According to Dennis and Gao [34], a *multithreaded* architecture differs from a single-threaded architecture in that there may be several enabled instructions from different threads which all are candidates for execution. Similar to the single-threaded machine, the state of the multithreaded machine consists of the memory state and the processor state; the latter, however, consists of a *collection* of activity specifiers and a *collection* of register contexts. A thread is a sequentially ordered block of instructions with a grain size greater than one (to distinguish multithreaded architectures from fine-grained dataflow architectures).

Another notion is the distinction between *blocking* and *non-blocking* threads. A *non-blocking thread* is formed such that its evaluation proceeds without blocking the processor pipeline (for instance, by remote memory accesses, cache misses or synchronization waits). Evaluation of a non-blocking thread starts as soon as all input operands are available, which is usually detected by some kind of dataflow principle. Thread switching is controlled by the compiler harnessing the idea of rescheduling, rather than blocking, when waiting for data. Access to remote data is organized in a split-phase manner by one thread sending the access request to memory and another thread activating when its data are available. Thus a program is compiled into many very small threads activating each other when data become available. The same mechanisms may also be used to synchronize interprocess communications to awaiting threads, thereby alleviating operating systems overhead. In contrast, a *blocking thread* might be blocked during execution by remote memory accesses or cache misses. (During synchronization, a switch is necessary to avoid deadlock.) The waiting time, during which the pipeline is blocked, is lost when using a von Neumann processor, but can be efficiently bridged by a fast context switch to another thread in a multithreaded processor. Switching to another thread in a single-threaded processor usually exhibits too much context switching overhead to mask the

latency efficiently. The original thread can be resumed when the reason for blocking is removed.

Use of non-blocking threads typically leads to many small threads that are appropriate for execution by a hybrid dataflow computer or by a multithreaded architecture that is closely related to hybrid dataflow. Blocking threads may just be the threads (e.g. P(OSIX) threads or Solaris threads) or whole UNIX processes of a multithreaded UNIX-based operating system, but may also be even smaller microthreads generated by a compiler to utilize the potentials of a multithreaded processor.

Note that we exclude in this survey hybrid dataflow architectures that are designed for the execution of non-blocking threads. Although these architectures are often called *multithreaded*, we have categorized them in a previous paper [35] as threaded dataflow or large-grain dataflow because a dataflow principle is applied to start the execution of non-blocking threads. Thus, multithreaded architectures (in the more narrow sense applied here) stem from the modification of scalar RISC, very long instruction word (VLIW) or even superscalar RISC processors.

1.5. Paper organization

The rest of this survey paper is organized as follows. Section 2 describes the various explicit multithreading approaches, which can be classified into interleaved multithreading, blocked multithreading and simultaneous multithreading. Further multithreading approaches, in particular, implicit multithreading, multithreading and scheduled dataflow, dual path branch execution models and multithreading approaches to signal processing and real-time event handling are briefly described in Section 3. Section 4 introduces the chip multiprocessors approach, which places a small number of distinct processors on a single chip. A comparison between the simultaneous multithreaded approach and the chip multiprocessor approach is given in Section 5. Finally, the main conclusions are summarized in Section 6.

2. MULTITHREADED PROCESSORS

2.1. Principal approaches

The minimal requirement for a multithreaded processor is the ability to pursue two or more threads of control in parallel within the processor pipeline—i.e. it must provide two or more independent program counters—and a mechanism that triggers a thread switch. Thread-switch overhead must be very low, from zero to only a few cycles. A fast context switch is supported by multiple program counters and often by multiple register sets on the processor chip.

The following principle approaches to multithreaded processors exist.

- *Interleaved multithreading technique*. An instruction of another thread is fetched and fed into the execution pipeline at each processor cycle (see Section 2.2).
- *Blocked multithreading technique*. The instructions of a thread are executed successively until an event occurs that may cause latency. This event induces a context switch (see Section 2.3).
- *Simultaneous multithreading*. The wide superscalar instruction issue is combined with the multiple-context approach. Instructions are simultaneously issued from multiple threads to the execution units of a superscalar processor (see Section 2.4).

Before we present the different multithreading approaches in detail, we briefly review the main principles of architectural approaches that exploit instruction-level parallelism and thread-level parallelism.

Figures 1a–1c demonstrates the different approaches possible with scalar (i.e. single-issue) processors: single-threaded (Figure 1a), with interleaved multithreading (Figure 1b) and with blocked multithreading (Figure 1c).

Another way to look at latencies that arise in a pipelined execution is the *opportunity cost* in terms of the instructions that might be processed while the pipeline is interlocked, for example, waiting for a remote reference to return. The opportunity cost of single-issue processors is the number of cycles lost by latencies. Multiple-issue processors (e.g. superscalar, VLIW, etc.) potentially execute more than one IPC. In this case, it should be clear that latency cycles are cycles where no instruction can be issued and that issue bandwidth is the maximum number of instructions that can be issued per cycle. Thus, the opportunity cost for multiple-issue processors is the product of the latency cycles and the issue bandwidth plus the number of unfilled places in not fully filled issue slots. We expect that future single-threaded processors will continue to exploit further superscalar or other multiple-issue techniques, and thus further increase the opportunity cost of remote-memory accesses.

Figures 1d–1i demonstrate the different approaches possible with four-issue processors: single-threaded superscalar (Figure 1d), single-threaded VLIW (Figure 1g), superscalar with interleaved multithreading (Figure 1e), superscalar with blocked multithreading (Figure 1f), VLIW with interleaved multithreading (Figure 1h) and VLIW with blocked multithreading (Figure 1i).

The opportunity cost in single-threaded superscalar (Figure 1d) can be easily determined as the number of empty issue slots. It consists of horizontal losses (the number of empty places in the not fully filled issue slot) and the even more harmful vertical losses (cycles where no instructions can be issued). In VLIW processors (Figure 1g), horizontal losses appear as no-op operations. The opportunity cost of single-threaded VLIW is about the same as single-threaded superscalar. Interleaved multithreading superscalar (Figure 1e) and interleaved multithreading VLIW (Figure 1h) are able to fill the vertical losses of the single-threaded models by instructions of other threads, but not the horizontal losses. Further design possibilities, blocked multithreading superscalar (Figure 1f) and blocked multithreading VLIW (Figure 1i) models would fill several succeeding cycles with instructions of the same

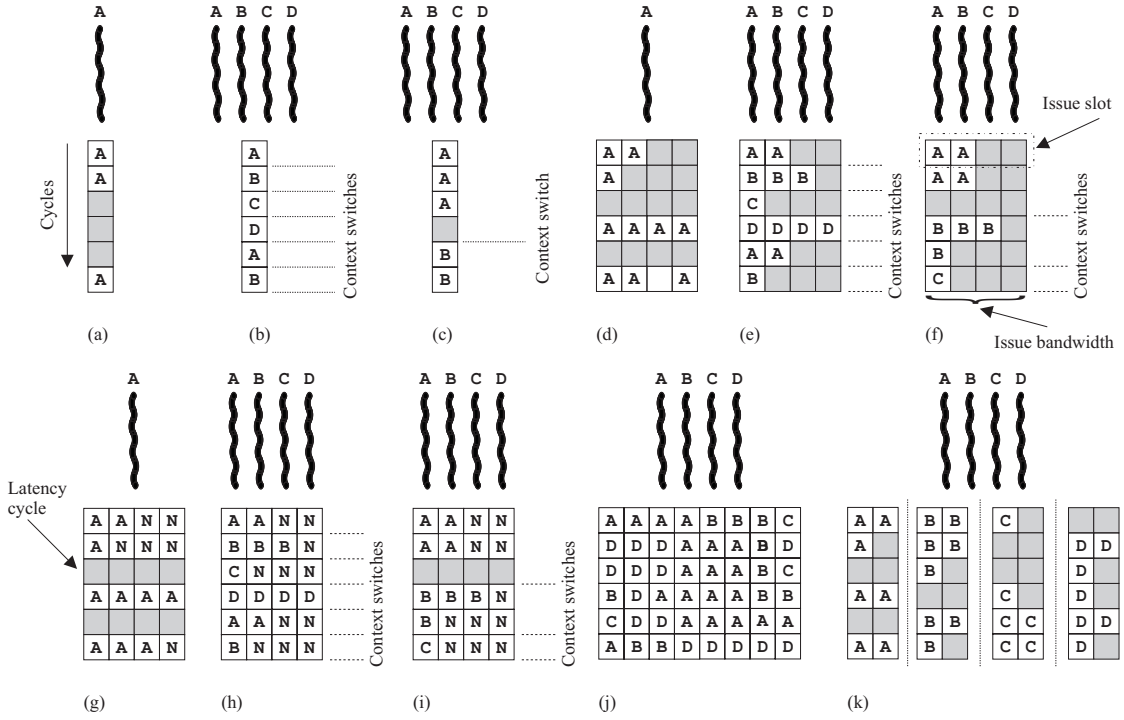

FIGURE 1. Different approaches possible with scalar processors: (a) single-threaded scalar; (b) interleaved multithreading scalar; (c) blocked multithreading scalar. Different approaches possible with multiple-issue processors: (d) superscalar; (e) interleaved multithreading superscalar; (f) blocked multithreading superscalar; (g) VLIW; (h) interleaved multithreading VLIW; (i) blocked multithreading VLIW; (j) simultaneous multithreading; and (k) chip multiprocessor. Each row represents the issue slots for a single execution cycle. An empty box represents an unused slot; `N` stands for a no-op operation.

thread before context switching. The switching event is more difficult to implement and a context-switching overhead of one to several cycles might arise.

Figures 1j and 1k demonstrate a four-threaded eight-issue SMT processor (Figure 1j) and a CMP with four two-issue processors (Figure 1k). The processor model in Figure 1j exploits ILP by selecting instructions from any thread (four in this case) that can potentially issue. If one thread has high ILP, it may fill all horizontal slots depending on the issue strategy of the SMT processor. If multiple threads each have low ILP, instructions of several threads can be issued and executed simultaneously. In the CMP with four two-issue CPUs on a single chip that is represented in Figure 1k, each CPU is assigned a thread from which it can issue up to two instructions each cycle. Thus, each CPU has the same opportunity cost as in a two-issue superscalar model. The CMP is not able to hide latencies by issuing instructions of other threads. However, because horizontal losses will be smaller for two-issue than for high-bandwidth superscalars, a CMP of four two-issue processors will reach a higher utilization than an eight-issue superscalar processor (see Table 3 and [36]).

2.2. Interleaved multithreading

In the *interleaved multithreading* model (also called *fine-grain multithreading*) the processor switches to a different thread after each instruction fetch. In principle, an instruction of a thread is fed into the pipeline after the retirement of the previous instruction of that thread. Since interleaved multithreading eliminates control and data dependences between instructions in the pipeline, pipeline hazards cannot arise and the processor pipeline can be easily built without the necessity of complex forwarding paths. This leads to a very simple and therefore potentially very fast pipeline—no hardware interlocking is necessary. Moreover, the context-switching overhead is zero cycles. Memory latency is tolerated by not scheduling a thread until the memory transaction has completed. This model requires at least as many threads as pipeline stages in the processor. Interleaving the instructions from many threads limits the processing power accessible to a single thread, thereby degrading the single-thread performance. There are two possibilities to overcome this deficiency.

- The *dependence lookahead technique* (e.g. used in Cray MTA [33]) adds several bits to each instruction format in the ISA. The additional opcode bits allow the compiler to state the number of instructions directly following in program order that are not data- or control-dependent on the instruction being executed. This allows the instruction scheduler in the interleaved multithreading processor to feed non-data- or control-dependent instructions of the same thread successively into the pipeline. The dependence lookahead technique may be applied to speed up single-thread performance

or in the case where a workload does not comprise enough threads.

- The *interleaving technique* [37] adds caching and full pipeline interlocks to the interleaved multithreading approach. Contexts are interleaved on a cycle-by-cycle basis, yet a single-thread context is also efficiently supported.

The most well-known examples of interleaved multithreading processors were used in the Heterogeneous Element Processor (HEP) [38], the Horizon [39] and the Cray Multithreaded Architecture (MTA) [33] multiprocessors. The HEP system had up to 16 processors while the other two consisted up to 256 processors. Each of these processors supported up to 128 threads. While HEP uses instruction lookahead only if there is no other work, the Horizon and Cray MTA employ explicit dependence lookahead techniques.

Further interleaved multithreading processor proposals include the Multilisp Architecture for Symbolic Applications (MASA) [40], the SB-PRAM/HPP [41] and MicroUnity's MediaProcessor [42], as an example of a multithreaded signal processor. The SB-PRAM is 32-threaded and the MediaProcessor interleaves five contexts.

In principle, interleaved multithreading can also be combined with a superscalar instruction issue, but simulations [36] confirm the intuition that SMT is the more efficient technique (see Section 5).

Some machines as well as ongoing projects that use interleaved multithreading are described in more detail below.

2.2.1. *HEP*

The HEP system [13] was a multiple instruction multiple data (MIMD) shared-memory multiprocessor system developed by Denelcor Inc., Denver, CO between 1978 and 1985, and it was a pioneering example of a multithreaded machine. The HEP system was designed to have up to 16 processors (Figure 2) with up to 128 threads per processor. The 128 threads were supported by replicating the register files 128 times (32 general registers and eight branch target registers per thread). The processor pipeline had eight stages, matching the number of processor cycles necessary to fetch a data item from memory in register. There were three types of operations: a memory reference operation (M-op), an arithmetic/logical operation (A-op) and a branch or simple arithmetic/logical operation (C-op). Up to eight threads were in execution concurrently within a single HEP processor. However, the pipeline did not allow more than one memory, branch or divide instruction to be in the pipeline at the given time. If thread queues were all empty, the next instruction from the last thread dispatched was examined for independence from the previous instruction and if so, the instruction was also issued. In contrast to all other interleaved multithreading processors, all threads within a HEP processor shared the same register set. Multiple processors and data memories were interconnected via a pipelined switch and any register-memory or data-memory location

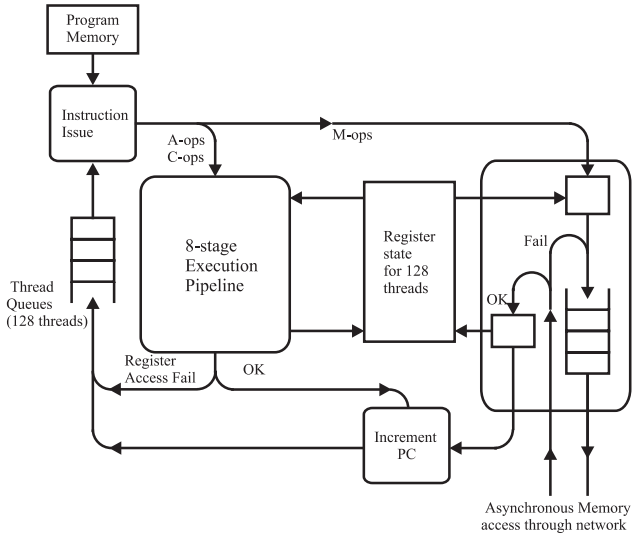

FIGURE 2. The HEP processor (A-op, arithmetic/logical operation; C-op, branch or simple arithmetic/logical operation; M-op, memory reference operation).

could be used to synchronize two processes on a producer–consumer basis by a full/empty bit synchronization on a data memory word.

2.2.2. *MASA*

The MASA [40] was an interleaved multithreading processor proposal for parallel symbolic computation with various features intended for effective Multilisp program execution. MASA featured a tagged architecture, multiple contexts, fast trap handling and a synchronization bit in every memory word. Its principal novelty was the use of multiple contexts both to support interleaved execution from separate instruction streams and to speed up procedure calls and trap handling (in the same manner as register windows).

2.2.3. *Cray MTA*

The Cray MTA computer [33] features a powerful VLIW instruction set, uniform access time from any processor to any memory location and zero-cost synchronization and swapping between threads of control. It was designed by the Tera Computer Company, Seattle, WA, which has been a part of Cray since April 2000.

The MTA inherits much of the design philosophy of the HEP as well as a (paper) architecture called Horizon [39]. The latter was designed for up to 256 processors and up to 512 memory modules in a $16 \times 16 \times 6$ node internal network. Horizon (like HEP) employed a global address space and memory-based synchronization through the use of full/empty bits at each location. Each processor supported 128 independent instruction streams by 128 register sets with context switches occurring at every clock cycle. Unlike HEP, Horizon allows multiple memory operations from a thread to be in the pipeline simultaneously.

MTA systems are constructed from resource modules, each of which contains up to six resources. A resource can

be a *computational processor* (CP), an *I/O processor* (IOP), an *I/O cache* (IOC) unit and either two or four *memory units* (MUs). Each resource is individually connected to a separate routing node in the partially connected 3-D torus interconnection network. Each routing node has three or four communication ports and a resource port. There are several routing nodes per CP, rather than the several CPs per routing node. In particular, the number of routing nodes is at least $p^{3/2}$, where p is the number of CPs. This allows bisection bandwidth to scale linearly with p, while the network latency scales with $p^{1/2}$. The communication link is capable of supporting data transfers to and from memory on each clock tick in both directions, as are all of the links between the routing nodes themselves.

The Cray MTA custom chip CP (Figure 3) is a multithreaded VLIW pipelined processor using the interleaved multithreading technique. Each thread is associated with one 64-bit stream status word, 32 64-bit general registers and eight 64-bit target registers. The processor may switch context every cycle (3 ns cycle period) between as many as 128 distinct threads (called *streams* by the designers of the MTA), thereby hiding up to 128 cycles (384 ns) of memory latency. Since the context switching is so fast, the processor has no time to swap the processor state. Instead, it has multiples of 128 of everything, i.e. 128 stream status words, 4096 general registers and 1024 target registers. Dependences between instructions are explicitly encoded by the compiler using *explicit dependence lookahead*. Each instruction contains a 3-bit lookahead field that explicitly specifies how many instructions from this thread will be issued before encountering an instruction that depends on the current one. Since seven is the maximum possible lookahead value, at most eight instructions (i.e. 24 operations) from each thread can be concurrently executing in different stages of a processor's pipeline. In addition, each thread can issue as many as eight memory references without waiting for earlier ones to finish, further augmenting the memory latency tolerance of the processor. The CP has a load/store architecture with three addressing modes and 32 general-purpose 64-bit registers. The 3-wide VLIW instructions are 64 bits. Three operations can be executed simultaneously per instruction: a memory reference operation (M-op), an arithmetic/logical operation (A-op) and a branch or simple arithmetic/logical operation (C-op). If more than one operation in an instruction specifies the same register or setting of condition codes, then M-op has higher priority than A-op which in turn has higher priority than C-op.

The clock speed is nominally 333 MHz, giving each processor a data path bandwidth of 10^9 64-bit results per second and a peak performance of 1 Gflops. The peak memory bandwidth is 2.67 Gbyte s^{-1} and it is claimed that the processor sustains well over 95% of that rate.

Every processor has a clock register that is synchronized exactly with its counterparts in the other processors and counts up once per cycle. In addition, the processor counts the total number of unused instruction issue slots (measuring the degree of underutilization of the processor) and the time integral of the number of instruction streams ready to issue

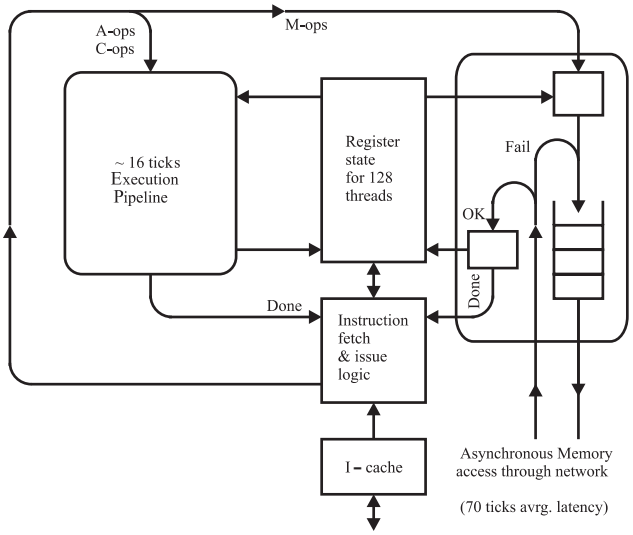

FIGURE 3. The MTA computational processor (A-op, arithmetic/logical operation; C-op, branch or simple arithmetic/logical operation; M-op, memory reference operation).

(measuring the degree of overutilization of the processor). All three counters are user-readable in a single unprivileged operation. Eight counters are implemented in each of the protection domains of the processor. All are user-readable in a single unprivileged operation. Four of these counters accumulate the number of instructions issued, the number of memory references, the time integral of the number of instruction streams and the time integral of the number of messages in the network. These counters are also used for job accounting. The other four counters are configurable to accumulate events from any four of a large set of additional sources within the processor, including memory operations, jumps, traps and so on.

Thus, the Cray MTA exploits parallelism at all levels, from fine-grained ILP within a single processor to parallel programming across processors, to multiprogramming among several applications simultaneously. Consequently, processor scheduling occurs at many levels and managing these levels poses unique and challenging scheduling concerns [43].

After many delays the MTA reached the market in December 1997 when a single-processor system (clock speed 145 MHz) was delivered to the San Diego Supercomputer Center. In May 1999, the system was upgraded to eight processors and a network with 64 routing nodes. Each processor runs at 260 MHz and is theoretically capable of 780 Mflops.

Early MTA systems had been built using GaAs technology for all logic design. As a result of the semiconductor market's focus on CMOS technology for computer systems, Cray started a transition to using CMOS technology in the MTA.

2.2.4. M-Machine

The MIT M-Machine [44] supports both public and private registers for each thread and uses interleaved

multithreading. Each processor supports four hardware resident user V-threads and each V-thread supports four resident H-threads. All the H-threads in a given V-thread share the same address space and each H-thread instruction is a three-wide VLIW. Event and exception handling are each performed by a separate V-thread. Swapping processor-resident V-threads with one stored in memory requires about 150 cycles (1.5 μs). The M-Machine (like HEP, Horizon and Cray MTA) employs full-empty bits for efficient, low-level synchronization. Moreover it supports message passing and guarded pointers with base and bounds for access control and memory protection.

2.2.5. SB-PRAM and HPP

The SB-PRAM (SB stands for Saarbrücken) [45] or High-Performance PRAM (HPP) [41] is a MIMD parallel computer with shared address space and uniform memory access time due to its motivation: building a multiprocessor that is as close as possible to the theoretical machine model CRCW-PRAM. Processor and memory modules are connected by a butterfly network. Network latency is hidden by pipelining several so-called virtual processors on one physical processor node in interleaved multithreading mode. Instructions of 32 so-called virtual processors are interleaved round-robin in a single SB-PRAM processor, which is therefore classified as a 32-threaded interleaved multithreading processor. The project is in progress at the University of Saarland, Saarbrücken, Germany. A first prototype was running with four processors and this was recently upgraded to 64 processors which in total support up to 2048 threads.

2.2.6. SPELL

In 1997, the Jet Propulsion Laboratory in collaboration with several other institutions (The California Institute of Technology, Princeton University, University of Notre Dame, University of Delaware, The State University of New York at Stonybrook, Tera Computer Company (now Cray), Argone National Laboratories) initiated a project whose aim is to design the first petaflops computer by 2005–2007. The computer will be based on radically new hardware technologies such as superconductive processors, optical holographic storages and optical packet switched networks [46]. There will be 4096 superconductive processors, called SPELL processors. A SPELL processor consists of 16 multistream units, where each multistream unit is a 64-bit, deeply pipelined integer processor capable of executing up to eight parallel threads. As a result, each SPELL is capable of running in parallel up to 128 threads, arranged in 16 groups of eight threads [47].

2.3. Blocked multithreading

The *blocked multithreading* approach (sometimes also called *coarse-grain multithreading*) executes a single thread until it reaches a situation that triggers a context switch. Usually such a situation arises when the instruction execution reaches a long-latency operation or a situation where a latency may arise. Compared to the interleaved multithreading technique, a smaller number of threads is needed and a single thread can execute at full speed until the next context switch. Single-thread performance is similar to the performance of a comparable processor without multithreading.

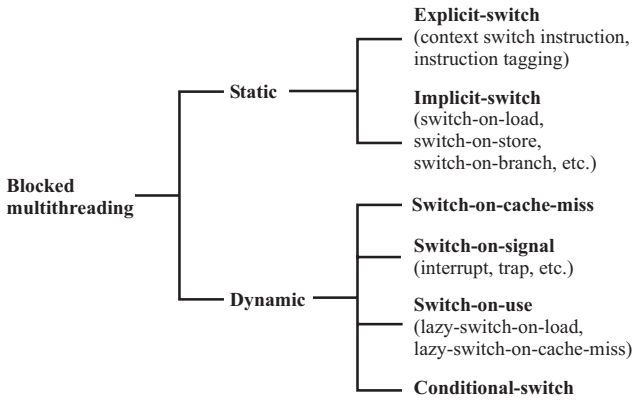

FIGURE 4. Blocked multithreading models.

In the following we classify blocked multithreading processors by the event that triggers a context switch (Figure 4) [48].

- *Static* models. A context switch occurs each time the same instruction is executed in the instruction stream. The context switch is encoded by the compiler. The main advantage of this technique is that context switching can be triggered already in the fetch stage of the pipeline. The context switching overhead is one cycle (if the fetched instruction triggers the context switch and is discarded), zero cycles (if the fetched instruction triggers a context switch but is still executed in the pipeline) and almost zero cycles (if a context switch buffer is applied, see the Rhamma processor below in this section). There are two main variations of the static blocked multithreading model.
  - The *explicit-switch* static model, where context switch is triggered explicitly. This model is simple to implement. The *context switch instruction* model has additional instruction for triggering context switch. However, this instruction causes one-cycle context switching overhead. In the *instruction tagging* model the compiler tags instructions and so implements an explicit context switch, but it does this without losing a cycle or dividing instructions into classes.
  - The *implicit-switch* model, where each instruction belongs to a specific instruction class and a context switch decision depends on the instruction class of the fetched instruction. Instruction classes that cause context switch include load, store and branch instructions. The *switch-on-load* static model switches after each load instruction to bridge memory access latency. However, assuming an on-chip data cache (D-cache), the

thread switch occurs more often than necessary, which makes an extremely fast context switch necessary, preferably with zero-cycle context switch overhead. The *switch-on-store* static model switches after store instructions. The model may be used to support the implementation of sequential consistency so that the next memory access instruction can only be performed after the store has completed in memory. The *switch-on-branch* static model switches after branch instructions. The model can be applied to simplify processor design by renouncing branch prediction and speculative execution. The branch misspeculation penalty is avoided, but single-thread performance is decreased. However, it may be effective for programs with a high percentage of branches that are hard to predict or are even unpredictable.

- *Dynamic* models. The context switch is triggered by a dynamic event. In general, all the instructions between the fetch stage and the stage that triggers the context switch are discarded, leading to a higher context switch overhead than static context switch models. Several dynamic models can be defined.
  - The *switch-on-cache-miss* dynamic model switches the context if a load or store misses in the cache. The idea is that only those loads that miss in the cache and those stores that cannot be buffered have long latencies and cause context switches. Such a context switch is detected in a late stage of the pipeline. A large number of subsequent instructions have already entered the pipeline and must be discarded. Thus context switch overhead is considerably increased.
  - The *switch-on-signal* dynamic model switches context on the occurrence of a specific signal, for example, signaling an interrupt, trap or message arrival.
  - The *switch-on-use* dynamic model switches when an instruction tries to use the still missing value from a load (which, e.g., missed in the cache). For example, when a compiler schedules instructions so that a load from shared memory is issued several cycles before the value is used, the context switch should not occur until the actual use of the value. To implement the switch-on-use model, a *valid bit* is added to each register (by a simple form of scoreboard). The bit is cleared when a load to the corresponding register is issued and set when the result returns from the network. A thread switches context if it needs a value from a register whose valid bit is still cleared. This model can also be seen as a *lazy* model that extends either the switch-on-load static model (called *lazy-switch-on-load*) or the switch-on-cache-miss dynamic model (called *lazy-switch-on-cache-miss*).
  - The *conditional-switch* dynamic model couples an explicit switch instruction with a condition. The context is switched only when the condition is fulfilled, otherwise the context switch is ignored. A conditional-switch instruction may be used, for example, after a group of load/store instructions. The context switch is ignored if all load instructions (in the preceding group) hit the cache; otherwise, the context switch is performed. Moreover, a conditional-switch instruction could also be added between a group of loads and their subsequent use to realize a lazy context switch (instead of implementing the switch-on-use model).

The explicit-switch, conditional-switch and switch-on-signal techniques enhance the instruction set architecture (ISA) by additional instructions. The implicit-switch technique may favor a specific ISA encoding to simplify instruction class detection. All other techniques are microarchitectural techniques without the necessity of ISA changes.

A previous classification [49, 50] concerns multithreading techniques only in a shared-memory multiprocessor environment and is restricted to only a few of the variations of the multithreading techniques described above. In particular, the switch-on-load model in [50] switches only on instructions that load data from remote memory, while storing data in remote memory does not cause context switching. Likewise, the *switch-on-miss* model is defined so that the context is only switched if a load from remote memory misses in the cache.

Several well-known processors use the blocked multithreading approach. The MIT Sparcle [51] and the MSparc processors [52] use both the switch-on-cache-miss and the switch-on-signal dynamic models. The CHoPP 1 [53] uses the switch-on-cache-miss dynamic model, while Rhamma [54, 55] applies several static and dynamic block multithreading models. These, as well as some other processors using the blocked multithreading approach, are described below.

2.3.1. CHoPP 1

The CHoPP 1 [53] was designed by the CHoPP Sullivan Computer Corporation (ANTs since 1999) in 1987. The system was a shared-memory MIMD with up to 16 powerful processors. High sequential performance is due to the issuing of multiple instructions on each clock cycle, zero-delay branch instructions and fast execution of individual instructions. Each processor can support up to 64 threads and uses the switch-on-cache-miss dynamic interleaving model.

2.3.2. MDP in J-Machine

The MIT Jellybean Machine (J-Machine) [56] is so-called because it is to be built entirely of a large number of 'jellybean' components. The initial version uses an $8 \times 8 \times 16$ cube network, with possibilities of expanding to

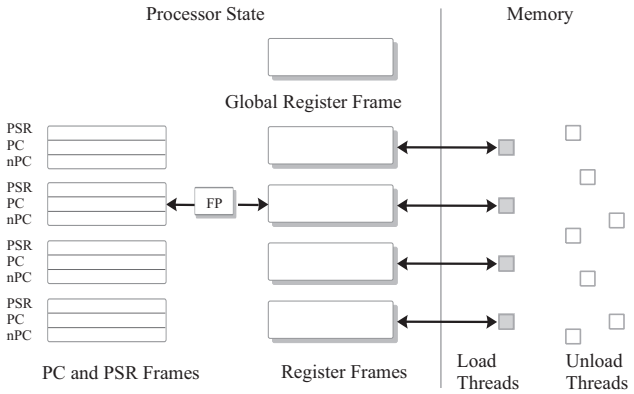

FIGURE 5. Sparcle register usage (PSR, Processor Status Register; PC, Program Counter; nPC, Next Program Counter; FP, Frame Pointer).

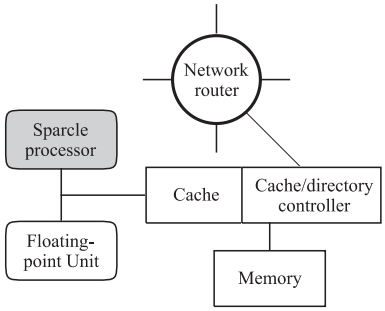

FIGURE 6. An Alewife node.

64k nodes. The 'jellybeans' are message-driven processor (MDP) chips, each of which has a 36-bit processor, a 4k word memory and a router with communication ports for bidirectional transmissions in three dimensions. External memory of up to 1M words can be added per processor. The MDP creates a task for each arriving message. In the prototype, each MDP chip has four external memory chips that provide 256k memory words. However, access is through a 12-bit data bus and, with an error correcting cycle, the access time is four memory cycles per word. Each communication port has a 9-bit data channel. The routers provide support for automatic routing from source to destination. The latency of a transfer is 2 μs across the prototype machine, assuming no blocking. When a message arrives, a task is created automatically to handle it in 1 μs. Thus, it is possible to implement a shared memory model using message passing, in which a message provides a fetch address and an automatic task sends a reply with the desired data.

2.3.3. MIT Sparcle

The MIT Sparcle processor [51] is derived from a SPARC RISC processor. The eight overlapping register windows of a SPARC processor are organized as four independent non-overlapping thread contexts, each using two windows, one as a register set, the other as a context for trap and message handlers (Figure 5).

Context switches are used only to hide long memory latencies since small pipeline delays are assumed to be hidden by proper ordering of instructions by an optimizing compiler. The MIT Sparcle processor switches to another context in the case of a remote cache miss or a failed synchronization (switch-on-cache-miss and switch-on-signal strategies). Thread switching is triggered by external hardware, i.e. by the cache/directory controller. Reloading of the pipeline and the software implementation of the context switch cause a context switch cost of 14 processor cycles.

The MIT Alewife DSM multiprocessor [57] is based on the multithreaded MIT Sparcle processor. The multiprocessor has been operational since May 1994. A node in the Alewife multiprocessor comprises a Sparcle processor, an external floating-point unit, a cache and a directory-based cache controller that manages cache-coherence, a network router chip and a memory module (Figure 6).

The Alewife multiprocessor uses a low-dimensional direct interconnection network. Despite its distributed-memory architecture, Alewife allows efficient shared-memory programming through a multilayered approach to locality management. Communication latency and network bandwidth requirements are reduced by a directory-based cache-coherence scheme referred to as LimitLESS directories. Latencies still occur although communication locality is enhanced by run-time and compile-time partitioning and placement of data and processes.

2.3.4. MSparc

An approach similar to the MIT Sparcle processor was taken at the University of Oldenburg, Germany with the MSparc processor [52]. MSparc supports up to four contexts on chip and is compatible with standard SPARC processors. Switching is supported by hardware and can be achieved within one processor cycle. However, a four cycle overhead is introduced due to pipeline refill. The multithreading policy is blocked multithreading with the switch-on-cache-miss policy as in the MIT Sparcle processor.

2.3.5. Rhamma

The Rhamma processor [54, 55] was developed between 1993 and 1997 at the University of Karlsruhe, Germany, as an experimental microprocessor that bridges all kinds of latencies by a very fast context switch. The *execution unit* (EU) and *load/store unit* (LSU) are decoupled and work concurrently on different threads (Figure 7). A number of register sets used by different threads are accessed by the LSU as well as the EU. Both units are connected by FIFO buffers for so-called continuations, each denoting the thread tag and the instruction pointer of a thread.

The EU and the LSU are modeled by a four-stage instruction pipeline (instruction fetch, decode, operand fetch and a combined execute and write-back stage). The EU is based on a conventional RISC processor, with an instruction

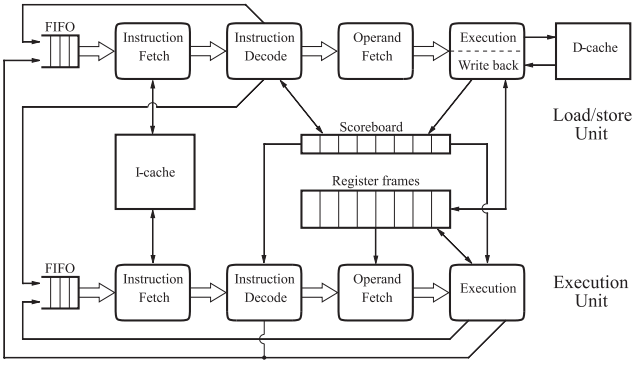

FIGURE 7. Overall structure of the Rhamma processor.

set that is extended by thread management instructions. A scoreboard determines operand availability prior to the execute/write-back stage.

The Rhamma processor combines several static and dynamic blocked multithreading techniques. Due to the decoupling of the execution and the load/store pipelines, a context switch is performed if an instruction is to be fetched but belongs to the other instruction class. The context switch is recognized by the predecoder in the instruction fetch stage by means of a tag in the instruction opcode (implicit-switch technique).

In the case of a load/store, the continuation is stored in the FIFO buffer of the LSU, a new continuation is fetched from the EU's FIFO buffer and the context is switched to the register set given by the new thread tag and the new instruction pointer. The LSU loads and stores data of a different thread in a register set concurrently to the work of the EU. There is a loss of one processor cycle in the EU for each sequence of load/store instructions. Only the first load/store instruction forces a context switch in the EU; succeeding load/store instructions are executed in the LSU.

A context switch is also performed in the EU after the fetch of a control flow (branch, jump or context management) instruction (implicit-switch technique). When context switching on a control instruction, the instruction is still fed into the pipeline (in contrast to a load/store instruction). In this case context switching is always performed without the loss of a cycle.

The loss of one processor cycle in the EU when fetching the first of a sequence of load/store instructions can be avoided by a so-called *context switch buffer* (CSB) whenever the sequence is executed the second time. The CSB is a hardware buffer, collecting the addresses of load/store instructions that have caused context switches. Before fetching an instruction from the I-cache, the IF stage checks whether the address can be found in the context switch buffer. In that case the thread is switched immediately and an instruction of another thread is fetched instead of the load/store instruction. No bubble occurs.

Additionally, two further context-switching techniques are applied. First, the availability of register values is tested with a scoreboard in the instruction decode stages of both pipelines. If an operand value is unavailable, pipeline interlocking is avoided by a context switch (switch-on-use technique). Due to full forwarding techniques implemented in the execution pipeline, the switch-on-use technique triggers a context switch only when a missing operand is not loaded fast enough from memory.

Second, a so-called `sync` instruction performs a context switch only when acknowledges of load/store instructions are pending (conditional-switch technique). The `sync` instruction is provided to ease the implementation of different consistency models.

2.3.6. PL/PS Machine

The PL/PS Machine (Preload and Poststore) [58] is most similar to the Rhamma processor. It also decouples memory accesses from thread execution by providing separate units. This decoupling eliminates thread stalls due to memory accesses and makes thread switches due to cache misses unnecessary. Threads are created when all data is preloaded into the register set holding the thread's context, and the results from an execution thread are poststored. Threads are non-blocking and each thread is enabled when the required inputs are available (i.e. data driven at a coarse grain). The separate load/store/sync processor performs preloads and schedules ready threads on the pipeline. The pipeline processor executes the threads which will require no memory accesses. On completion the results from the thread are poststored by the load/store/sync processor.

2.3.7. The DanSoft nanothreading approach

The *nanothreading* approach uses multithreading but spares the hardware complexity of providing multiple register sets. The DanSoft nanothreading [59] proposed for the DanSoft processor dismisses full multithreading for a nanothread that executes in the same register set as the main thread. The DanSoft nanothread requires only a 9-bit PC, some simple control logic and it resides in the same page as the main thread. Whenever the processor stalls on the main thread, it automatically begins fetching instructions from the nanothread. Only one register set is available, so the two threads must share the register set. Typically the nanothread will focus on a simple task, such as prefetching data into a buffer, which can be done asynchronously to the main thread.

In the DanSoft processor, nanothreading is used to implement a new branch strategy that fetches both sides of a branch. A static branch prediction scheme is used, where branch instructions include 3 bits to direct the instruction stream. The bits specify eight levels of branch direction. For the middle four cases, denoting low confidence on the branch prediction, the processor fetches from both the branch target and the fall-through path. If the branch is mispredicted in the main thread, the back-up path executed in the nanothread generates a misprediction penalty of only 1 to 2 cycles.

The DanSoft processor proposal is a dual-processor CMP, each processor featuring a VLIW instruction set and the

nanothreading technique. Each processor contains an integer processor, but the two processor cores share a floating-point unit as well as the system interface.

However, the nanothread technique might also be used to fill the instruction issue slots of a wide superscalar approach as in SMT.

2.3.8. Microthreading

The *microthreading* technique of Bolychevsky *et al.* [60] is similar to nanothreading. All threads share the same register set and the same run-time stack. However, the number of threads is not restricted to two. When a context switch arises, the program counter is stored in a continuation queue. The PC represents the minimum possible context information for a given thread. Microthreading is proposed for a modified RISC processor.

Both techniques—nanothreading as well as microthreading—are proposed in the context of a blocked multithreading technique, but might also be used to fill the instruction issue slots of a wide superscalar approach as in SMT.

The drawback to nanothreading and microthreading is that the compiler has to schedule registers for all threads that may be active simultaneously, because all threads execute in the same register set.

The solution to this problem has been recently described by Jesshope and Luo in [61, 62]. These papers describe the dynamic allocation of registers using vector instruction sets and also the ease with which the architecture can be developed as a CMP.

2.3.9. Sun's MAJC

Sun proposed in 1999 its MAJC-5200 [63] that can be classified as a dual-processor chip with block-multithreaded processors. Special Java-directed instructions are provided motivating the acronym MAJC for Micro Architecture for Java Computing. Instruction, data, thread and process-level parallelism is exploited in the basic MAJC architecture by supporting explicit multithreading (so-called vertical multithreading), implicit multithreading (called speculative multithreading) and chip multiprocessors. Instruction-level parallelism is utilized by VLIW packets containing from one to four instructions and data-level parallelism through SIMD (single-instruction, multiple-data) instructions in particular for multimedia applications. Thread-level parallelism is utilized through compiler-supported explicit multithreading. The architecture allows one to combine several multithreaded processors on a chip to harness process-level parallelism. Single-threaded program execution may be accelerated by speculative multithreading with so-called microthreads that are dependent on non-speculative threads. Virtual channels communicate shared register values between the non-speculative thread and a speculative microthread respectively between different microthreads with produced-consumer synchronization [64]. In the case of misspeculation the speculative microthread is discarded.

2.3.10. Multithreaded PowerPC processor

IBM developed a multithreaded PowerPC processor, which is used in the IBM iSeries and pSeries commercial servers. The processor—originally code-named SStar—is called RS64 IV and it became available for purchase in the fourth quarter of 2000. Because of its optimization for commercial server workload (i.e. on-line transaction processing, enterprise resource planning, Web serving and collaborative groupware) with typically large and function-rich applications, a two-threaded block-interleaving approach with a switch-on-cache-miss model was chosen. A thread-switch buffer, which holds up to eight instructions from the background thread, reduces the cost of pipeline reload after a thread switch. These instructions may be introduced into the pipeline immediately after a thread switch. A significant throughput increase by multithreading while adding less than 5% to the chip area was reported in [65].

2.4. Simultaneous multithreading

Interleaved multithreading and blocked multithreading are multithreading techniques which are most efficient when applied to scalar RISC or VLIW processors. Combining multithreading with the superscalar technique naturally leads to a technique where several hardware contexts are active simultaneously, competing each cycle for all available resources. This technique, called *simultaneous multithreading* (SMT), inherits from superscalars the ability to issue multiple instructions each cycle; and like multithreaded processors it contains hardware resources for multiple contexts. The result is a processor that can issue multiple instructions from multiple threads *each cycle*.

The SMT approach combines a wide superscalar instruction issue with the multithreading approach by providing several register sets on the processor and issuing instructions from threads simultaneously. Therefore, the issue slots of a wide-issue processor can be filled by operations of several threads. Latencies occurring in the execution of single threads are bridged by issuing operations of the remaining threads loaded on the processor. In principle, the full issue bandwidth can be utilized. The SMT fetch unit can take advantage of the interthread competition for instruction bandwidth in two ways. First, it can partition this bandwidth among the threads and fetch from *several threads* each cycle. In this way, it increases the probability of fetching only non-speculative instructions. Second, the fetch unit can be selective about *which threads* it fetches. For example, it may fetch those that will provide the most immediate performance benefit (see the ICOUNT feedback technique in SMT at the University of Washington below). SMT processors can be organized in two ways.

- *Resource sharing*. Instructions of different threads share all resources like the fetch buffer, the physical registers for renaming registers of different register sets, the instruction window and the reorder buffer. Thus SMT adds minimal hardware complexity to conventional superscalars; hardware designers can focus on building a fast single-threaded superscalar and add

multithread capability on top. The complexity added to superscalars by multithreading includes the thread tag for each internal instruction representation, multiple register sets and the abilities of the fetch and the retire units to fetch/retire instructions of different threads.

- *Resource replication*. The second organizational form replicates all internal buffers of a superscalar such that each buffer is bound to a specific thread. Instruction fetch, decode, rename and retire units may be multiplexed between the threads or be duplicated themselves. The issue unit is able to issue instructions of different instruction windows simultaneously to the execution units. This form of organization adds more changes to the organization of superscalar processors but leads to a natural partitioning of the instruction window, and simplifies the issue and retire stages.

Thread-level parallelism can come from either multithreaded, parallel programs or from multiple, independent programs in a multiprogramming workload, while ILP is utilized from the individual threads. Because a SMT processor simultaneously exploits coarse- and fine-grained parallelism, it uses its resources more efficiently and thus achieves better throughput and speedup than single-threaded superscalar processors for multithreaded (or multiprogramming) workloads. The trade-off is a slightly more complex hardware organization.

The main drawback to SMT may be that it complicates the issue stage, which is always central to the multiple threads. A functional partitioning as demanded for processors of the 10^9-transistor era cannot be reached easily.

Projects using different configurations of simultaneous multithreading are discussed below.

2.4.1. *MARS-M*

The MARS-M multithreaded computer system [66] was developed and manufactured within the Russian Next-Generation Computer Systems program during 1982–1988. The MARS-M was the first system where the technique of simultaneous multithreading was implemented in a real design. The system has a decoupled multiprocessor architecture with execution, address, control, memory and peripheral multithreaded subsystems working in parallel and communicating via multiple register FIFO queues. The execution and address subsystems are multiple-unit VLIW processors with simultaneous multithreading. Within each of these two subsystems up to four threads can run in parallel on their hardware contexts allocated by the control subsystem, while sharing the subsystem's set of pipelined functional units and resolving resource conflicts on a cycle-by-cycle basis. The control processor uses interleaved multithreading with a zero-overhead context switch upon issuing a memory load operation. In total, up to 12 threads can run simultaneously within MARS-M with a peak instruction issue rate of 26 instructions per cycle. Medium scale integrated (MSI) and large scale integrated (LSI) ECL elements with a minimum delay of 2.5 ns are used to implement processor logic. There are 739 boards, each of which can contain up to 100 ICs mounted on both sides.

2.4.2. *Matsushita Media Research Laboratory processor*

The multithreaded processor of the Media Research Laboratory of Matsushita Electric Ind. (Japan) was another pioneering approach to SMT [67]. Instructions of different threads are issued simultaneously to multiple execution units. Simulation results on a parallel ray-tracing application showed that, using eight threads, a speedup of 3.22 in the case of one load/store unit and 5.79 in the case of two load/store units can be achieved over a conventional RISC processor. However, caches or translation lookaside buffers (TLBs) are not simulated, nor is a branch prediction mechanism.

2.4.3. *Multistreamed Superscalar*

Serrano *et al.* [68, 69] at the University of Santa Barbara, CA, extended the interleaved multithreading (then called multistreaming) technique to a general purpose superscalar processor architecture and presented an analytical model of multithreaded superscalar performance, backed up by simulation.

2.4.4. *Irvine Multithreaded Superscalar*

This multithreaded superscalar processor approach, developed at the University of California at Irvine, combines out-of-order execution within an instruction stream with the simultaneous execution of instructions of different instruction streams [70, 71]. A particular superscalar processor called the Superscalar Digital Signal Processor (SDSP) is enhanced to run multiple threads. The enhancements are directed by the goal of minimal modification to the superscalar base processor. Therefore, most resources on the chip are shared by the threads, as for instance the register file, reorder buffer, instruction window, store buffer and renaming hardware. Based on simulations a performance gain of 20–55% due to multithreading was achieved across a range of benchmarks.

A multithreaded superscalar processor model was evaluated in [72] using several video decode, picture processing and signal filter programs as workloads. The programs were parallelized at the source code level by partitioning the main loop and distributing the loop iterations to several threads. The rather disappointing speedup that was reported for multithreading results from algorithmic restrictions and from the already high IPC in the single-threaded model. The latter is only possible because multimedia instructions were not used. Otherwise a large part of the IPC in the single-threaded model would be hidden by the SIMD parallelism within the multimedia instructions.

2.4.5. *SMT at the University of Washington*

The SMT processor architecture [73], proposed in 1995 at the University of Washington, Seattle, WA, coined the term *simultaneous multithreading*. Simulations were conducted to evaluate processor configurations of an up to 8-threaded and 8-issue superscalar based on an enhanced Alpha 21164 processor architecture. This maximum configuration showed a throughput of 6.64 IPC due to multithreading using

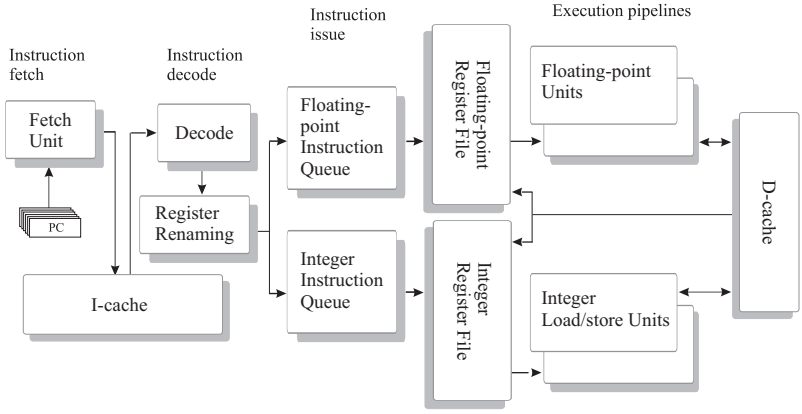

FIGURE 8. SMT processor architecture.

the SPEC92 benchmark suite and assuming a processor with 32 execution units (among them multiple load/store units).

The next approach was based on a hypothetical out-of-order instruction issue superscalar microprocessor that resembles the MIPS R10000 and HP PA-8000 [36, 74]. This approach evaluated more realistic processor configurations, and presented implementation issues and solutions to register file access and instruction scheduling for a minimal change to superscalar processor organization.

In the simulations of the latter architectural model (Figure 8) eight threads and an eight-issue superscalar organization are assumed. Eight instructions are decoded, renamed and fed to either the integer or floating-point instruction window. Unified buffers are used in contrast to thread-specific queues in the Karlsruhe Multithreaded Superscalar approach (see below). When operands become available, up to eight instructions are issued out of order per cycle, executed and retired. Each thread can address 32 architectural integer (and floating-point) registers. These registers are renamed to a large physical register file of 356 physical registers. The larger SMT register file requires a longer access time. To avoid increasing the processor cycle time, the SMT pipeline is extended by two stages to allow two-cycle register reads and two-cycle writes. Renamed instructions are placed into one of two instruction windows. The 32-entry integer instruction window handles integer and all load/store instructions, while the 32-entry floating-point instruction window handles floating-point instructions. Three floating-point and six integer units are assumed. All execution units are fully pipelined and four of the integer units also execute load/store instructions. The I- and D-caches are multiported and multibanked, but are common to all threads.

The multithreaded workload consists of a program mix of SPEC92 benchmark programs that are executed simultaneously as different threads. The simulations evaluated different fetch and instruction issue strategies.

An RR.2.8 fetching scheme to access the multiported I-cache, i.e. in each cycle two times eight instructions are fetched in a round-robin policy from two different threads, was superior to other schemes like RR.1.8, RR.4.2 and RR.2.4 with less fetching capacity. As a fetch policy, the ICOUNT feedback technique, which gives highest fetch priority to the threads with the fewest instructions in the decode, renaming and queue pipeline stages, proved superior to the BRCOUNT scheme which gives highest priority to those threads that are least likely to be on a wrong path, and the MISSCOUNT scheme which gives priority to the threads that have the fewest outstanding D-cache misses. The IQPOSN policy that gives lowest priority to the oldest instructions by penalizing those threads with instructions closest to the head of either the integer or the floating-point queue is nearly as good as ICOUNT, and better than BRCOUNT and MISSCOUNT, which are all better than round-robin fetching. The ICOUNT.2.8 fetching strategy reached an IPC of about 5.4 (the RR.2.8 only reached about 4.2). Most interesting is the fact that neither mispredicted branches nor blocking due to cache misses, but a mix of both and perhaps some other effects proved to be the best fetching strategy.

In a single-threaded processor, choosing instructions for issue that are least likely to be on a wrong path is always achieved by selecting the oldest instructions, those deepest into the instruction window. For the SMT processor several different issue strategies have been evaluated, such as oldest instructions first, speculative instructions last and branches first. Issue bandwidth is not a bottleneck and all strategies seem to perform equally well, so the simplest mechanism is to be preferred. Also doubling the size of instruction windows (but not the number of searchable instructions for issue) has no significant effect on the IPC. Even an infinite number of execution units increases throughput by only 0.5%.

Further research looked at compiler techniques for SMT [75] and at extracting threads of a single program designed for multithreaded execution on an SMT [76]. The *threaded multipath execution* model, which exploits existing hardware on a SMT processor to execute simultaneously alternate paths of a conditional branch in a thread, is presented in [77]. Threaded Multiple Path Execution employs eager execution of branches in an SMT processor model. It extends the SMT processor by introducing additional hardware to test for unused processor resources (unused hardware threads), a confidence estimator, mechanisms for

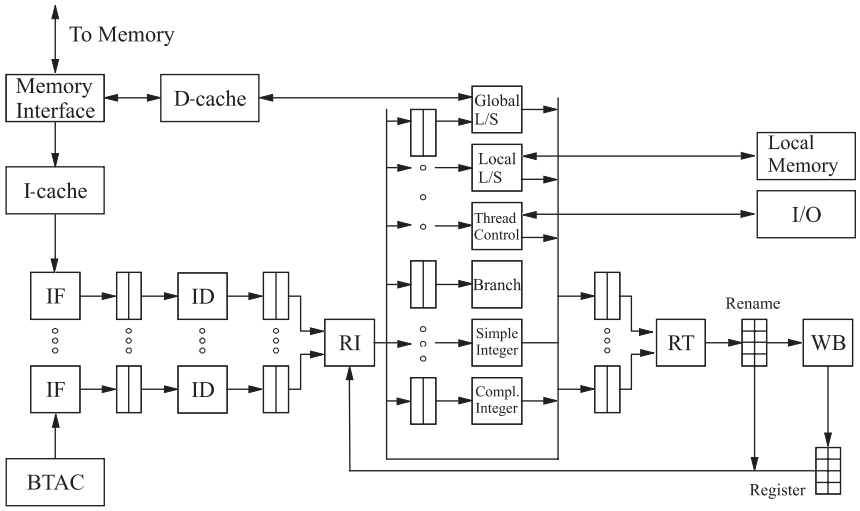

FIGURE 9. The SMT Multimedia processor (IF, fetch unit; ID, decode unit; RI, rename/issue unit; RT, retirement unit; WB, write-back unit; BTAC, branch target address cache).

fast starting and finishing threads, priorities attached to the threads, support for speculatively executed memory access operations and an additional bus for distributing the contents of the register mapping table (Mapping Synchronization Bus (MSB)). If the hardware detects that a number of processor threads are not processing useful instructions, the confidence estimator is used to decide whether only one continuation of a conditional branch should be followed (high confidence) or both continuations should be followed simultaneously (low confidence). The MSB is used to provide a thread that starts execution of one continuation speculatively with the valid register map. Such register mappings between different register sets incur an overhead of 4–8 cycles. Such a speculative execution increases single program performance by 14–23%, depending on the misprediction penalty, for programs with a high branch misprediction rate. Instruction recycling has been explored on a multi-path SMT processor proposed in [78]. Recently, SMT has been evaluated with database workloads [79] achieving roughly a three-fold increase in instruction throughput with an eight-threaded SMT over a single-threaded superscalar with similar resources.

Another recent paper of Seng *et al.* [80] investigates use of simultaneous multithreading for reduction of power consumption. Harnessing the extra parallelism provided by multiple threads allows the processor to rely much less on speculation. They showed that a SMT processor utilizes up to 22% less energy per instruction than a single-threaded architecture.

2.4.6. *Karlsruhe Multithreaded Superscalar processor*

While the SMT processor at the University of Washington surveys enhancements of the Alpha 21164 processor, the *Multithreaded Superscalar* processor approach from the University of Karlsruhe [81, 82] is based on an extended PowerPC 604 processor with thread-local instruction, issue and reorder buffers (*resource replication* model instead of the *resource sharing* model used in the SMT at the University of Washington). Both approaches, however, are similar in their instruction-issuing policy.

Using an instruction mix with 20% load and store instructions, the performance results show, for an eight-issue processor with four to eight threads, that two instruction fetch units, two decode units, four integer units, 16 rename registers, four register ports and completion queues with 12 slots are sufficient. The single load/store unit proves the principal bottleneck because it cannot be easily duplicated. The multithreaded superscalar processor (eight-threaded, eight-issue) is able to hide completely latencies caused by 4–2–2–2 burst cache refills (4–2–2–2 assumes that four times 64-bit portions are transmitted over the memory bus, the first portion reaches the processor 4 cycles after the cache miss indication, the next portion 2 cycles later, etc.). It reaches the maximum throughput of 4.2 IPC that is possible with a single load/store unit.

2.4.7. *SMT Multimedia processor*

Subsequent SMT research at the University of Karlsruhe has explored microarchitecture models for a SMT processor with multimedia enhancements [83, 84]. The SMT Multimedia processor (see Figure 9) features single or multiple fetch (IF) and decode (ID) units, a single rename/issue (RI) unit, multiple, decoupled reservation stations, multiple execution units, in particular up to four combined integer/multimedia units, a complex integer/multimedia unit, a branch unit, separate local and global load/store units, a single retirement (RT) and write-back (WB) unit, rename registers, a BTAC and separate I- and D-caches that are shared by all active threads. Thread-specific instruction buffers (between IF and ID), issue buffers (between ID and RI) and reorder buffers (in front of RT) are employed. Each thread executes in a separate architectural register set. However, there is no fixed allocation of threads to (execution) units. The pipeline performs an in-order

instruction fetch, decode, rename/issue to reservation stations, out-of-order dispatch from the reservation stations to the execution units, out-of-order execution and an in-order retirement and write-back.

The rename/issue stage simultaneously selects instructions from all issue buffers up to its maximum issue bandwidth (SMT feature). The integer units are enhanced by multimedia processing capabilities (multimedia unit feature). A thread control unit performs thread start, stop, synchronization and I/O operations. A local RAM memory is included which is accessed by the local load/store unit.

The simulations showed that a single-threaded, eight-issue maximum processor (assuming an abundance of resources) reaches an IPC count of only 1.60, while an eight-threaded, eight-issue processor reaches an IPC of 6.07. A more realistic processor model reaches an IPC of 1.21 in the single-threaded eight-issue and 3.11 in the eight-threaded, eight-issue model. Increasing the issue bandwidth from 4 to 8 yields only a marginal gain (except for the four- to eight-threaded maximum processor). Increasing the number of threads from single to two- or four-threaded yields a high gain for the two- to eight-issue model and a significant gain for the eight-threaded model. The steepest performance increases arise for the four-issue model from the single-threaded (IPC of 1.21) to the two-threaded (IPC of 2.07) and to the four-threaded (IPC of 2.97) cases. A two-threaded, four-issue and a four-threaded, four-issue processor configuration (as realistic next-generation processors) were suggested in [83].

The most surprising finding was that smaller reservation stations for the thread unit and the global and the local load/store units, as well as smaller reorder buffers, increased the IPC value for the multithreaded models. Intuition suggests a better performance with larger buffers. However, large reservation stations (and large reorder buffers) draw too many highly speculative instructions into the pipeline. Smaller buffers limit the speculation depth of fetched instructions and lead to the fact that only non-speculative instructions or instructions with low speculation depth are fetched, decoded, issued and executed in a SMT processor. An abundance of speculative instructions hurts the performance of a SMT processor. Another reason for the negative effect of large reorder buffers and of large reservation stations for the load/store and thread control units lies in the fact that these instructions have a long latency and typically have two to three integer instructions as consumers. The effect is that the consuming integer instructions eat up space in the integer reservation station, thus blocking instructions from other threads from entering it. This multiplication effect is made even worse by a non-speculative execution of store and thread control instructions.

Subsequent research [85] investigated a cost/benefit analysis of various SMT multimedia processor models. Transistor count and chip space estimations (applying the tool described in [86]) showed that the additional hardware cost of an eight-threaded SMT processor over a single-threaded processor is a negligible 5% transistor increase, but a 36% chip space increase for a 300M transistor chip assumed for the years 2006 to 2009; it requires a 78% increase of the transistor amount for the processor models with realistic memory hierarchy and 21% more for the contemporary scaled processor models. The chip space increase of the eight-threaded, eight-issue over the single-threaded, eight-issue model with realistic memory hierarchy is about 207%, in the case of the contemporary scaled processor models it is 63%. Even more favorable is the comparison of the single-threaded, eight-issue models with the four-threaded, eight-issue SMT model. Maximum processor models require a 2% increase in transistor count and a 9% increase in chip space, but they yield a threefold speedup; the models with realistic memory hierarchy require a 31% increase in transistor count and a 53% increase in chip space, but they yield a nearly twofold speedup; and the contemporary scaled models require a 9% increase in transistor count and a 27% increase in chip space, resulting in a 1.5-fold speedup.

Similar results were reached by Burns and Gaudiot [87] who estimated the layout area for SMT. They identified which layout blocks are affected by SMT, determined the scaling of chip space requirements using an O-calculus and compared SMT versus single-threaded processor space requirements by scaling a R10000-based layout to 0.18 μm technology.

2.4.8. SMV processor

The Simultaneous Multithreaded Vector (SMV) architecture [88], designed at the Polytechnic University of Catalunya (Barcelona, Spain), combines simultaneous multithreaded execution and out-of-order execution with an integrated vector unit and vector instructions.

Figure 10 depicts the SMV architecture. The fetch engine selects one of eight threads and fetches four instructions on its behalf. The decoder renames the instructions, using a per-thread rename table, and then sends all instructions to several common execution queues, each instruction to one queue. Inside the queues, the instructions of different threads are indistinguishable and no thread information is kept except in the reorder buffer and memory queue. Register names preserve all dependences. Independent threads use independent rename tables, which prevents false dependences and conflicts from occurring. The vector unit has 128 vector registers, each holding 128 64-bit registers, and it has four general-purpose independent execution units. The number of registers is the product of the number of threads and the number of physical registers required to sustain good performance on each thread.

2.4.9. Alpha 21464

Compaq unveiled its Alpha EV8 21464 proposal in 1999 [89], a four-threaded, eight-issue SMT processor, which closely resembles the SMT processor proposed in [76]. The processor proposal features out-of-order execution, a large on-chip L2 cache, a direct RAMBUS interface and an on-chip router for system interconnect

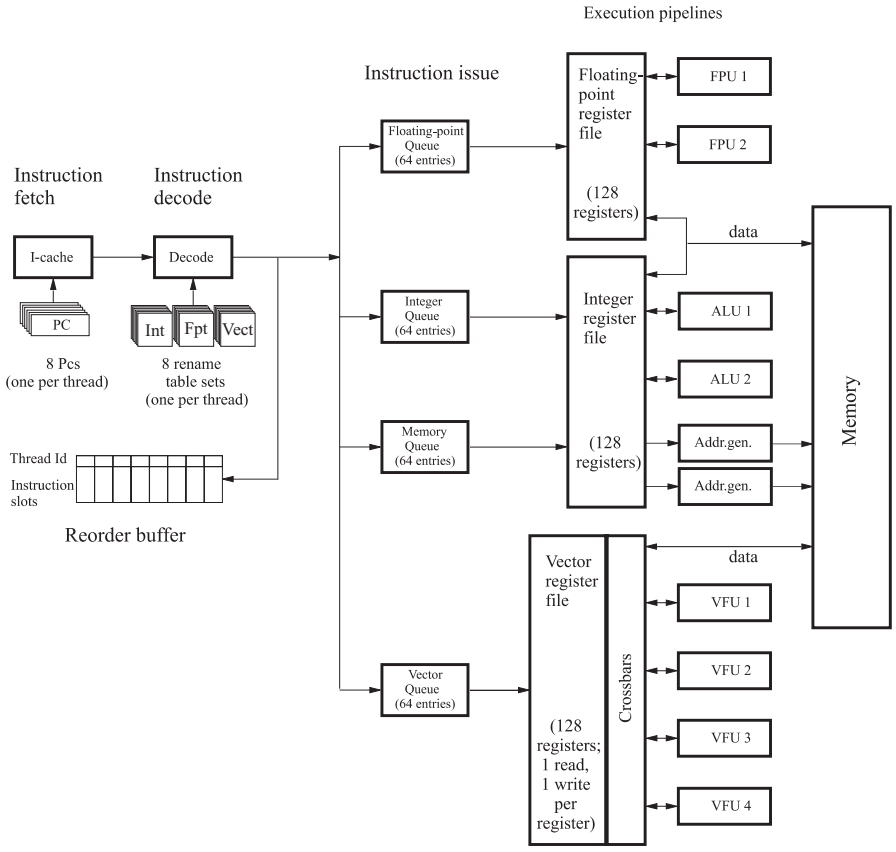

FIGURE 10. SMV architecture (FPU, floating-point unit; ALU, arithmetic/logic unit; VFU, vector functional unit).

of a directory based, cache-coherent non-uniform memory access (NUMA) multiprocessor. This 250 million transistor chip was planned for year 2003. In the meantime, the project has been abandoned, as Compaq sold the Alpha processor technology to Intel.

2.4.10. Blue Gene

Simultaneous multithreading is mentioned as a processor technique for the building block of the IBM Blue Gene system—a five-year effort to build a petaflops supercomputer started in December 1999 [90].

3. FURTHER APPROACHES

3.1. Implicit multithreading

Another set of solutions is to apply an even higher degree of speculation in combination with a functional partitioning of the processor. Here thread-level parallelism is utilized, typically in combination with thread-level speculation. A thread in such architectures refers to any contiguous region of the static or dynamic instruction sequence. Examples of such architectural approaches are:

- the multiscalar;
- the trace processor;
- the single-program speculative multithreading architecture;
- the superthreaded architecture;
- the dynamic multithreading processor;
- the speculative multithreaded processor;
- the MEM-slicing algorithm in the context of the Atlas CMP; and
- the technique of helper threads as, for example, in simultaneous subordinate microthreading.

Multiscalar processors [16, 17, 18, 19] divide a single-threaded program into a collection of tasks (in the following identified with threads) that are distributed to a number of ring-connected parallel processing units under the control of a single hardware sequencer. Each of the processing units fetches and executes instructions belonging to its assigned task. A static program is represented as a control flow graph (CFG), where basic blocks are nodes and arcs represent the flow of control from one basic block to another. Dynamic program execution can be viewed as walking through the CFG, generating a dynamic sequence of basic blocks which have to be executed for a particular run of the program. The multiscalar paradigm supports two forms of speculation: control speculation, which is used by a hardware thread sequencer, and data dependence speculation, speculating that a load does not depend on instructions executing in predecessor threads. If a control speculation turns out to be incorrect, the processor discards the speculative thread and all its following threads. For data dependence speculation a thread loads data from memory with the expectation that the predecessor threads will not store a value to the same memory location [91]. An address resolution buffer (ARB)

is provided to hold speculative memory operations and to detect violations of memory dependences. The ARB checks that the speculation was correct, squashing instructions if it was not. The speculative versioning cache (SVC) [92] extends the ARB concept by using distributed caches instead of a centralized buffer. SVC eliminates the latency and bandwidth problems of the ARB.

Trace processors [20, 21, 22] partition a processor into multiple distinct cores—similar to multiscalar—and break the program into traces. Traces are collected by a trace cache [1, 93] which is a special instruction cache that captures dynamic instruction sequences. One core of a trace processor executes the current trace while the other cores execute future traces speculatively.

The *superthreaded architecture* [24, 25] applies a thread pipelining execution model that allows threads with data and control dependences to be executed in parallel. As in the multiscalar approach, a compiler statically partitions the CFG of a program into threads to be executed on the thread processing units of the superthreaded architecture. The thread processing units are connected by a unidirectional bus to exchange data that arise from loop-carried dependences, similar to the multiscalar proposal. The superthreading technique is designed for the exploitation of loop-level parallelism where each interaction is executed within another thread. The superthreaded architecture, like the multiscalar approach, is closely related to a CMP approach. However, both the superthreaded and the multiscalar approaches use more closely coupled processing elements and are designed to increase single-thread performance using a compiler-supported task partitioning.

The multiscalar as well as the *single-program speculative multithreading architecture* [23] support speculation on data dependences that are unknown at compile time. In contrast, the superthreaded processor does not speculate on data dependences. The processor performs run-time data dependence checking for load operations and, if a load operation is detected to be data dependent on a store operation of a predecessor thread, it waits for the stored data from the predecessor thread. This technique avoids squashing caused by data dependence violations as in the multiscalar and it reduces the hardware complexity of detecting memory dependence violations.

The *dynamic multithreading processor* [26] has hardware to create threads at procedure and loop boundaries and executes the threads speculatively on a simultaneous multithreaded pipeline. A thread spawns a new thread when it encounters a procedure call or a backward branch. The latter is speculatively treated as the end of a loop. Thread-level dataflow and data value prediction are used. Loads are issued speculatively to memory assuming that there are no conflicting stores from prior threads. Data misprediction recovery is organized using large trace buffers that hold all the speculative instructions and results.

The *speculative multithreaded processor* [27, 94] also uses hardware to partition a program into threads that execute successive iterations of the same loop. A loop detection scheme to dynamically detect loops without compiler or user intervention with the aim of obtaining multiple threads from a sequential program is presented in [94].

One essential question for implicit multithreaded architectures is how to partition the program and when to spawn a speculative thread. Codrescu and Wills [95] investigate different dynamic partitioning schemes, in particular thread-generation by dynamically parallelizing loop iterations, procedure calls or using fixed instruction length blocks. Their results suggest memory instructions as the best instruction type to use as slice instructions to begin and break speculative threads. A new, flexible algorithm—called the *MEM-slicing algorithm*—is proposed that generates a thread starting from a slice instruction up to a maximum thread length. All approaches are evaluated in the context of the Atlas CMP.

Another step in the direction of multithreading is simultaneous subordinate microthreading (SSMT) [30] which is a modification of superscalars to run threads at microprogram level concurrently. A new microthread is spawned either event-driven by hardware or by an explicit spawn instruction. The subordinate microthread could be used, for example, to improve branch prediction of the primary thread or to prefetch data.

Similar approaches apply multithreading for event-handling of internal events by rapidly spawning helper threads that execute simultaneously to the main thread [96, 97]. Speculative slices may be executed as helper threads to prefetch data into the cache and to generate branch predictions without affecting the execution state of the processors [98]. The speculative precomputation technique uses idle thread contexts in a multithreaded architecture based on a SMT Itanium model to trigger future cache miss events by pre-computing future memory accesses and prefetching these data [99]. Similarly, software-controlled pre-execution for speculative address generation and prefetching is proposed by Luk [100] using idle threads on an Alpha 21464-like SMT processor.

We call all these architectures *implicit multithreaded architectures* because they concurrently execute several threads from a single sequential program. These architectures share with SMT the ability to execute several (in this case implicitly generated) threads concurrently. However, the microarchitectures of these approaches are often organized differently from SMT. In SMT instructions of different threads are issued simultaneously from a single central instruction window. The multiscalar, trace and superthreaded approaches feature a decentralized ILP execution model with control-dependence-based decentralization [101]. A functional partitioning within the architecture splits execution into several closely coupled execution units where each execution unit is responsible for its own thread execution. Such a decentralized ILP execution model leads to a natural functional partitioning within the processors. By its decentralized organization it is potentially better suited to the requirement that future processors have short on-chip wires. The microarchitecture of such approaches is therefore often more closely related to CMPs than to

SMT. Implicit multithreaded architectures in the forms of speculative multithreading and helper threads are currently hot research topics.

3.2. Scheduled dataflow

In contrast to most multithreading approaches that search for better performance at the expense of an increased hardware complexity, the scheduled dataflow (SDF) of Kavi *et al.* [102] defines a simple, yet powerful multithreaded execution paradigm that is based on dataflow concepts.

A root of multithreaded execution is the coarse-grained dataflow execution model that relies on non-blocking threads generated from single-assignment dataflow programs. Threads start execution as soon as all operands are available. Such threads may be generated with the aim of decoupling memory accesses from execute instructions.

Kavi *et al.* [102] present a decoupled scheduled dataflow architecture as a redesign of the PL/PS Machine [58]. A (dataflow) program is compiler-partitioned into functional execution and memory-access threads. All memory accesses are decoupled from the (functional execution) threads. Data is pre-loaded into the thread's context (registers) and all results are post-stored after the completion of the thread's execution. The decoupling of memory accesses from thread execution requires a separate unit to perform the necessary pre-loads and post-stores and to control the allocation of hardware thread contexts to enabled threads.

The analytical analysis showed that the SDF processor can outperform other classical dataflow architectures, hybrid models and decoupled multithreaded architectures (e.g. the Rhamma processor).

3.3. Dual path branch execution models

A number of research projects survey eager execution—dual path execution of branches. They extend either superscalar or SMT processors. All need some kind of architecture that is able to pursue two threads in parallel.

The nanothreaded DanSoft processor (see Section 2.3.7) implements a multiple path execution model using confidence information from a static branch prediction mechanism. The information is stored in some additional branch opcode bits. The hardware dynamically decides whether dual path execution should be applied using the information from these opcode bits.

The *threaded multipath execution* model exploits Idle thread slots in a SMT processor to execute simultaneously alternate paths of a conditional branch (see Section 2.4.5).

The Simultaneous Speculation Scheduling (*S*3) [103, 104, 105, 106] is a combined compiler and architecture technique to control multiple path execution. The *S*3 technique can be applied to dual path branch speculation in the case of unpredictable branches and to multiple path speculative execution of loop iterations. Loop-carried memory dependences that cannot be disproven by the compiler are handled by data dependence speculation. Architectural requirements are the ability to pursue more than one thread concurrently and three new instructions (fork, sync, wait).

Other eager execution techniques are Selective Dual Path Execution [107], Limited Dual Path Execution [108], PolyPath architecture [29] and Threaded Multiple Path Execution [77]. The Disjoint Eager Execution technique [109] assigns resources to branch paths with the highest cumulative execution probabilities.

Closely related to SMT are architectural proposals that only slightly enhance a superscalar processor by the ability to pursue two or more threads only for a short time. In principle, predication is the first step in this direction. A fully predicated instruction set like, for example, IA-64 and many signal processors permit the system to pursue two branch paths simultaneously by defining two virtual threads. However, no additional instruction pointer or register window is applied. Predicated instructions are fetched, decoded and dispatched to the instruction window(s). In the most common form of predication, predicated instructions with unresolved predicates cannot be executed, because the predicate is treated as additional data register input and the instruction is only issued to the reservation station/execution unit if the predicate is computed and true.

An enhanced form of predication is able to issue and execute a predicated instruction even if the predicate is not yet solved. Consequently, all instructions with a false predicate are discarded in the retirement stage, all instructions with a true predicate are committed. The latter form of predication allows one to view the predicate as a form of tagging of the instruction stream as is necessary for multithreaded processors to distinguish instructions of different threads in the pipeline. However, only a single instruction pointer is still used, all instructions are still in one instruction stream—the threads are still virtual.

A further step is dynamic predication [28] as applied for the Polypath architecture [29] that is an enhanced superscalar to handle multiple threads internally. The PolyPath architecture enhances a superscalar processor by a limited multi-path execution feature to employ eager execution of branches. It does not support the execution of independent threads in hardware, but it feeds instructions from both possible continuations of a conditional branch into the superscalar pipeline. This speculation on the outcome of the conditional branches is completely implemented in hardware. The instructions passing the pipeline are extended by a context tag. In our opinion, the PolyPath processor is in fact a multithreaded architecture since all instructions with the same tag can be considered to belong to the same virtual thread. However, the architecture cannot benefit from coarse-grained parallelism. Extending the instructions with a context tag has to be done in all processor resources (instruction window, store queues, etc.). Beside this tagging mechanism, the PolyPath architecture implements a Jacobsen, Rotenberg and Smith (JRS) confidence estimator [110]. If the confidence of the prediction of a branch is low, both possible continuations are speculatively executed; otherwise, a normal check-point mechanism is used to follow only the more likely outcome of the branch.

3.4. Multithreading approaches for signal processing and real-time event handling

The rapid context-switching ability of the multithreading technique leads to approaches that apply multithreading in new areas, in particular embedded real-time systems and signal processing.

Recently, multithreading has been proposed in future high-end processors for event-handling of internal events by rapidly spawning helper threads that execute simultaneously to the main thread (see Section 3.1). However, the fast context-switching ability of multithreading has rarely been explored in the context of microcontrollers for the handling of external hardware events in embedded systems.

3.4.1. Multithreading technique for signal processors

Wittenburg *et al.* [111] looked at the application of the SMT technique to signal processors using combining instructions that are applied to registers of several register sets simultaneously instead of multimedia operations. Simulations with a Hough transformation as workload showed a speedup of up to six compared to a single-threaded processor without multimedia extensions.

3.4.2. The EVENTS mechanism

Metzner and Niehaus [112] propose the use of multithreaded processors for real-time event handling. Several block-multithreaded MSparc processors (see Section 2.3.4) are supervised by an processor-external thread scheduler, called EVENTS [112, 113], which triggers contest switches due to real-time scheduling techniques and assigns computation-intensive real-time threads to the different MSparc processors. The EVENTS mechanism is implemented as a field of field programmable gate-arrays (FPGAs).

In contrast to EVENTS the Komodo microcontroller applies real-time scheduling algorithms on an instruction-by-instruction basis deeply embedded in the multithreaded processor core.

3.4.3. Komodo microcontroller

The Komodo microcontroller [114, 115], is a multithreaded Java microcontroller aimed at embedded real-time systems with a hardware event handling mechanism that allows the handling of simultaneous overlapping events with hard real-time requirements. The main purpose for the use of multithreading within the Komodo microcontroller is not latency utilization, but extremely fast reaction on real-time events.

Real-time Java threads are used as *interrupt service threads* (ISTs)—a new hardware event handling mechanism that replaces the common *interrupt service routines* (ISRs). Occurrence of an event activates an assigned IST instead of an ISR. The Komodo microcontroller supports the concurrent execution of multiple ISTs, zero-cycle overhead context switching, and triggers ISTs by a switch-on-signal context switching strategy.

As shown in Figure 11, the four-stage pipelined processor core consists of an instruction-fetch unit, a decode unit, an operand fetch unit and an execution unit. Four stack register sets are provided on the processor chip. A Java bytecode instruction is decoded either to a single micro-op, a sequence of micro-ops or a trap routine is called. Each micro-op is propagated through the pipeline with its thread ID. Micro-ops from multiple threads can be simultaneously present in the different pipeline stages.

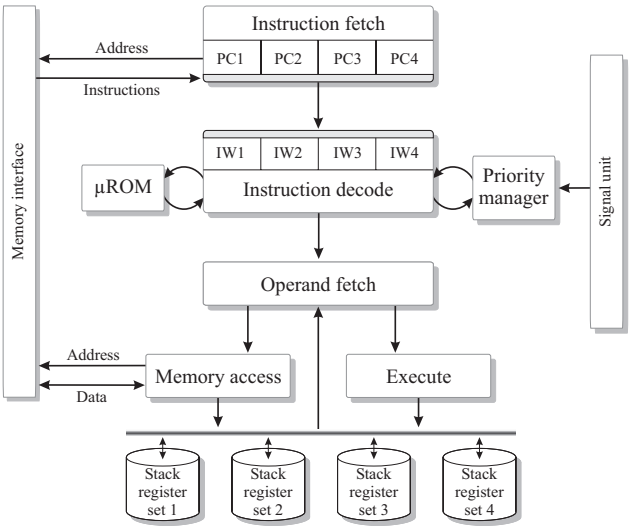

FIGURE 11. Block diagram of the Komodo microcontroller (IW, instruction window; PC, program counter).

The instruction fetch unit holds four program counters (PC) with dedicated status bits (e.g. thread active/suspended); each PC is assigned to a separate thread. Four byte portions are fetched over the memory interface and put in the according instruction window (IW). Several instructions may be contained in the fetch portion, because of the average Java bytecode length of 1.8 bytes. Instructions are fetched depending on the status bits and fill levels of the IWs.

The instruction decode unit contains the IWs, dedicated status bits (e.g. priority) and counters for the implementation of the proportional share scheme. A priority manager decides from which IW the next instruction will be decoded.

The real-time scheduling algorithms fixed priority preemptive (FPP), earliest deadline first (EDF), least laxity first (LLF) and guaranteed percentage (GP) scheduling are implemented in the priority manager for next instruction selection [116]. The priority manager applies one of the implemented scheduling schemes for IW selection. However, latencies may result from branches or memory accesses. To avoid pipeline stalls, instructions from threads of less priority can be fed into the pipeline. The decode unit predicts the latency after such an instruction and proceeds with instructions from other IWs.

External signals are delivered to the signal unit from the peripheral components of the microcontroller core as, for example, timer, counter or serial interface. The corresponding IST will be activated by the occurrence of such a signal. As soon as an IST activation ends its assigned

real-time thread is suspended and its status is stored. An external signal may activate the same thread again.

Further investigations within the Komodo project [117] cover real-time scheduling schemes on a multithreaded processor [116], real-time garbage collection on a multithreaded microcontroller [118] and a distributed real-time middleware called OSA+ [117].

4. CHIP MULTIPROCESSORS

4.1. Principal alternatives

Today the most common organizational principles for multiprocessors are the symmetric multiprocessor (SMP), the distributed shared memory multiprocessor (DSM) and the message-passing shared-nothing multiprocessor [31, 119]. The SMP and the DSM multiprocessors feature a common address space, which is implemented in the SMP as a single global memory where each memory word can be accessed in uniform access time by all processors (UMA, uniform memory access). In the DSM multiprocessor a common address space is maintained despite physically distributed memory modules. A processor in a DSM can access data in its local memory faster than in the remote memory (the memory module local to another processor). DSM multiprocessors are therefore NUMA systems. Shared-nothing multiprocessors feature physically distributed memory modules and no common address space. Therefore, communication can only be performed by passing messages between processors. Shared-nothing multiprocessors are highly scalable but harder to program than shared-memory multiprocessors. They are beyond the scope of our discussion of CMPs, which, by their tight physical coupling on a single chip, may also feature a very tight coupling of instruction streams, usually expressed by a common memory organization.

The principal organizational forms of multiprocessors do not regard cache organization. Commodity microprocessors, which are usually used today as building blocks for multiprocessors, contain on-chip caches, often coupled with off-chip secondary cache memories. Shared-memory multiprocessors maintain cache coherence by a cache coherence protocol which is a bus-snooping coherence protocol for SMPs or a directory-based coherence protocol for DSMs. SMPs consist of a moderate number of commodity microprocessors with cache memories coupled by a fast memory bus with the global memory. In the latest SMPs the memory bus is replaced by an address bus (necessary for bus-snooping) and a data crossbar switch for faster transfer of cache lines. SMPs are the starting point for CMPs.

In order to develop insight about the most appropriate memory hierarchy level for connecting the CPUs in a CMP, three alternatives were compared by Nayfeh *et al.* [120]: a shared-main-memory multiprocessor (i.e. the typical symmetric multiprocessor today), a shared-secondary-cache multiprocessor and a shared-primary-cache multiprocessor. They found that, when applications have a high or moderate degree of interprocessor communication, both shared-primary-cache and shared-secondary-cache architectures perform similarly and outperform the shared-main-memory architecture substantially. There are two reasons for this. First, the shared cache was assumed to be large enough to accommodate most of the working sets of independent threads running on different CPUs, so that the cache miss rate is low. Second, when there is interprocessor communication, it is handled very efficiently in the shared (primary or secondary) cache. Even for applications with little or no interprocessor communication, the performance of the shared-primary-cache architecture is still slightly better than the shared-main-memory architecture.

As examples of CMPs, we will describe the Hydra chip multiprocessor [121, 122], a research processor currently being designed at Stanford University in an effort to evaluate the shared-secondary-cache CMP, and the IBM POWER4 chip [123] as a commercial CMP. These are not the first projects on CMP design, though. In 1994, Texas Instruments introduced the TMS320C80 multimedia video processor (MVP) [124], a variant of the shared-primary-cache CMP which contained five processors on a single chip.

4.1.1. TI TMS320C8x multimedia video processors

The Texas Instruments TMS320C8x (or 'C8x) family of processors are CMPs suitable for system-level and embedded implementations [124]. Applications include image processing, 2-D and 3-D graphics, audio/video digital compression and playback, real-time encryption/decryption and digital telecommunications. The processor is dubbed MVP, the multimedia video processor. A single MVP replaces several system components by integrating multiple processors, memory control logic, I-cache and internal memory, an advanced DMA controller and video timing generation logic ('C80 only) onto a single chip. They provided an order of magnitude increase in computational power over existing digital signal processors (DSPs) and general-purpose processors in 1994.

Two types of processors are combined in the MVP architecture: a single RISC *master processor* (MP) and a number of VLIW DSP-like *parallel processors* (PPs) (Figure 12). Moreover, the chip contains a programmable DMA *transfer controller* (TC), a video controller (VC) and a boundary-scan test access port (TAP). All processors are interconnected by a crossbar with I-caches and data RAM and parameter RAM areas.

The 'C8x family consists of two members, the 'C80 [124], which features four PPs and the 'C82 [125] with only two on-chip PPs.

The MP functions as a 32-bit RISC master control processor, and is intended to handle interrupts and external requests and control the operation of the system as a whole. The MP has an ANSI/IEEE 754-1985 compliant floating-point unit. Although the MP ISA is classified as a RISC, it has the capability of issuing explicit parallel operations in a VLIW fashion. Up to three such operations can be issued: a multiply, an arithmetic/logical and a load/store operation.

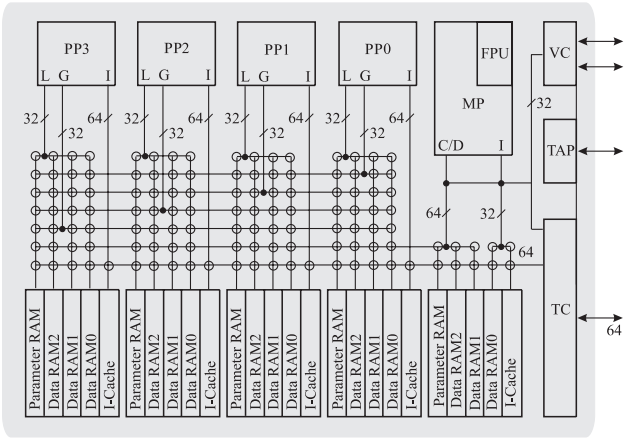

FIGURE 12. TI's multimedia video multiprocessor (PP, parallel processor; MP, master processor; FPU, floating-point unit; VC, video controller; TAP, boundary-scan test access port; TC, transfer controller).

The PPs are designed to perform 64-bit multiply-intensive, multiple pixel and bit-field manipulation operations on data in registers and internal RAM. The PPs are a collection of 32-bit fixed-point subprocessors connected by data path multiplexers. This allows a wide variety of operations to issue simultaneously. A local and global address unit, a three-input ALU and a multiplier unit are included along with a barrel rotator (an execution unit that is similar to a barrel shifter but performs rotation, as opposed to shifting, by an arbitrary number of bits in one instruction cycle), bit-detection hardware, mask generator and a bitfield expander. In this way, an algorithm can be mapped onto the PP in far fewer instructions than with traditional DSP and superscalar RISC processors. The multiplier supports signed/signed or unsigned/unsigned (but not signed/unsigned) multiplies when 16-bit input operands are used and signed/unsigned or unsigned/unsigned (but not signed/signed) multiplies when 8-bit input operands are used. The ALU is capable of executing all 256 possible logical operations on three variables. This allows operations that may take several instructions on most DSPs to be performed in a single ALU operation.

The TC is designed to handle all off-chip data transfer operations required by the MP and PPs. The TC is programmed by a 64-byte data structure called a *packet transfer request* (PTREQ), describing the organization of the transfer desired, and by internal processor interrupts. The TC completely controls the bandwidth utilization of the chip. Hardware for prioritizing requests such as cache-service interrupts, individual PTREQ and direct external access requests (DEA) is built into the TC. Each PTREQ can be linked to another PTREQ structure, providing an unbroken transition from one transfer to another, while giving the TC the opportunity to service other transfer types. The MP and each PP may make independent PTREQs; the TC services them according to an internal priority scheme. The TC thus provides the DSP programmer with the necessary tool to interleave data transfer and computation.

The same MVP's memory space is used for program and data accesses, and is shared by all of the processors on the chip. The MVP crossbar allows each PP to perform two independent parallel data accesses to the on-chip shared RAMs and one instruction fetch every cycle. Each PP has three crossbar ports. The *global port* connects to any of the shared RAMs. If an access is attempted over this port to an address not in the shared RAMs, a DEA request is sent to the TC. The *local port* connects to any of its local RAMs. When a PP attempts a memory access over this port to an address not in local RAMs, the access is diverted to the global port and tried on the following cycle. Finally, the *instruction port* accesses instructions from the PP's I-cache.

The MVP is not a general-purpose microprocessor. Rather it is designed and used as an extremely fast digital signal processor. Due to its complex architectural structure—a CMP structure with two different kinds of processors, one of which even features a VLIW ISA—it is difficult to develop a compiler that generates efficient code. Therefore, most programs have to be hand-coded in assembly language, which is not at all easy.

4.1.2. Hydra chip multiprocessor

While the TI MVP is an existing commercial microprocessor, the Hydra chip multiprocessor is simulated in software to evaluate the CMP alternatives for future 10^9-transistor chips. The Hydra proposal [122] is composed of four two-issue superscalar CPUs on a single chip. Each of the CPUs is similar to a MIPS R10000 processor with reduced functionality and is attached to its own on-chip primary I- and D-caches. In addition, a single, unified secondary cache is included on the chip (Figure 13).

The Hydra memory system uses a four-level arrangement, that is, a primary, secondary, tertiary SRAM cache and a DRAM main memory. It is a shared-secondary cache CMP.

The individual primary I- and D-caches are designed to supply each CPU in a single cycle in order to support the very high memory bandwidth needed to sustain processor performance. Each primary cache has 16 kbyte capacity and is organized in 32-byte lines. The connection to its CPU is provided by a 64-bit bus. In order to maintain cache coherence, each D-cache must snoop the write bus and invalidate any lines to which other CPUs write. Measurements have shown that, in this way, in typical applications more than 90% of loads hit in primary caches and thus save the progress further down the memory hierarchy. The large 512 kbyte secondary cache acts as a large on-chip cache to back up the primary caches with a nearby memory which is five or more cycles slower than primary. The secondary cache is a kind of write buffer between the CPUs and the outside world. On the other hand, the secondary also acts as a communication medium through which the four CPUs can communicate using shared-memory mechanisms. The 8 Mbyte off-chip tertiary cache has an access time of 10 cycles to the first

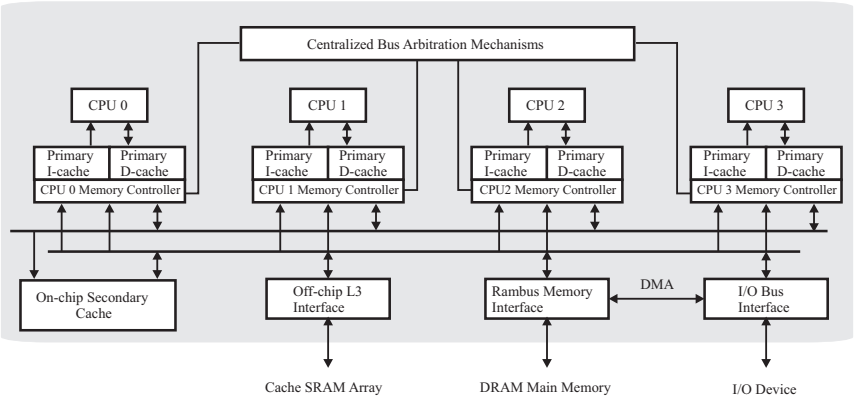

FIGURE 13. A schematic overview of Hydra.

word and is accessed through a 128-bit port that operates at half the processor speed. Even with a large tertiary cache, applications exist with large enough working sets to miss in all of the caches frequently. For this reason, up to 128 Mbyte of DRAM with at least 50 cycles access time can be attached to the Hydra chip via a Rambus memory interface.

The *read bus* and *write bus* are the principal paths of communication across the Hydra chip. The read and the write bus are 256-bit and 64-bit wide, respectively. Hammond and Olukotun [122] found that the contention for both the read bus and the write bus slows performance by only a few percent over a perfect crossbar, even in the worst cases. Furthermore, with the write bus-based architecture, no sophisticated coherence protocols are necessary to keep the on-chip caches coherent.

The 1998 Hydra CMP [122] addresses an expensive, high-end design, with many high-speed SRAM and DRAM chips, directly attached to the Hydra chip. Alternative designs are possible, however, in systems with different constraints. One interesting alternative is a design with no off-chip tertiary at all. In this way the system cost can be reduced dramatically since expensive SRAM chips are eliminated and the number of I/Os on the Hydra chip is halved. Another alternative is that the secondary cache is replaced with on-chip DRAM, thus making the tertiary cache superfluous. The performance of a Hydra CMP with 256 Mbyte DRAM is evaluated in [126]. On floating-point applications with large working sets, the on-chip DRAM Hydra performed on average 52% faster than the secondary-on-chip/tertiary-off-chip Hydra.

Assuming a shared (primary or secondary) cache, the interthread data exchange can be handled efficiently. Consequently, speculative loop iteration threads and data speculation support are proposed for the Hydra single chip multiprocessor [127].

4.1.3. IBM POWER4

The IBM POWER4 chip [123] is a symmetric CMP with two 64-bit processors running at 1.3 GHz with 64 kbyte L1 I-cache and 32 kbyte L1 D-cache. Each processor can fetch up to 8 IPC and has a sustained completion rate of up to 5 IPC. The POWER4 chip has 174 million transistors [128].

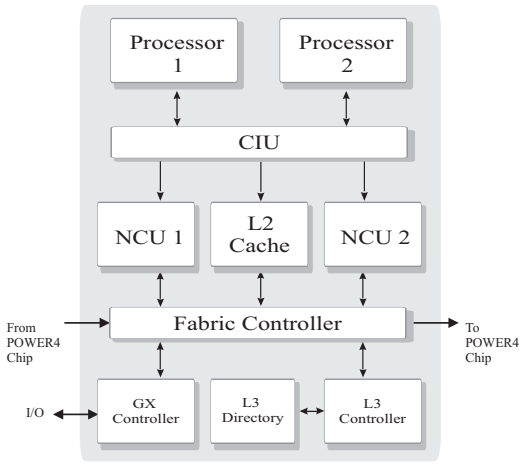

FIGURE 14. The POWER4 chip (NCU, non-cacheable unit; CIU, core interface unit).

Four POWER4 chips are packed in a single multi-chip module (MCM) as an eight-processor SMP. Four MCMs can be interconnected to form a 32-processor SMP.

The components of the POWER4 chip are shown in Figure 14. The two processors share a unified 1.41 Mbyte L2 cache through a *core interface unit* (CIU). The CIU is a crossbar switch between the L2 cache and the two processors. Each processor has associated with it a *non-cacheable unit* (NCU) responsible for handling instruction-serializing functions and performing any non-cacheable operations in the storage hierarchy. Although the L3 cache is on a separate chip, its *L3 directory* and the *L3 controller* are located on the POWER4 chip. The *fabric controller* provides master control of the internal buses, performs snooping and coherency duties and directs unidirectional external buses to/from the other POWER4 chips residing on the same or neighboring MCMs. The GX controller is responsible for controlling the flow of information in and out of the system.

The POWER4 was announced at Microprocessor Forum 1999 and is to be shipped in 2002.

TABLE 1. IPC results of the Tullsen *et al.* [73] and the Sigmund and Ungerer [81] simulations.

Number × (threads, issue)	Tullsen *et al.*	Sigmund and Ungerer
1 × (8, 8)	6.64	4.19
8 × (1, 1)	5.13	6.07
2 × (4, 4)	6.80	6.80
1 × (4, 8)	4.15	3.37
4 × (1, 2)	3.44	4.32
2 × (1, 4)	1.94	2.56

5. SIMULTANEOUS MULTITHREADING VERSUS CHIP MULTIPROCESSOR

5.1. Simulations

In this section, we compare the two main architectural principles capable of exploiting *multiple threads* of instructions, i.e. thread-level (or coarse-grained) parallelism, namely the CMP and the SMT approaches.

Sigmund and Ungerer [81] simulated various configurations of the SMT model in combination with the CMP approach and compared them to Tullsen *et al.*'s [73] simulations (see Table 1). The simulations produced slightly different results to Tullsen *et al.*'s simulations, especially viewing the eight-threaded, eight-issue superscalar approach $1 \times (8, 8)$ (in Table 1 $p \times (t, i)$ means p processors per chip, each processor equipped with t threads and i issue slots).

The reason for the difference in results follows from the high number of execution units in Tullsen *et al.*'s approach; for example, up to eight load/store units are used in Tullsen *et al.*'s simulation, ignoring hardware cost and design problems, whereas the performance of Sigmund and Ungerer's SMT model is restricted by the assumption of a single load/store unit. In Tullsen *et al.*'s simulations the SMT approach performs better than the CMP approach, whereas in Sigmund and Ungerer's simulations the CMP reaches a higher throughput than the SMT approach, when using the same issue bandwidth and number of threads (comparing the SMT of $1 \times (8, 8)$ with the CMP of $8 \times (1, 1)$). However, if chip costs are taken into consideration, a four-threaded, four-issue superscalar processor shows the best performance/cost relation [81].

Further simulations by Eggers *et al.* [36] compared SMT, wide-issue superscalar, interleaved multithreading superscalar and two-CPU and four-CPU CMPs. Comparison of the simulated processor architecture configurations is given in Table 2.

The simulation results which are given in Table 3 were obtained on a workload which consisted of a group of coarse-grained (parallel threads) and medium-grained (parallel loop iterations) parallel programs.

The average instruction throughput of an eight-issue superscalar was an IPC of 3.3, which is already high compared to other measured superscalar IPCs, but rather low compared to the eight instructions possibly issued per cycle. The superscalar's inability to exploit more ILP or any thread-level parallelism contributed to its lower performance. By exploiting thread-level parallelism, an interleaved multithreading superscalar technique provided an average instruction throughput of 4.2 IPC. This IPC occurred with only four threads while performance fell with additional threads. One of the reasons for this is that an interleaved multithreading superscalar can issue instructions from only one thread each cycle and therefore cannot hide conflicts from interthread competition for shared resources. SMT obtained better speedups than CMP2 and CMP4, the latter being CMPs with respectively two four-issue and four two-issue CPUs. Speedups on the CMPs were hindered by the fixed partitioning of their hardware resources across the CPUs. Bridging of latencies is only possible in the multithreaded processor approaches and not in CMPs. CPUs in CMPs were idle when thread-level parallelism was insufficient. Exploiting large amounts of ILP in the unrolled loops of individual threads was not possible due to the CPU's smaller issue bandwidth in CMPs. On the other hand, an SMT processor dynamically partitions its resources among threads, and therefore can respond well to variations in both types of parallelism, exploiting them interchangeably.

In contrast to Eggers *et al.* [36] who compared architectures having constant total issue bandwidth (i.e. number of CPUs × CPU issue bandwidth), Hammond *et al.* [129] established a standard chip area and integration density, and determined the parameters for three architectures: superscalar, CMP and SMT (Table 4). They argue that design complexity for a 16-issue CMP is similar to that of a 12-issue superscalar or a 12-issue SMT processor.

In this case, a CMP with eight two-issue CPUs outperforms a 12-issue superscalar and a 12-issue, eight-threaded SMT processor on four `SPEC95` benchmark programs. Figure 15 shows the performance of the superscalar, SMT and CMP on the four benchmarks relative to a single two-issue superscalar.

The CMP achieved higher performance than the SMT due to a total of 16 issue slots instead of 12 issue slots for the SMT.

5.2. Discussion

The performance race between SMT and CMP has yet to be decided. Certainly, CMP will be easier to implement, but only SMT has the ability to hide latencies. A functional partitioning as required by the on-chip wire-delay of future microprocessors is not easily achieved with a SMT processor due to the centralized instruction issue. A separation of the thread queues as in the Karlsruhe multithreaded superscalar processor is a possible solution, although it does not remove the central instruction issue.

A combination of SMT and CMP is proposed in [81] and in [130]. Perhaps the simulations of Sigmund and Ungerer show the desirability of a CMP consisting of moderately equipped (e.g. four-threaded, four-issue superscalar) SMTs.

Future CMPs will most likely be SMPs and usually feature separate primary I- and D-caches per on-chip CPU and an optional unified secondary cache. If the CPUs always execute threads of the same process, the secondary cache

TABLE 2. Processor architectures simulated by Eggers *et al.* [36].

Features	Superscalar	Interleaved multithreading superscalar	SMT	CMP2	CMP4
No. of CPUs	1	1	1	2	4
CPU issue bandwidth	8	8	8	4	2
No. of threads	1	8	8	1 per CPU	1 per CPU
No. of architectures registers	32	32 per thread	32 per thread	32 per CPU	32 per CPU

TABLE 3. IPC when executing a parallel workload.

Threads	Superscalar	Interleaved multithreading superscalar	SMT	CMP2	CMP4
1	3.3	3.3	3.3	2.4	1.5
2	NA	4.1	4.7	4.3	2.6
4	NA	4.2	5.6	NA	4.2
8	NA	3.5	6.1	NA	NA

TABLE 4. Processor architectures simulated by Hammond *et al.* [129].

Features	Superscalar	CMP	SMT
No. of CPUs	1	8	1
CPU issue bandwidth	12	2 per CPU	12
No. of threads	1	1 per CPU	8
No. of architectural registers	32	32 per CPU	32 per thread

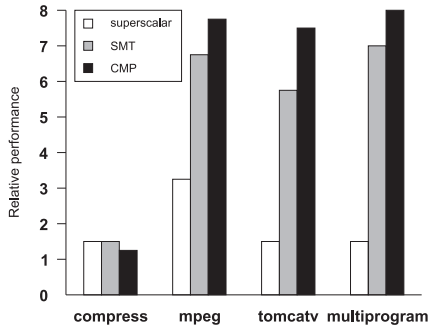

FIGURE 15. Relative performance of superscalar, SMT and CMP.

organization will be simplified, because different processes do not have to be distinguished.

Similarly, if all (hardware-supported) threads of a SMT processor always execute threads of the same process, preferably in SPMD fashion, a unified (primary) I-cache may prove useful, since the code can be shared between the threads. The primary D-cache may be unified or separated between the threads depending on the access mechanism used.

If CMP or SMT are the design choice of the future, the impact on multiprocessor development will favor shared-memory multiprocessors (either SMPs or DSMs) over message-passing machines. In the future, we will observe merging of SMT and CMP with today's multiple-issue processor techniques.

Sohi [131] expects that the distinction between the SMT and CMP microarchitectures is likely to be blurred over time, as we have already seen for the implicit multithreading approaches above. Increasing wire delays will require decentralization of critical processor functionality favouring CMP, while flexible resource allocation policies will enhance resource sharing as in SMT. In either case, multithreaded processors will logically appear to be collections of processing elements with additional support for speculative execution. Thus, in addition to executing parallel threads, the logical processors could execute single programs that are divided into speculative threads [131].

6. CONCLUSIONS

Depending on the specific multithreaded processor design, only a single-issue instruction pipeline (as in scalar RISC processors) is used or a single issue unit issues instructions from different instruction streams simultaneously. The latter are called simultaneous multithreaded processors and they combine the multithreading technique with a wide-issue superscalar processor.

We further distinguish between implicit multithreaded processor techniques that concurrently execute several threads from a sequential program and explicit multithreaded processors that are able to execute threads (lightweighted processes) of several processes concurrently. Note that explicit multithreaded processors aim at a low execution time of a multithreaded workload, while superscalar and

implicit multithreaded processors aim at a low execution time of a single program.

We expect explicit multithreaded processor techniques, in particular the simultaneous multithreading techniques, as well as chip multiprocessor techniques to be used in the next generation of microprocessors. Recently announced are SMT and CMP processors by IBM (POWER4), Compaq (Alpha 21464) and Sun (MAJC-5200). While POWER4 is a CMP processor and the abandoned Alpha 21464 was a SMT processor, the MAJC processor has two CPUs on the same chip with each CPU being a four-threaded blocked interleaving VLIW processor [132].

Still hot research trends are the implicit multithreaded processor techniques and the related helper thread approach. Both target the performance increase of single-threaded programs by dynamically utilizing speculative thread-level parallelism.

Moreover, we expect that future research will also focus on the deployment of multithreading techniques in the fields of signal processors, microcontrollers, in particular for real-time applications, and for power management.

This survey paper should clarify the terminology and demonstrate the research results achieved for such new architectural approaches.

REFERENCES

[1] Patt, Y. N. *et al.* (1997) One billion transistors, one uniprocessor, one chip. *Computer*, **30**, 51–57.

[2] Šilc, J., Ungerer, T. and Robič, B. (2000) A survey of new research directions in microprocessors. *Microproc. Microsyst.*, **24**, 175–190.

[3] Lam, M. S. and Wilson, R. P. (1992) Limits of control flow on parallelism. In *Proc. 18th ISCA*, Toronto, Canada, May 27–30, pp. 46–57. ACM Press, New York.

[4] Wall, D. W. (1991) Limits of instruction-level parallelism. In *Proc. Int. Conf. ASPLOS-IV*, Santa Clara, CA, April 8–11, pp. 176–188. ACM Press, New York.

[5] Butler, M. *et al.* (1991) Single instruction stream parallelism is greater than two. In *Proc. 18th ISCA*, Toronto, Canada, May 27–30, pp. 276–286. ACM Press, New York.

[6] Lipasti, M. H. and Shen, J. P. (1997) Superspeculative microarchitecture for beyond AD 2000. *Computer*, **30**, 59–66.

[7] Chrysos, G. Z. and Emer, J. S. (1998) Memory dependence prediction using store sets. In *Proc. 25th ISCA*, Barcelona, Spain, June 30–July 4, pp. 142–153. ACM/IEEE Computer Society Press, Los Alamitos, CA.

[8] Lipasti, M. H. and Shen, J. P. (1997) The performance potential of value and dependence prediction. *Lecture Notes Comput. Sci.*, **1300**, 1043–1052.

[9] Lipasti, M. H., Wilkerson, C. B. and Shen, J. P. (1996) Value locality and load value prediction. In *Proc. Int. Conf. ASPLOS-VII*, Cambridge, MA, October 1–5, pp. 138–147. ACM Press, New York.

[10] Rychlik, B. *et al.* (1998) Efficiency and performance impact of value prediction. In *Proc. Conf. PACT*, Paris, France, October 13–17, pp. 148–154. IEEE Computer Society Press, Los Alamitos, CA.

[11] Iannucci, R. A., Gao, G. R., Halstead, R. and Smith, B. (1994) *Multithreaded Computer Architecture: A Summary of the State of the Art*. Kluwer Academic Publishers, Dordrecht.

[12] Šilc, J., Robič, B. and Ungerer, T. (1999) *Processor Architecture: From Dataflow to Superscalar and Beyond*. Springer, Berlin.

[13] Smith, B. J. (1981) Architecture and applications of the HEP multiprocessor computer system. *SPIE Real-Time Signal Processing IV*, **298**, 241–248.

[14] Lee, D. C. *et al.* (1998) Execution characteristics of desktop applications on Windows NT. In *Proc. 25th ISCA*, Barcelona, Spain, June 30–July 4, pp. 27–38. ACM/IEEE Computer Society Press, Los Alamitos, CA.

[15] Sohi, G. S. and Roth, A. (2001) Speculative multithreaded processor. *Computer*, **34**, 66–73.

[16] Franklin, M. (1993) *The Multiscalar Architecture*. Computer Science Technical Report No. 1196, University of Wisconsin-Madison, WI.

[17] Sohi, G. S. (1997) Multiscalar: another fourth-generation processor. *Computer*, **30**, 72.

[18] Sohi, G. S., Breach, S. E. and Vijaykumar, T. N. (1995) Multiscalar processors. In *Proc. 22nd ISCA*, Santa Margherita Ligure, Italy, June 22–24, pp. 414–425. ACM Press, New York.

[19] Vijaykumar, T. N. and Sohi, G. S. (1998) Task selection for a multiscalar processor. In *Proc. 31st Int. Symp. MICRO*, Dallas, TX, November 30–December 2, pp. 81–92. IEEE Computer Society, Los Alamitos, CA.

[20] Rotenberg, E. *et al.* (1997) Trace processors. In *Proc. 30th Int. Symp. MICRO*, Research Triangle Park, NC, December 1–3, pp. 138–148. IEEE Computer Society, Los Alamitos, CA.

[21] Smith, J. E. and Vajapeyam, S. (1997) Trace processors: moving to fourth-generation microarchitectures. *Computer*, **30**, 68–74.

[22] Vajapeyam, S. and Mitra, T. (1997) Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. In *Proc. 24th ISCA*, Denver, CO, June 2–4, pp. 1–12. ACM/IEEE Computer Society Press, Los Alamitos, CA.

[23] Dubey, P. K. *et al.* (1995) *Single-program Speculative Multithreading (SPSM) Architecture: Compiler-assisted Fine-grain Multithreading*. IBM Research Report RC 19928 (02/06/95), Yorktown Heights, New York.

[24] Li, Z. *et al.* (1996) Compiler techniques for concurrent multithreading with hardware speculation support. *Lecture Notes Comput. Sci.*, **1239**, 175–191.

[25] Tsai, J.-Y. and Yew, P.-C. (1996) The superthreaded architecture: thread pipelining with run-time data dependence checking and control speculation. In *Proc. Conf. PACT*, Boston, MA, October, pp. 35–46. IEEE Computer Society Press, Los Alamitos, CA.

[26] Akkary, H. and Driscoll, M.A. (1998) A dynamic multithreading processor. In *Proc. 31st Int. Symp. MICRO*, Dallas, TX, November 30–December 2, pp. 226–236. IEEE Computer Society, Los Alamitos, CA.

[27] Marcuello, P., Gonzales, A. and Tubella, J. (1998) Speculative multithreaded processors. In *Proc. Int. Conf. Supercomp.*, Melbourne, Australia, July 13–17, pp. 77–84. ACM Press, New York.

[28] Klauser, A. *et al.* (1998) Dynamic hammock predication for non-predicated instruction sets. In *Proc. Conf. PACT*, Paris, France, October 13–17, pp. 278–285. IEEE Computer Society Press, Los Alamitos, CA.

[29] Klauser, A., Paithankar, A. and Grunwald, D. (1998) Selective eager execution on the PolyPath architecture. In *Proc. 25th ISCA*, Barcelona, Spain, June 30–July 4, pp. 250–259. ACM/IEEE Computer Society Press, Los Alamitos, CA.

[30] Chappell, R. S. *et al.* (1999) Simultaneous subordinate microthreading (SSMT). In *Proc. 26th ISCA*, Atlanta, GA, May 2–4, pp. 186–195. ACM/IEEE Computer Society Press, Los Alamitos, CA.

[31] Culler, D. E., Singh, J. P. and Gupta, A. (1999) *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, San Francisco.

[32] Barroso, L. A., Gharachorloo, K. and Bugnion, E. (1998) Memory system characterization of commercial workloads. In *Proc. 25th ISCA*, Barcelona, Spain, June 30–July 4, pp. 3–14. ACM/IEEE Computer Society Press, Los Alamitos, CA.

[33] Alverson, R. *et al.* (1990) The Tera computer system. In *Proc. Int. Conf. Supercomputing*, Amsterdam, The Netherlands, June, pp. 1–6.

[34] Dennis, J. B. and Gao, G. R. (1994) Multithreaded architectures: principles, projects, and issues. In Iannucci, R. A. *et al.* (eds), *Multithreaded Computer Architecture: A Summary of the State of the Art*. Kluwer Academic Publishers, Dordrecht.

[35] Šilc, J., Robič, B. and Ungerer, T. (1998) Asynchrony in parallel computing: from dataflow to multithreading. *Parall. Distr. Comput. Practices*, **1**, 57–83.

[36] Eggers, S. J. *et al.* (1997) Simultaneous multithreading: a platform for next-generation processors. *IEEE Micro*, **17**, 12–19.

[37] Laudon, J., Gupta, A. and Horowitz, M. (1994) Interleaving: a multithreading technique targeting multiprocessors and workstations. In *Proc. Int. Conf. ASPLOS-VI*, San Jose, CA, October 4–7, pp. 308–318. ACM Press, New York.

[38] Smith, B. J. (1985) The architecture of HEP. In Kowalik, J. S. (ed.), *Parallel MIMD Computation: HEP Supercomputer and Its Applications*. MIT Press, Cambridge, MA.

[39] Thistle, M. and Smith, B. J. (1988) A processor architecture for Horizon. In *Proc. Supercomputing Conf.*, Orlando, FL, November, pp. 35–41. IEEE Computer Society Press, Los Alamitos, CA.

[40] Halstead, R. H. and Fujita, T. (1988) MASA: a multithreaded processor architecture for parallel symbolic computing. In *Proc. 15th ISCA*, Honolulu, HW, May–June, pp. 443–451. IEEE Computer Society Press, Los Alamitos, CA.

[41] Formella, A., Keller, J. and Walle, T. (1996) HPP: a high performance PRAM. *Lecture Notes Comput. Sci.*, **1123**, 425–434.

[42] Hansen, C. (1996) MicroUnity's MediaProcessor architecture. *IEEE Micro*, **16**, 34–41.

[43] Alverson, G. *et al.* (1995) Scheduling on the Tera MTA. *Lecture Notes Comput. Sci.*, **949**, 19–44.

[44] Fillo, M. *et al.* (1995) The M-machine multicomputer. In *Proc. 28th Int. Symp. MICRO*, Ann Arbor, MI, November 29–December 1, pp. 146–156. IEEE Computer Society, Los Alamitos, CA.

[45] Bach, P. *et al.* (1997) Building the 4 processor SB-PRAM prototype. In *Proc. 30th Hawaii Int. Symp. Sys. Sci.*, January, pp. 5:14–23. IEEE Computer Society Press, Los Alamitos, CA.

[46] Sterling, T. (1997) Beyond 100 teraflops through superconductors, holographic storage, and the data vortex. In *Proc. Int. Symp. on Supercomputing*, Tokyo, Japan.

[47] Dorojevets, M. (2000) COOL multithreading in HTMT SPELL-1 processors. *Int. J. High Speed Electron. Sys.*, **10**, 247–253.

[48] Kreuzinger, J. and Ungerer, T. (1999) Context-switching techniques for decoupled multithreaded processors. In *Proc. 25th Euromicro Conf.*, Milano, Italy, September 4–7, pp. 1:248–251. IEEE Computer Society Press, Los Alamitos, CA.

[49] Boothe, R. F. (1993) *Evaluation of Multithreading and Caching in Large Shared Memory Parallel Computers*. Technical Report UCB/CSD-93-766, Computer Science Division, University of California, Berkeley, CA.

[50] Boothe, R. F. and Ranade, A. (1992) Improved multithreading techniques for hiding communication latency in multiprocessors. In *Proc. 19th ISCA*, Gold Coast, Australia, May, pp. 214–223. ACM Press, New York.

[51] Agarwal, A., Kubiatowicz, J., Kranz, D. *et al.* (1993) Sparcle: an evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, **13** (June), 48–61.

[52] Mikschl, A. and Damm, W. (1996) Msparc: a multithreaded Sparc. *Lecture Notes Comput. Sci.*, **1123**, 461–469.

[53] Mankovic, T. E., Popescu, V. and Sullivan, H. (1987) CHoPP principles of operations. In *Proc. 2nd Int. Supercomputer Conf.*, May, pp. 2–10.

[54] Grünewald, W. and Ungerer, T. (1996) Towards extremely fast context switching in a block multithreaded processor. In *Proc. 22nd Euromicro Conf.*, Prague, Czech Republic, September 2–5, pp. 592–599. IEEE Computer Society Press, Los Alamitos, CA.

[55] Grünewald, W. and Ungerer, T. (1997) A multithreaded processor designed for distributed shared memory systems. In *Proc. Int. Conf. Advances in Parall. Distrib. Comput.*, Shanghai, China, March, pp. 206–213.

[56] Dally, W. J. *et al.* (1992) The message-driven processor: a multicomputer processing node with efficient mechanisms. *IEEE Micro*, **12**, 23–39.

[57] Agarwal, A. *et al.* (1995) The MIT Alewife machine: architecture and performance. In *Proc. 22nd ISCA*, Santa Margherita Ligure, Italy, June 22–24, pp. 2–13. ACM Press, New York.

[58] Kavi, K. M., Levine, D. L. and Hurson, A. R. (1997) A non-blocking multithreaded architecture. In *Proc. 5th Int. Conf. Advanced Comput.*, Madras, India, December, pp. 171–177.

[59] Gwennap, L. (1997) DanSoft develops VLIW design. *Microproc. Report*, **11**, 18–22.

[60] Bolychevsky, A., Jesshope, C. R. and Muchnik, V. B. (1996) Dynamic scheduling in RISC architectures. *IEE Proc. Comput. Dig. Tech.*, **143**, 309–317.

[61] Jesshope, C. R. and Luo, B. (2000) Micro-threading: a new approach to future RISC. In *Proc. ACAC*, Canberra, Australia, January 31–February 3, pp. 34–41.

[62] Jesshope, C. R. (2001) Implementing an efficient vector instruction set in a chip multi-processor using micro-threaded pipelines. *Australia Comp. Sci. Commun.*, **23**, 80–88.

[63] Tremblay, M. (1999) A VLIW convergent multiprocessor system on a chip. In *Proc. Microprocessor Forum*, San Jose, CA.

[64] Tremblay, M. *et al.* (2000) The MAJC architecture: a synthesis of parallelism and scalability. *IEEE Micro*, **20**, 12–25.

[65] Borkenhagen, J. M. *et al.* (2000) A multithreaded PowerPC processor for commercial servers. *IBM J. Res. Develop.*, **44**, 885–898.

[66] Dorozhevets, M. N. and Wolcott, P. (1992) The El'brus-3 and MARS-M: recent advances in Russian high-performance computing. *J. Supercomp.*, **6**, 5–48.

[67] Hirata, H. *et al.* (1992) An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proc. 19th ISCA*, Gold Coast, Australia, May, pp. 136–145. ACM Press, New York.

[68] Serrano, M. J., Yamamoto, W., Wood, R. and Nemirovsky, M. D. (1994) Performance estimation in a multi-streamed superscalar processor. *Lecture Notes Comput. Sci.*, **794**, 213–230.

[69] Yamamoto, W. and Nemirovsky, M. D. (1995) Increasing superscalar performance through multistreaming. In *Proc. Conf. PACT*, Limassol, Cyprus, June 26–29, pp. 49–58. IEEE Computer Society Press, Los Alamitos, CA.

[70] Gulati, M. and Bagherzadeh, N. (1996) Performance study of a multithreaded superscalar microprocessor. In *Proc. 2nd Int. Symp. HPCA*, San Jose, CA, February 3–7, pp. 291–301. IEEE Computer Society Press, Los Alamitos, CA.

[71] Loikkanen, M. and Bagherzadeh, N. (1996) A fine-grain multithreading superscalar architecture. In *Proc. Conf. PACT*, Boston, MA, October, pp. 163–168. IEEE Computer Society Press, Los Alamitos, CA.

[72] Pontius, M. and Bagherzadeh, N. (1999) Multithreaded extensions enhance multimedia performance. In *Proc. 3rd Workshop MTEAC*, Orlando, FL, January 9.

[73] Tullsen, D. M., Eggers, S. J. and Levy, H. M. (1995) Simultaneous multithreading: maximizing on-chip parallelism. In *Proc. 22nd ISCA*, Santa Margherita Ligure, Italy, June 22–24, pp. 392–403. ACM Press, New York.

[74] Tullsen, D. M. *et al.* (1996) Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proc. 23rd ISCA*, Philadelphia, PA, May 22–24, pp. 191–202. ACM Press, New York.

[75] Lo, J. L. *et al.* (1997) Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Trans. Comput. Sys.*, **15**, 322–354.

[76] Tullsen, D. M. *et al.* (1999) Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proc. 5th Int. Symp. HPCA*, Orlando, FL, January 9–13, pp. 54–58. IEEE Computer Society Press, Los Alamitos, CA.

[77] Wallace, S., Calder, B. and Tullsen, D. M. (1998) Threaded multiple path execution. In *Proc. 25th ISCA*, Barcelona, Spain, June 27–July 1, pp. 238–249. ACM/IEEE Computer Society Press, Los Alamitos, CA.

[78] Wallace, S., Tullsen, D. M. and Calder, B. (1999) Instruction recycling on a multiple-path processor. In *Proc. 5th Int. Symp. HPCA*, Orlando, FL, January 9–13, pp. 44–53. IEEE Computer Society Press, Los Alamitos, CA.

[79] Lo, J. L. *et al.* (1998) An analysis of database workload performance on simultaneous multithreaded processors. In *Proc. 25th ISCA*, Barcelona, Spain, June 27–July 1, pp. 39–50. ACM/IEEE Computer Society Press, Los Alamitos, CA.

[80] Seng, J. S., Tullsen, D. M. and Cai, G. Z. N. (2000) Power-sensitive multithreaded architecture. In *Proc. ICCD*, Austin, TX, September 17–20, pp. 199–206.

[81] Sigmund, U. and Ungerer, T. (1996) Evaluating a multithreaded superscalar microprocessor versus a multiprocessor chip. In *Proc. 4th PASA Workshop in Parallel Systems and Algorithms*, Jülich, Germany, April, pp. 147–159.

[82] Sigmund, U. and Ungerer, T. (1996) Identifying bottlenecks in multithreaded superscalar multiprocessors. *Lecture Notes Comput. Sci.*, **1123**, 797–800.

[83] Oehring, H., Sigmund, U. and Ungerer, T. (1999) MPEG-2 video decompression on simultaneous multithreaded multimedia processors. In *Proc. Conf. PACT*, Newport Beach, CA, October 12–16, pp. 11–16. IEEE Computer Society Press, Los Alamitos, CA.

[84] Oehring, H., Sigmund, U. and Ungerer, T. (2000) Performance of simultaneous multithreaded multimedia-enhanced processors for MPEG-2 video decompression. *J. Syst. Arch.*, **46**, 1033–1046.

[85] Sigmund, U., Steinhaus, M. and Ungerer, T. (2000) Transistor count and chip space assessment of multimedia-enhanced simultaneous multithreaded processors. In *Proc. 4th Workshop MTEAC*, Monterrey, CA, December 10.

[86] Steinhaus, M. *et al.* (2001) Transistor count and chip space estimation of simple-scalar-based microprocessor models. In *Proc. Workshop on Complexity-Effective Design*, Göteborg, Sweden, June 30.

[87] Burns, J. and Gaudiot, J.-L. (2000) Quantifying the SMT layout overhead—Does SMT pull its weight? In *Proc. 6th Int. Symp. HPCA*, Toulouse, France, January 8–12, pp. 109–120. IEEE Computer Society Press, Los Alamitos, CA.

[88] Espasa, R. and Valero, M. (1997) Exploiting instruction- and data-level parallelism. *IEEE Micro*, **17**, 20–27.

[89] Emer, J. (1999) Simultaneous multithreading: multiplying Alpha's performance. In *Proc. Microprocessor Forum*, San Jose, CA.

[90] Allen, F. *et al.* (2001) Blue Gene: a vision for protein science using a petaflops supercomputer. *IBM Syst. J.*, **40**, 310–326.

[91] Moshovos, A. I. (1998) *Memory Dependence Prediction*. PhD Thesis, University of Wisconsin-Madison.

[92] Gopal, S. *et al.* (1998) Speculative versioning cache. In *Proc. 4th Int. Symp. HPCA*, Las Vegas, NE, January 31–February 4, pp. 195–205. IEEE Computer Society Press, Los Alamitos, CA.

[93] Rotenberg, E. *et al.* (1996) Trace cache: a low latency approach to high bandwidth instruction fetch. In *Proc. 29th Int. Symp. MICRO*, Paris, France, December 2–4, pp. 24–34. IEEE Computer Society, Los Alamitos, CA.

[94] Tubella, J. and Gonzalez, A. (1998) Control speculation in multithreaded processors through dynamic loop detection. In *Proc. 4th Int. Symp. HPCA*, Las Vegas, NE, January 31–February 4, pp. 14–23. IEEE Computer Society Press, Los Alamitos, CA.

[95] Codrescu, L. and Wills, D. S. (2000) On dynamic speculative thread partitioning and the MEM-slicing algorithm. *J. Universal Comp. Sci.*, **6**, 908–927.

[96] Keckler, S. W., Chang, A., Lee, W. S. and Dally, W. J. (1999) Concurrent event handling through multithreading. *IEEE Trans. Comput.*, **48**, 903–916.

[97] Zilles, C. B., Emer, J. S. and Sohi, G. S. (1999) The use of multithreading for exception handling. In *Proc. 32nd Int. Symp. MICRO*, Haifa, Israel, November 16–18, pp. 219–229. IEEE Computer Society Press, Los Alamitos, CA.

[98] Zilles, C. and Sohi, G. (2001) Execution-based prediction using speculative slices. In *Proc. 28th ISCA*, Göteborg,

Sweden, June 30–July 4, pp. 2–13. ACM/IEEE Computer Society Press, Los Alamitos, CA.

[99] Collins, J. D. *et al.* (2001) Speculative precomputation: long-range prefetching of delinquent loads. In *Proc. 28th ISCA*, Göteborg, Sweden, June 30–July 4, pp. 14–25. ACM/IEEE Computer Society Press, Los Alamitos, CA.

[100] Luk, H.-K. (2001) Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proc. 28th ISCA*, Göteborg, Sweden, June 30–July 4, pp. 40–51. ACM/IEEE Computer Society Press, Los Alamitos, CA.

[101] Ranganathan, N. and Franklin, M. (1998) An empirical study of decentralized ILP execution models. In *Proc. Int. Conf. ASPLOS-VIII*, San Jose, CA, 3–7 October, pp. 272–281. ACM Press.

[102] Kavi, K. M., Arul, J. and Giorgi, R. (2000) Execution and cache performance of the Scheduled Dataflow Architecture. *J. Universal Comp. Sci.*, **6**, 948–967.

[103] Unger, A., Ungerer, T. and Zehendner, E. (1998) A compiler technique for speculative execution of alternative program paths targeting multithreaded architectures. In *Proc. Yale Multithreaded Programming Workshop*, New Haven, CT, June.

[104] Unger, A., Ungerer, T. and Zehendner, E. (1998) Static speculation, dynamic resolution. In *Proc. 7th Workshop Compilers for Parallel Computers*, Linkoping, Sweden, June–July, pp. 243–253.

[105] Unger, A., Ungerer, T. and Zehendner, E. (1999) Compiler supported speculative execution on SMT processors. In *Proc. 3rd Workshop MTEAC*, Orlando, FL, 9 January.

[106] Unger, A., Ungerer, T. and Zehendner, E. (1999) Simultaneous speculation scheduling. In *Proc. 11th Symp. on Computer Architecture and High Performance Computing*, Natal, Brazil, 29 September–2 October, pp. 175–182.

[107] Heil, T. H. and Smith, J. E. (1996) *Selective Dual Path Execution*. Technical Report, Department of Electrical and Computer Engineering, University of Wisconsin-Madison.

[108] Tyson, G., Lick, K. and Farrens, M. (1997) *Limited Dual Path Execution*. Technical Report CSE-TR 346-97, University of Michigan.

[109] Uht, A. K. and Sindagi, V. (1995) Disjoint eager execution: an optimal form of speculative execution. In *Proc. 28th Int. Symp. MICRO*, Ann Arbor, MI, November 29–December 1, pp. 313–325. IEEE Computer Society, Los Alamitos, CA.

[110] Jacobsen, E., Rotenberg, E. and Smith, J. E. (1996) Assigning confidence to control conditional branch predictions. In *Proc. 28th Int. Symp. MICRO*, Paris, France, 2–4 December, pp. 142–152. IEEE Computer Society, Los Alamitos, CA.

[111] Wittenburg, J. P., Meyer, G. and Pirsch, P. (1999) Adapting and extending simultaneous multithreading for high performance video signal processing applications. In *Proc. 3rd Workshop MTEAC*, Orlando, FL, January 9.

[112] Metzner, A. and Niehaus, J. (2000) MSPARC: multithreading in real-time architectures. *J. Universal Comp. Sci.*, **6**, 1034–1051.

[113] Lüth, K., Metzner, A., Piekenkamp, T. and Risu, J. (1997) The events approach to rapid prototyping for embedded control system. In *Proc. Workshop Zielarchitekturen eingebetteter Systeme*, pp. 45–54.

[114] Brinkschulte, U. *et al.* (1999) A multithreaded Java microcontroller for thread-oriented real-time event-handling. In *Proc. Conf. PACT*, Newport Beach, CA, October 12–16, pp. 34–39. IEEE Computer Society Press, Los Alamitos, CA.

[115] Brinkschulte, U. *et al.* (1999) The Komodo project: thread-based event handling supported by a multithreaded Java microcontroller. In *Proc. 25th Euromicro Conf.*, Milano, Italy, September 4–7, pp. 2:122–128. IEEE Computer Society Press, Los Alamitos, CA.

[116] Kreuzinger, J. *et al.* (2000) Real-time scheduling on multithreaded processors. In *Proc. 7th Int. Conf. Real-Time Comp. Sys. and Applications*, Cheju Island, South Korea, December, pp. 155–159.

[117] Brinkschulte, U. *et al.* (2001) A microkernel middleware architecture for distributed embedded real-time systems. In *Proc. 20th IEEE Symp. Reliable Distributed Systems*, New Orleans, October 28–31, pp. 218–226.

[118] Fuhrmann, S. *et al.* (2001) Real-time garbage collection for a multithreaded Java microcontroller. In *Proc. 4th IEEE Int. Symp. Object-Oriented Real-Time Distr. Comput.*, Magdeburg, Germany, May, pp. 69–76.

[119] Milutinović, V. (2000) *Surviving the Design of Microprocessor and Multimicroprocessor Systems: Lessons Learned*. Wiley-Interscience, New York.

[120] Nayfeh, B.A., Hammond, L. and Olukotun, K. (1996) Evaluation of design alternatives for a multiprocessor microprocessor. In *Proc. 23rd ISCA*, Philadelphia, PA, May 22–24, pp. 67–77. ACM Press, New York.

[121] Olukotun, K. *et al.* (1996) The case for a single-chip multiprocessor. In *Proc. Int. Conf. ASPLOS-VII*, Cambridge, MA, October 1–5, pp. 2–11. ACM Press, New York.

[122] Hammond, L. and Olukotun, K. (1998) *Considerations in the Design of Hydra: A Multiprocessor-on-chip Microarchitecture*. Technical Report CSL-TR-98-749, Computer Systems Laboratory, Stanford University.

[123] Diefendorff, K. (1999) Power4 focuses on memory bandwidth. *Microproc. Rep.*, **13**.

[124] Texas Instruments (1994) *TMS320C80 Technical Brief*. Texas Instruments, Houston, TX.

[125] Golston, J. (1996) Single-chip H.324 videoconferencing. *IEEE Micro*, **16**, 21–33.

[126] Yamauchio, T., Hammond, L. and Olukotun, K. (1997) *A Single Chip Multiprocessor Integrated with DRAM*. Technical Report CSL-TR-97-731, Computer Systems Laboratory, Stanford University.

[127] Hammond, L., Willey, M. and Olukotun, K. (1998) Data speculation support for a chip multiprocessor. In *Proc. Int. Conf. ASPLOS-VIII*, San Jose, CA, October 3–7, pp. 58–69. ACM Press, New York.

[128] Warnock, J. D. *et al.* (2002) The circuit and physical design of the POWER4 microprocessor. *IBM J. Res. Develop.*, **46**, 27–52.

[129] Hammond, L., Nayfeh, B. A. and Olukotun, K. (1997) A single-chip multiprocessor. *Computer*, **30**, 79–85.

[130] Krishnan, V. and Torellas, J. (1998) A clustered approach to multithreaded processors. In *Proc. IPPS/SPDP Conf.*, Orlando, FL, March–April, pp. 627–634.

[131] Sohi, G. S. (2000) Microprocessors—10 years back, 10 years ahead. *Lecture Notes Comp. Sci.*, **2000**, 208–218.

[132] Sudharsanan, S. (2000) MAJC-5200: a high performance microprocessor for multimedia computing. *Lecture Notes Comput. Sci.*, **1800**, 163–170.