A Survey of Processors with Explicit Multithreading

THEO UNGERER

University of Augsburg

BORUT ROBIČ

University of Ljubljana

AND

JURIJ ŠILC

Jožef Stefan Institute

Hardware multithreading is becoming a generally applied technique in the next generation of microprocessors. Several multithreaded processors are announced by industry or already into production in the areas of high-performance microprocessors, media, and network processors.

A multithreaded processor is able to pursue two or more threads of control in parallel within the processor pipeline. The contexts of two or more threads of control are often stored in separate on-chip register sets. Unused instruction slots, which arise from latencies during the 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 thread contexts that are loaded 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 to utilize a larger part of the issue bandwidth by issuing instructions from different threads simultaneously.

Explicit multithreaded processors are multithreaded processors that apply processes or operating system threads in their hardware thread slots. These processors optimize the throughput of multiprogramming workloads rather than single-thread performance. We distinguish these processors from implicit multithreaded processors

Categories and Subject Descriptors: C.1 [**Computer Systems Organization**]: Processor Architectures; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Pipeline processors*

General Terms: Design, Performance

Additional Key Words and Phrases: Blocked multithreading, interleaved multithreading, simultaneous multithreading

---

Authors' addresses: T. Ungerer, Institute of Computer Science, University of Augsburg, Eichleitnerstrasse 30, D-86135 Augsburg, Germany; email: ungerer@informatik.uni-augsburg.de; B. Robič, Faculty of Computer and Information Science, University of Ljubljana, Tržaška 25, Sl-1000 Ljubljana, Slovenia; email: borut.robic@fri.uni-lj.si; J. Šilc, Computer Systems Department, Jožef Stefan Institute, Jamova 39, Sl-1000 Ljubljana, Slovenia; email: jurij.silc@ijs.si.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
©2003 ACM 0360-0300/03/0300-0029 $5.00

that utilize thread-level speculation by speculatively executing compiler- or machine-generated threads of control that are part of a single sequential program.

This survey paper explains and classifies the explicit multithreading techniques in research and in commercial microprocessors.

1. INTRODUCTION

A multithreaded processor is able to concurrently execute instructions of different threads of control within a single pipeline. The minimal requirement for such a processor is the ability to pursue two or more threads of control in parallel within the processor pipeline, that is, the processor must provide two or more independent program counters, an internal tagging mechanism to distinguish instructions of different threads within the pipeline, and a mechanism that triggers a thread switch. Thread-switch overhead must be very low, from zero to only a few cycles. Multithreaded processor features often, but not always, multiple register sets on the processor chip.

The current interest in hardware multithreading stems from three objectives:

—Latency reduction is an important task when designing a microprocessor. Latencies arise from data dependencies between instructions within a single thread of control. Long latencies are caused by memory accesses that miss in the cache and by long running instructions. Short latencies may be bridged within a superscalar processor by executing succeeding, nondependent instructions of the same thread. Long latencies, however, stall the processor and lessen its performance.

—Shared-memory multiprocessors suffer from memory access latencies that are several times longer than in a single-processor system. When accessing a nonlocal memory module in a distributed-shared memory system, the memory latency is enhanced by the transfer time through the communication network. Additional latencies arise in a shared memory multiprocessor from thread synchronizations, which cause idle times for the waiting thread. One solution to fill these idle times is to switch to another thread. However, a thread switch on a conventional processor causes saving of all registers, loading the new register values, and several more administrative tasks that often require too much time to prove this method as an efficient solution.

—Contemporary superscalar microprocessors [Šilc et al. 1999] are able to issue four to six instructions each clock cycle from a conventional linear instruction stream. VLSI technology will allow future microprocessors with an issue bandwidth of 8–32 instructions per cycle [Patt et al. 1997]. 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 single control flow branch speculation have reported parallelism around 7 instructions per cycle (IPC) with infinite resources [Wall 1991; Lam and Wilson 1992] and around 4 IPC with large sets of resources (e.g., 8 to 16 execution units) [Butler et al. 1991]. 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. Control speculation may be enhanced by data dependence and value speculation techniques to increase performance of a single program thread [Lipasti et al. 1996; Lipasti and Shen 1997; Chrysos and Emer 1998; Rychlik et al. 1998].

Multithreading pursues a different set of solutions by utilizing coarse-grained parallelism [Iannucci et al. 1994].

The notion of a thread in the context of multithreaded processors differs from

the notion of software threads in multithreaded operating systems. In 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), an operating system thread (a light-weighted process, e.g., a POSIX thread), a compiler-generated thread (subordinate microthread, microthread, nanothread etc.), or even a hardware-generated thread.

1.1. Explicit and Implicit Multithreading

The solution surveyed in this paper is the utilization of coarser-grained parallelism by *explicit multithreaded processors* that interleave the execution of instructions of different user-defined threads (operating system threads or processes) in the same pipeline. 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. Thread-switching is performed automatically by the processor due to a hardware-based thread-switching policy. This ability is in contrast to conventional processors or today's superscalar processors, which use busy waiting or a time-consuming, operating system-based thread switch.

Depending on the specific multithreaded processor design, either a single-issue instruction pipeline (as in scalar processors) is used, or multiple instructions from possibly 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 more often utilized by potentially issuing instructions from different threads simultaneously.

A different approach increases the performance of sequential programs by applying thread-level speculation. A thread in such processor proposals refers to any contiguous region of the static or dynamic instruction sequence. We call such processors *implicit multithreaded processors*, which refers to any processor that can dynamically generate threads from single-threaded programs and execute such speculative threads concurrent with the lead thread. In case of misspeculation, all speculatively generated results must be squashed. Threads generated by implicit multithreaded processors are always executed speculatively, in contrast to the threads in explicit multithreaded processors.

Examples of implicit multithreaded processor proposals are the multiscalar processor [Franklin 1993; Sohi et al. 1995; Sohi 1997; Vijaykumar and Sohi 1998], the trace processor [Rotenberg et al. 1997; Smith and Vajapeyam 1997; Vajapeyam and Mitra 1997], the single-program speculative multithreading architecture [Dubey et al. 1995], the superthreaded architecture [Tsai and Yew 1996; Li et al. 1996], the dynamic multithreading processor [Akkary and Driscoll 1998], and the speculative multithreaded processor [Marcuello et al. 1998]. Some approaches, in particular the multiscalar scheme, use compiler-support for thread generation. In these examples 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. As a result, some of these approaches may rather be viewed as very closely coupled chip multiprocessors, because multiple subordinate processing units execute different threads under control of a single sequencer unit.

1.2. Multithreading and Multiprocessors

The first multithreaded processor approaches in the 1970s and 1980s applied multithreading at user-thread-level to solve the memory access latency problem. This problem 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. The latency becomes a problem if the processor spends a large fraction of its time sitting idle and waiting for remote accesses to complete [Culler et al. 1998]. 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.

Older multithreaded processor approaches from the 1980s usually extend scalar RISC processors by a multithreading technique and focus on effectively bridging very long remote memory access latencies. Such processors will only be useful as processor nodes in distributed shared-memory multiprocessors. However, developing a processor that is specifically designed for such 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 the Cray MTA). Therefore, newer multithreaded processor approaches also strive for tolerating all latencies (even single-cycle latencies) that arise from primary cache misses that hit in secondary cache, from long-latency operations, or even from unpredictable branches.

1.3. Multithreading and Dataflow

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. According to Dennis and Gao [1994], 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. A thread is viewed as a sequentially ordered block of instructions with a grain-size greater than 1 (to distinguish multithreaded architectures from fine-grained dataflow architectures). *Blocking* and *nonblocking* threads are distinguished. A *nonblocking 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 nonblocking 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 is available. Thus a program is compiled into many, very small threads activating each other when data become available. The same hardware 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, cache misses, or synchronization needs. 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 nonblocking threads typically leads to many small threads that are appropriate for execution by a hybrid dataflow computer [Šilc et al. 1998] 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

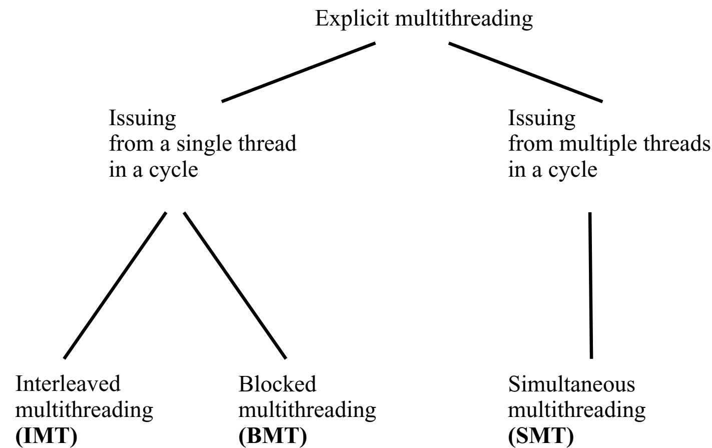

Fig. 1. Explicit multithreading.

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 nonblocking threads. Although these architectures are often called *multithreaded*, we have categorized them in a previous paper [Šilc et al. 1998] as threaded dataflow or large-grain dataflow because a dataflow principle is applied to start the execution of nonblocking threads. Thus, multithreaded architectures (in the more narrow sense applied here) stem from the modification of scalar RISC, VLIW/EPIC, or superscalar processors.

2. CLASSIFICATION OF EXPLICIT MULTITHREADING TECHNIQUES

Explicit multithreaded processors fall into two categories depending on whether they issue instructions from only a single thread or from multiple threads in a given cycle.

If instructions can be issued only from a single thread in a given cycle, the following two principal techniques of explicit multithreading are used (see Figure 1):

—*Interleaved multithreading* (IMT): An instruction of another thread is fetched and fed into the execution pipeline at each processor cycle (see Section 3).

—*Blocked multithreading* (BMT): The instructions of a thread are executed successively until an event occurs that may cause latency. This event induces a context switch (see Section 4).

When instructions can be issued from multiple threads in a given cycle, the following technique can be used:

—*Simultaneous multithreading* (SMT): Instructions are simultaneously issued from multiple threads to the execution units of a superscalar processor. Thus, the wide superscalar instruction issue is combined with the multiple-context approach (see Section 6).

Before we present these techniques in detail, we briefly review the main architectural approaches that integrate instruction-level parallelism and thread-level parallelism.

A 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 1 IPC, and thus the opportunity cost is the product of the latency

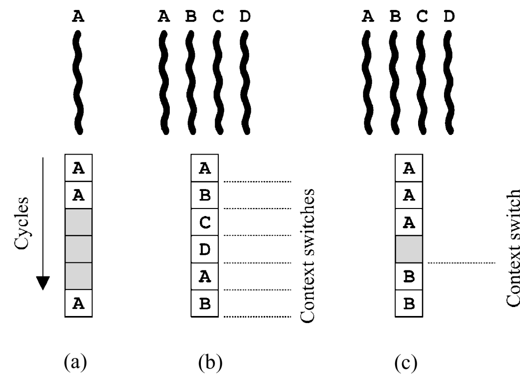

Fig. 2. Different approaches possible with single-issue (scalar) processors: (a) single-threaded scalar, (b) interleaved multithreading scalar, (c) blocked multithreading scalar.

cycles and the issue bandwidth plus the number of issue slots not fully filled. 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.

Figure 2 demonstrates the opportunity cost of different approaches possible with scalar processors, while the different approaches possible with multiple-issue processors are given in Figure 3. In all these cases instructions from only a single thread are issued in a given cycle.

The opportunity cost in single-threaded superscalar can be easily determined as the number of empty issue slots (see Figure 3(a)). It consists of horizontal losses (the number of empty places in not fully filled issue slot) and the even more harmful vertical losses (cycles where no instructions can be issued). In VLIW processors, horizontal losses appear as no-op operations (not shown in Figure 3). The opportunity cost of single-threaded VLIW is about the same as single-threaded superscalar. An IMT superscalar (or IMT VLIW) processor is able to fill the vertical losses of the single-threaded models by instructions of other threads, but not the horizontal losses. Further design possibilities, such as BMT superscalar or BMT VLIW processors, would fill several succeeding cycles with instructions of the same thread before context switching. The switching event is more difficult to implement and a context-switching overhead of one to several cycles might arise.

Let us now examine processors that can issue instructions from multiple threads in a given cycle. In addition to the SMT processors, one can include in this category (by the widest of definitions) also the chip multiprocessor (CMP) approach. While SMT uses a monolithic design with most resources shared among threads, CMP proposes a distributed design that uses a collection of independent processors with less resource sharing. For example, Figure 4a shows a four-threaded eight-issue SMT processor, and Figure 4b shows a CMP with four two-issue processors. The SMT processor 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 processor. If multiple threads each have low ILP, instructions of several threads can be issued and executed simultaneously. In the CMP processor with several multiple-issue processors (four two-issue processors in this case) on a single chip, each CPU is assigned a thread from which it can issue multiple instructions each cycle (up to two in this case). 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.

There are a number of tradeoffs to consider when choosing between SMT and CMP. The present report does not survey CMP design so these tradeoffs are outside its scope (but see Section 6 and the conclusions).

3. INTERLEAVED MULTITHREADING

3.1. Principles

The *interleaved multithreading* (IMT) technique (also called *fine-grain multithreading*) means that the processor

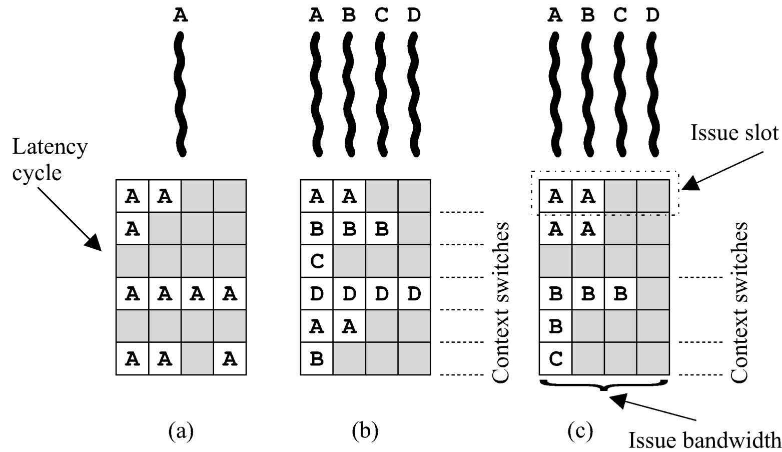

Fig. 3. Different approaches possible with multiple-issue processors: (a) single-threaded superscalar, (b) interleaved multithreading superscalar, (c) blocked multithreading superscalar.

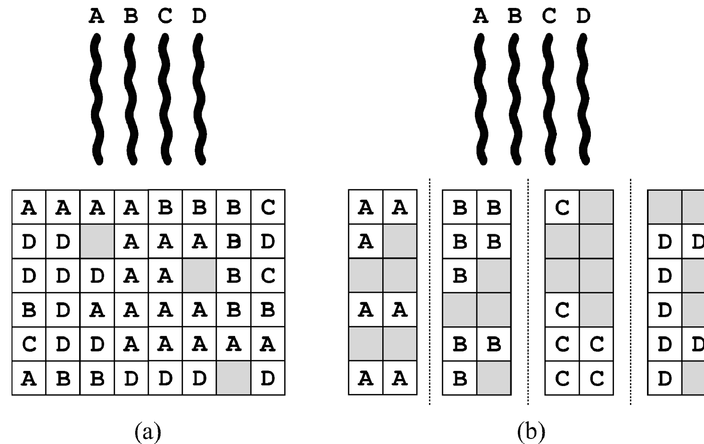

Fig. 4. Issuing from multiple threads in a cycle: (a) simultaneous multithreading, (b) chip multiprocessor.

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 IMT 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 or data forwarding 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. Two possibilities to overcome this deficiency are the following:

1. The *dependence look-ahead technique* 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 IMT processor to feed non-data- or control-dependent instructions of the same thread successively into the pipeline. The dependence look-ahead technique may be applied to speed up single-thread performance or to increase machine utilization in the case where a workload does not comprise enough threads.

2. The *interleaving technique* proposed by Laudon, Gupta, and Horowitz [1994], adds caching and full pipeline interlocks to the IMT technique. Contexts may be interleaved on a cycle-by-cycle basis, yet a single-thread context is also efficiently supported.

The most well-known examples of IMT processors are used in the Heterogeneous Element Processor (HEP), the Horizon, and the Cray Multi-Threaded Architecture (MTA) multiprocessors (see Section 5.1). The HEP system has up to 16 processors while the other two consist of up to 256 processors. Each of these processors supports up to 128 threads. While HEP uses instruction look-ahead only if there is no other work, the Horizon and Cray MTA employ the explicit dependence look-ahead technique. Further IMT processor proposals include the Multilisp Architecture for Symbolic Applications (MASA), the MIT M-Machine, the Media Processor of MicroUnity, and the processor SB-PRAM/HPP (all covered in the next section). An example of an IMT network processor is the five-threaded Lextra LX4580 (see Section 5.2).

In principle, the IMT technique can also be combined with a superscalar instruction issue, but simulations confirm the intuition that SMT is the more efficient technique [Eggers et al. 1997].

3.2. Examples of Past Commercial Machines and of Research Prototypes

HEP. The Heterogeneous Element Processor system, designed by Burton Smith and developed by Denelcor Inc., in Denver, CO, between 1978 and 1985, was a pioneering example of a multithreaded machine [Smith 1981]. The HEP system [Smith 1985] was designed to have up to 16 processors (Figure 5) with up to 128 threads per processor. The 128 threads were supported by a large number of registers dynamically allocated to threads. The processor pipeline had eight stages, matching the minimum number of processor cycles necessary to fetch a data item from memory to a register. Consequently 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. Allowing only a single memory instruction in the pipeline at a time resulted in poor single-thread performance and required a very large number of threads. If thread queues were all empty, the next instruction from the last thread dispatched was examined for independence from previous instructions and, if found to be so, the instruction was also issued.

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

MASA. The Multilisp Architecture for Symbolic Applications [Halstead and Fujita 1988] was an IMT processor proposal for parallel symbolic computation with various features intended for effective Multilisp [Halstead 1985] 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).

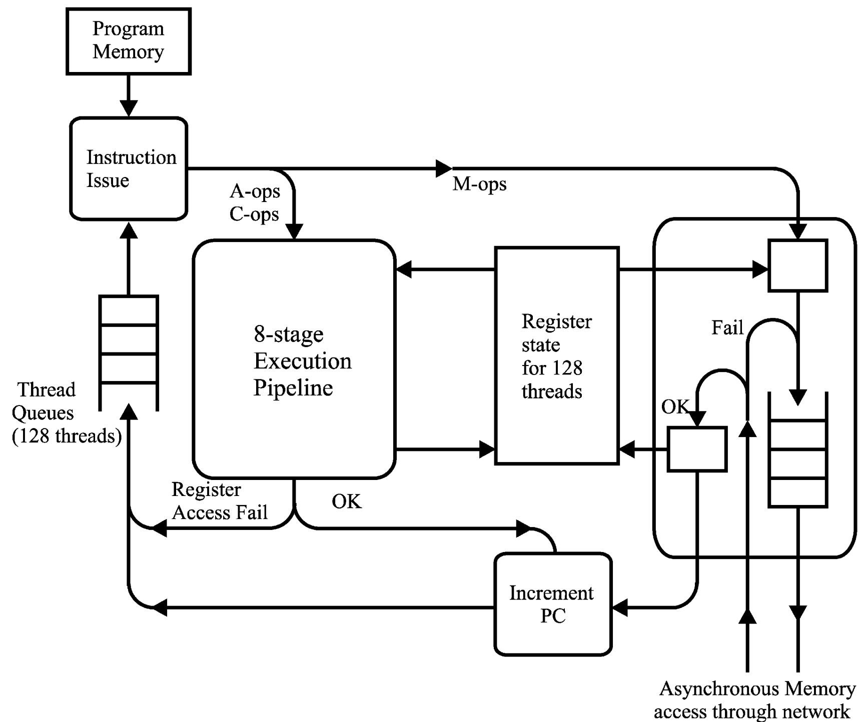

Fig. 5. The HEP processor.

M-Machine. The MIT M-Machine [Fillo et al. 1995] supports both public and private registers for each thread, and uses the IMT technique. 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 a processor-resident V-thread 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.

MicroUnity MediaProcessor. MicroUnity [Hansen 1996] proposed its MediaProcessor in 1996 as a processor specialized for video streaming applications. A superscalar processor kernel is enhanced by group instructions dedicated to video streaming applications and by five thread contexts. Instructions of the different threads are executed in the IMT fashion issuing instructions with a proposed 1-GHz clock frequency providing five independent 200-MHz threads. A very light-weight context switch upon synchronous exceptions and asynchronous events permits rapid handling of virtual memory system exceptions and I/O events.

SB-PRAM and HPP. The SB-PRAM (SB stands for *Saarbrücken* and PRAM for *parallel random access machine*) [Bach et al. 1997] or high-performance PRAM (HPP) [Formella et al. 1996] is a multiple instruction multiple data (MIMD) parallel computer with shared address space and uniform memory access time due to its motivation: building a multiprocessor that is

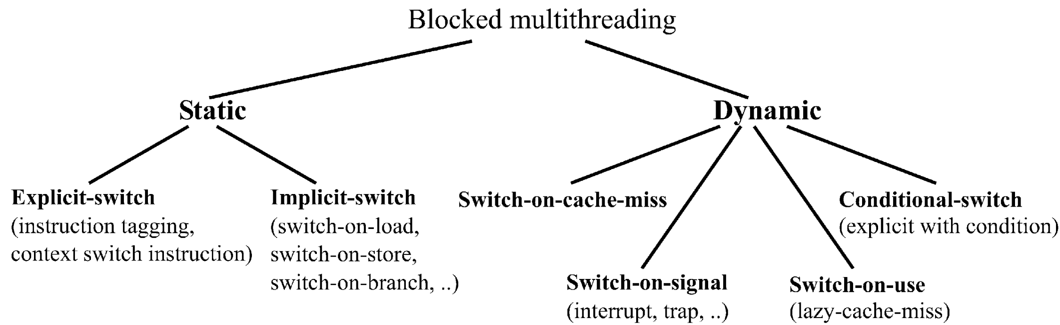

Fig. 6. Blocked multithreading.

as close as possible to the theoretical machine model CRCW-PRAM (CRCW stands for *concurrent read concurrent write*). Processor and memory modules are connected by a pipelined combining butterfly network. Network latency is hidden by pipelining several so-called virtual processors on one physical processor node in IMT mode. Instructions of 32 virtual processors are interleaved round-robin in a single SB-PRAM processor, which is therefore classified as 32-threaded IMT processor. The project is in progress at the University of Saarland, Saarbrücken, Germany. A first prototype was running with four processors, and recently a 64-processor model which in total supports up to 2048 threads has been completed. Hardware details and performance measurements are reported by Paul, Bach, Bosch, Fischer, Lichtenau, and Röhrig [2002].

4. BLOCKED MULTITHREADING

4.1. Principles

The *blocked multithreading* (BMT) technique (sometimes also called *coarse-grain multithreading*) means that a single thread is executed 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 IMT technique, a smaller number of threads is needed and a single thread can execute at full speed until the next context switch. If only a single thread runs on a BMT processor, there are no context switches and the processor performs just like a standard processor without multithreading.

In the following we classify BMT processors by the event that triggers a context switch (Figure 6) [Kreuzinger and Ungerer 1999]:

1. *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 BMT technique:
   - —The *explicit-switch* static model, where an explicit context switch instruction exists. This model is simple to implement. However, the additional instruction causes one-cycle context switching overhead.
   - —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 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 even are unpredictable.

2. *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, for example, 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 and will not occur at all if the load completes before the register is referenced.

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 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 [Boothe and Ranade 1992; Boothe 1993] concerns the BMT technique only in a shared-memory multiprocessor environment and is restricted to only a few of the models of the BMT technique described above. In particular, the switch-on-load in [Boothe and Ranade 1992] 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 BMT technique. The MIT Sparcle [Agarwal et al. 1993] and the MSparc processors [Mikschl and Damm 1996] use both switch-on-cache-miss and switch-on-signal dynamic models. The CHoPP 1 [Mankovic et al. 1987] uses the switch-on-cache-miss and switch-on-use dynamic models, while Rhamma [Gruenewald and Ungerer 1996, 1997] applies several static and dynamic models of BMT. The EVENTS mechanism and the Komodo microcontroller proposal apply multithreading for real-time event handling. These, as well as some other processors using the BMT technique, are described in the next section, while the current commercial high-performance processors—that is, the two-threaded IBM RS64 IV, the Sun MAJC—and the network processors of the Intel IXP, IBM Power NP, Vitesse IQ2x00, and AMCC nP families are covered in Section 5.

4.2. Examples of Past Commercial Machines and of Research Prototypes

CHoPP 1. The Columbia Homogeneous Parallel Processor 1 (CHoPP 1) [Mankovic et al. 1987] was designed by 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 issuing 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.

MDP in J-Machine. The MIT Jellybean Machine (J-Machine) [Dally et al. 1992] 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 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.

MIT Sparcle. The MIT Sparcle processor [Agarwal et al. 1993] is derived from a SPARC RISC processor. The eight overlapping register windows of a SPARC processor are organized as four independent nonoverlapping thread contexts, each using two windows, one as a register set, the other as a context for trap and message handlers (Figure 7).

Context switches are used only to hide long memory latencies since small pipeline delays are assumed to be hidden by the proper ordering of instructions by

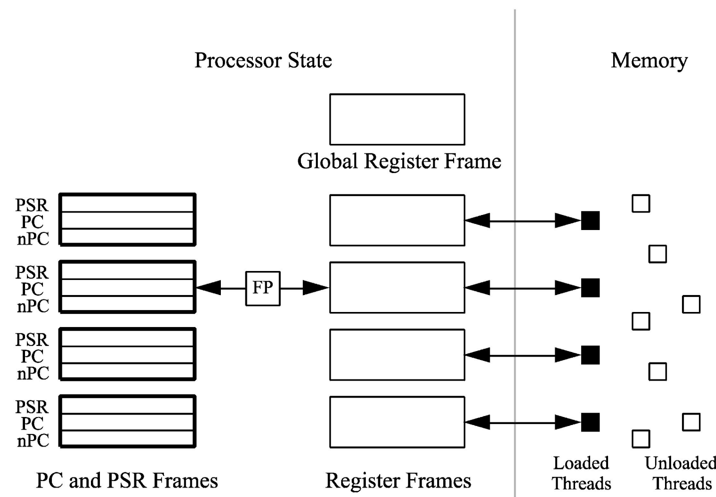

Fig. 7. Sparcle register usage.

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, that is, by the cache/directory controller. Reloading of the pipeline and the software implementation of the context switch requires 14 processor cycles.

The MIT Alewife distributed shared-memory multiprocessor [Agarwal et al. 1995] 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, cache, and a directory-based cache controller that manages cache-coherence, a network router chip, and a memory module (Figure 8).

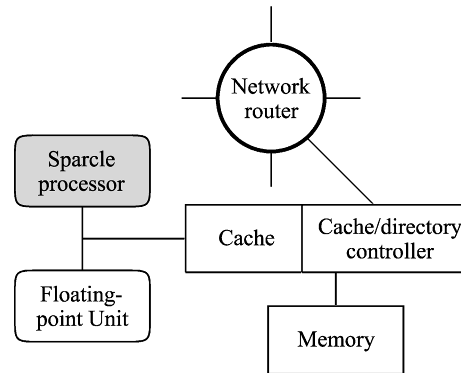

Fig. 8. An Alewife node.

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.

Rhamma. The Rhamma processor [Gruenewald and Ungerer 1996, 1997] was developed 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. 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 continuations, each denoting the thread tag and the instruction pointer

of a thread. The Rhamma processor combines several static and dynamic models of the BMT technique.

The EU of the Rhamma processor switches the thread whenever a load/store or control-flow instruction is fetched. Therefore, if the instruction format is chosen appropriately, a context switch can be recognized by *predecoding* the instructions already during the instruction fetch stage.

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.

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 instruction fetch 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.

PL/PS-Machine. The Preload and Poststore Machine (or PL/PS-Machine) [Kavi et al. 1997] 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 nonblocking 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.

The DanSoft nanothreading approach. The *nanothreading* and the *microthreading* approaches use multithreading but spare the hardware complexity of providing multiple register sets.

Nanothreading [Gwennap 1997], 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 program counter and 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, that 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 backup path executed in the nanothread generates a misprediction penalty of only one to two 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.

Microthreading. The *microthreading* technique [Bolychevsky et al. 1996] 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 BMT 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 in Jesshope and Luo [2000] and Jesshope [2001]. 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.

MSparc and the EVENTS mechanism. An approach similar to the MIT Sparcle processor was taken at the University of Oldenburg, Germany, with the MSparc processor [Mikschl and Damm 1996]. MSparc supports up to four contexts on a 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 BMT with the switch-on-cache-miss policy as in the MIT Sparcle processor.

The rapid context-switching ability of the multithreading technique leads to approaches that apply multithreading in new areas, in particular embedded real-time systems, which are proposed by the EVENTS approach and by the Komodo project.

The EVENTS scheduler [Lüth et al. 1997; Metzner and Niehaus 2000] acts as processor-external thread scheduler by supervising several MSparc processors. Contest switches are triggered due to real-time scheduling techniques and the computation-intensive real-time threads are assigned to the different MSparc processors. The EVENTS mechanism is implemented as a field of field programmable gate arrays (FPGAs).

Komodo microcontroller. The Komodo microcontroller [Brinkschulte et al. 1999a; 1999b] implements real-time scheduling algorithms on an instruction-by-instruction basis deeply embedded in the multithreaded processor core, in contrast to the processor-external EVENTS approach. The Komodo microcontroller is a multithreaded Java microcontroller aimed at embedded real-time systems with a hardware event handling mechanism that allows 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 reactions 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). The occurrence of an event activates an assigned IST instead of an ISR. The Komodo microcontroller supports the concurrent execution of multiple ISTs and zero-cycle overhead context switching, and triggers ISTs by a switch-on-signal context switching strategy.

As shown in Figure 9, 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

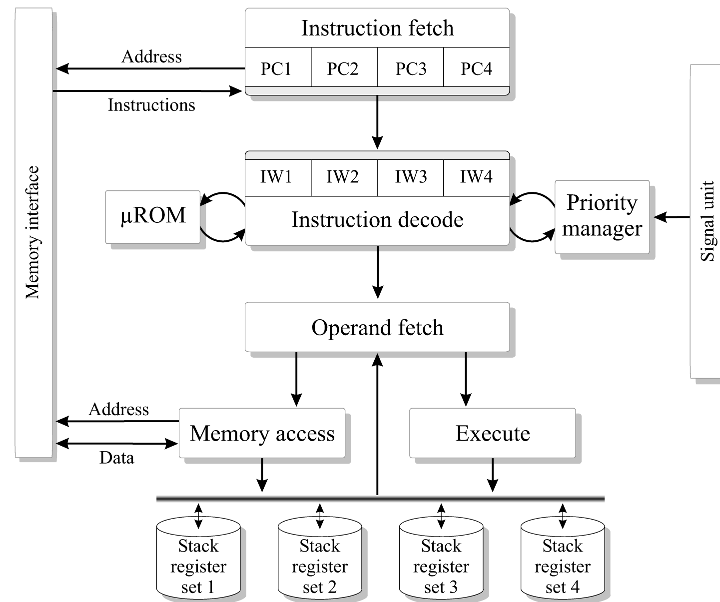

Fig. 9. Block diagram of the Komodo microcontroller.

micro-op or 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.

The instruction fetch unit holds four program counters (PCs) 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 appropriate 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, earliest deadline first, least laxity first, and guaranteed percentage (GP) scheduling are implemented in the priority manager for next instruction selection [Kreuzinger et al. 2000]. The GP scheduling scheme assigns portions of the processor execution power to each hardware thread and allows a strict isolation of the different real-time threads—an urgently demanded feature of real-time systems. GP scheduling is only efficient if implemented within a multithreaded processor [Brinkschulte et al. 2002].

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. By the occurrence of such a signal, the corresponding IST will be activated. 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 [Brinkschulte et al. 2000] have covered real-time garbage collection on a multithreaded microcontroller [Fuhrmann et al. 2001], and a distributed real-time middleware called OSA+ [Brinkschulte et al. 2000].

5. INTERLEAVED AND BLOCKED MULTITHREADING IN CURRENT AND PROPOSED MICROPROCESSORS

5.1. High-Performance Processors

Cray MTA. The Cray Multi-Threaded Architecture (MTA) computer [Alverson et al. 1990] features a powerful VLIW instruction set and interleaved multithreading technique. It was designed by the Tera Computer Company, Seattle, WA, now Cray Corporation.[1]

The MTA inherits much of the design philosophy of the HEP as well as a (paper) architecture called Horizon [Thistle and Smith 1988]. The latter was designed for up to 256 processors and up to 512 memory modules in a 16 × 16 × 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 allowed 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) (Figure 10). Each resource is individually connected to a separate routing node in the depopulated three-dimensional (3-D) torus interconnection network [Almasi and Gottlieb 1994]. Each routing node has three or four communication ports and a

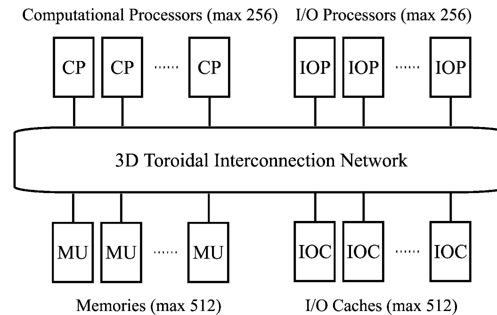

Fig. 10. The Cray MTA computer system.

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 the bisection bandwidth to scale linearly with p, while the network latency scales as $p^{1/2}$. 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 11) is a VLIW pipelined processor using the IMT technique. Each thread is associated with one 64-bit stream status word, thirty-two 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 128 of everything, that is, 128 stream status words, 4096 general registers, and 1024 target registers. Dependences between instructions are explicitly encoded by the compiler using *explicit dependence look-ahead*. Each instruction contains a 3-bit look-ahead 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 look-ahead value, at most eight instructions (i.e., 24 operations) from each

[1] Tera purchased Cray and adopted the name.

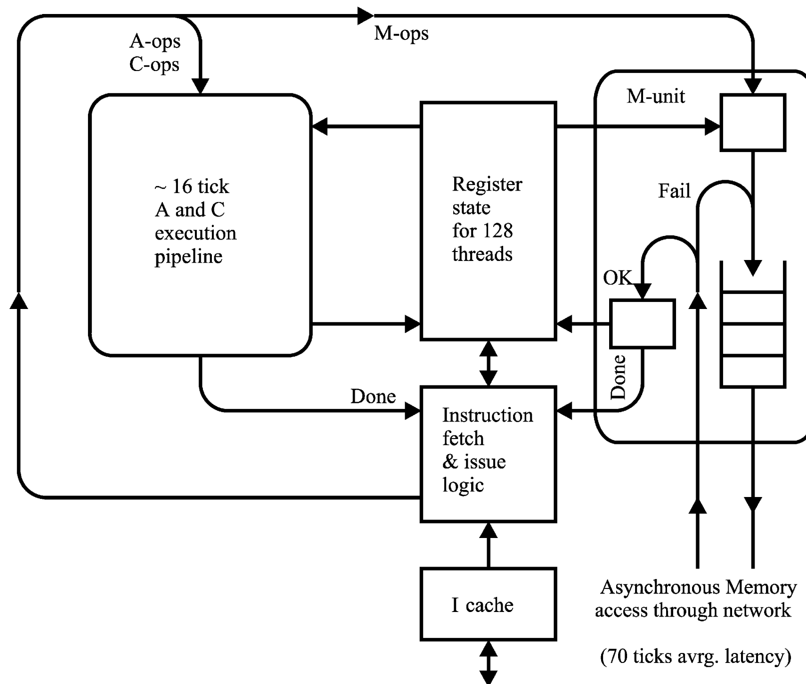

Fig. 11. The MTA computational processor.

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 three-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, the priority is M-op $\succ$ A-op $\succ$ C-op.

Every processor has a clock register that is synchronized 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 (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 [Alverson et al. 1995].

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. This machine was effectively retired in September 2001.

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. The latest in this development is the MTA-2, which is configured with 28 processors and 112 gigabytes (GB) of memory. The first Cray MTA-2 supercomputer system was shipped in late December 2001.

HTMT Architecture Project. 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), Argonne National Laboratories) initiated the *Hybrid Technology MultiThreaded* (HTMT) Architecture 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 [Sterling 1997]. There will be 4096 superconductive processors, called *SPELL*. 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 8 threads [Dorojevets 2000].

Sun MAJC. Sun proposed in 1999 its MAJC-5200 [Tremblay 1999] that can be classified as a dual-processor chip with BMT 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 single-instruction, multiple-data (SIMD) instructions in particular for multimedia applications. Thread-level parallelism is utilized through compiler-supported explicit multithreading. The architecture supports combining several multithreaded processors on a chip to harness process-level parallelism.

Single-threaded-program execution may be accelerated by speculative multithreading with microthreads that are dependent on nonspeculative threads [Tremblay et al. 2000]. Virtual channels communicate shared register values between different microthreads with produced-consumer synchronization. In case of misspeculation, the speculative microthread is discarded.

IBM RS64 IV. 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 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 so-called *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 Borkenhagen et al. [2000].

5.2. Network Processors

Currently multithreading proves successful in network processors. Network processors are expected to become the fundamental building blocks for networks in the same fashion that microprocessors are for today's personal computers [Glaskowsky 2002]. Network processors offer real-time processing of multiple data streams, enhanced security, and Internet protocol (IP) packet handling and forwarding capabilities.

Network processors apply multithreading to bridge latencies during memory accesses. Hard real-time events, that is, the deadline should never be missed, are requirements for network processors that need specific instruction scheduling during multithreaded execution. Multithreading is typically restricted to the internal architecture of the parallel working picoprocessors or microengines that perform the data traffic handling.

Intel IXP network processors. The Intel IXP1200 network processor [Intel Corporation 2002]—the first in the Intel Internet Exchange Architecture (Intel IXA) network processor family that uses multithreading—combines an Intel StrongARM processor core with six programmable multithreaded microengines. The IXP2400 network processor is based on an Intel XScale core with eight multithreaded microengines, and the IXP2800 contains even 16 multithreaded microengines [Intel Corporation 2002]. The microengines (four-threaded BMT) are used for packet forwarding and traffic management, whereas the XScale processor core is applied for control tasks. Data and event signals are shared among threads and microengines at virtual zero latency while maintaining coherence. The IXP2800 is claimed to be capable of enabling 10-Gb/s wire-speed processing (Gb = gigabits).

IBM PowerNP network processor. The IBM PowerNP NP46GS3 network processor [IBM Corporation 1999] integrates a switching engine, search engine, frame processors, and multiplexed medium access controllers (MACs). The NP4GS3 includes an embedded IBM PowerPC 405 processor core, 16 programmable multithreaded picoprocessors, 40 Fast Ethernet/4-Gb MACs, and hardware accelerators performing functions such as tree searches, frame forwarding, frame filtering, and frame alteration.

The multithreaded picoprocessors are used for packet manipulation while the processor core handles control functions. Each picoprocessor is a 32-bit, 133-MHz RISC core that supports two threads in the hardware and includes nine hardwired function units for tasks such as string copying, bandwidth-policy checking, and check summing.

The aggregated bandwidth of the NP4GS3 is some 4 Gb/s, allowing it to manage a single OC-48 optical channel or up to 40 Fast Ethernet ports [Glaskowsky 2002].

Vitesse IQ2x00 network processors. The Vitesse IQ2x00 family consists of IQ2000 and IQ2200 network processors, whose architecture is based on multiple packed processors. Each packed processor is a five-threaded BMT that switches on cache miss [Glaskowsky 2002].

Lexra network processors. The Lexra network processor employs packet processors, which are used in parallel to implement IP layer processing in software. The current packed processor is the LX4580 [Gelinas et al. 2002]. It implements the MIPS32 architecture, including recent architectural enhancements, along with instructions for optimized packet processing. By contrast with the Intel IXP and Vitesse IQ2x00, the LX4580 uses the IMT approach (with five threads).

AMCC nP network processors. AMCC's nP network processor family is based on the

MMC Networks' nPcore multithreaded packet processor. Each processor in this family consists of one or several nPcore processing units that are connected to an external search coprocessor and a host CPU [Glaskowsky 2002]. In particular, the nP7250 network processor has two nPcores with 2.4 Gb/s aggregated bandwidth. The current top of AMCC's line is the nP7510, whose six nPcore processing units offer 10 Gb/s.

Other network processors. A number of other network processors, such as the Motorola C-5, and the Fast Pattern Processor (as part of Agere's PayloadPlus chip set) seem to include multithreading features. Unfortunately, it is often hard to find out, because multithreading is not the main feature of network processors and often hardly mentioned. The market for network processors is extremely competitive, so the above-mentioned processors will soon be replaced by their successors.

6. SIMULTANEOUS MULTITHREADING

6.1. Principles

IMT and BMT are 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 all 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*. Therefore, not only can unused cycles in the case of latencies be filled by instructions of alternative threads, but so can unused issue slots within one cycle.

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 an SMT processor simultaneously exploits coarse- and fine-grain 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 tradeoff is a slightly more complex hardware organization.

If all (hardware-supported) threads of an SMT processor always execute threads of the same process, preferably in single-program multiple-data fashion, a unified (primary) I-cache may prove useful, since the code can be shared between the threads. Primary D-cache may be unified or separated between the threads depending on the access mechanism used.

The SMT technique combines a wide superscalar instruction issue with multithreading by providing several register sets on the processor and issuing instructions from several instruction queues 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 resources of SMT processors can be organized in two ways:

1. *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.

2. *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.

The fetch unit of an SMT processor 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 nonspeculative 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.

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 required by the on-chip wire-delay of future microprocessors is not easily achieved with an SMT processor due to the centralized instruction issue. A separation of the thread queues as in the SMT approaches with resource replication is a possible solution, although it does not remove the central instruction issue.

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. An enhanced form of predication is able to issue and execute predicated instruction even if the predicate is not yet solved. A further step is dynamic predication [Klauser et al. 1998a] as applied for the Polypath architecture [Klauser et al. 1998b] that is a superscalar enhanced to handle multiple threads internally. Another step to multithreading is simultaneous subordinate microthreading [Chappell et al. 1999], which is a modification of superscalars to run threads at microprogram level concurrently. A new microthread is spawned either by 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 preload data.

A competing approach to SMT is represented by the chip multiprocessors (CMPs) [Ungerer et al. 2002]. A CMP integrates two or more complete processors on a single chip. Every unit of a processer is duplicated and used independently of its copies on the chip. CMP is easier to implement, but only SMT has the ability to hide latencies. Examples of CMP are the Texas Instruments TMS320C8x [Texas Instruments 1994], the Hydra [Hammond and Olukotun 1998], the Compaq Piranha [Barroso et al. 2000], and the IBM POWER4 chip [Tendler et al. 2002].

Projects simulating different configurations of SMT are discussed next, while SMT approaches that can be found in current microprocessors are given in Section 7.

6.2. Examples of Prototypes and Simulated SMT Processors

MARS-M. The MARS-M multithreaded computer system [Dorozhevets and Wolcott 1992] was developed and manufactured within the Russian Next-Generation Computer Systems program during 1982–1988. The MARS-M was the first system where the SMT technique 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 SMT. 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 IMT 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 integration (MSI) and large-scale integration emitter-coupled logic (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. The MARS-M hardware is water-cooled and occupies three cabinets. The initial clock period of 100 ns was increased to 108 ns at the debugging stage.

Matsushita Media Research Laboratory processor. The multithreaded processor of the Media Research Laboratory of Matsushita Electric Ind. (Japan) was another pioneering approach to SMT [Hirata et al. 1992]. 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 of 5.79 in the case of two load/store units, can be achieved over a conventional RISC processor. However, caches or translation look-aside buffers are not simulated, nor is a branch prediction mechanism.

"Multistreamed superscalar" at the University of Santa Barbara. Serrano et al. [1994] and Yamamoto and Nemirovsky [1995] at the University of Santa Barbara, CA, extended the interleaved multithreading (then called *multistreaming*) technique to general-purpose superscalar processor architecture and presented an analytical model of multithreaded superscalar performance, backed up by simulation.

SMT processor at the University of Washington. The SMT processor architecture, proposed by Tullsen, Eggers, and Levy [1995] at the University of Washington, Seattle, WA, surveys enhancements of the Alpha 21164 processor. Simulations were conducted to evaluate processor configurations of an up to eight-threaded and eight-issue superscalar. This maximum configuration showed a throughput of 6.64 IPC due to multithreading using the SPEC92 benchmark suite and assuming a processor with 32 execution units (among them multiple load/store units). Descriptions and the results are based on the multiprogramming model.

The next approach was based on a hypothetical out-of-order instruction issue superscalar microprocessor that resembles the MIPS R10000 and HP PA-8000 [Tullsen et al. 1996; Eggers et al. 1997]. Both approaches followed the resource-sharing organization aiming at an only slight hardware enhancement of a superscalar processor. The latter 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 12), 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. 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 register file requires a longer access time. To avoid increasing the processor cycle time, the 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

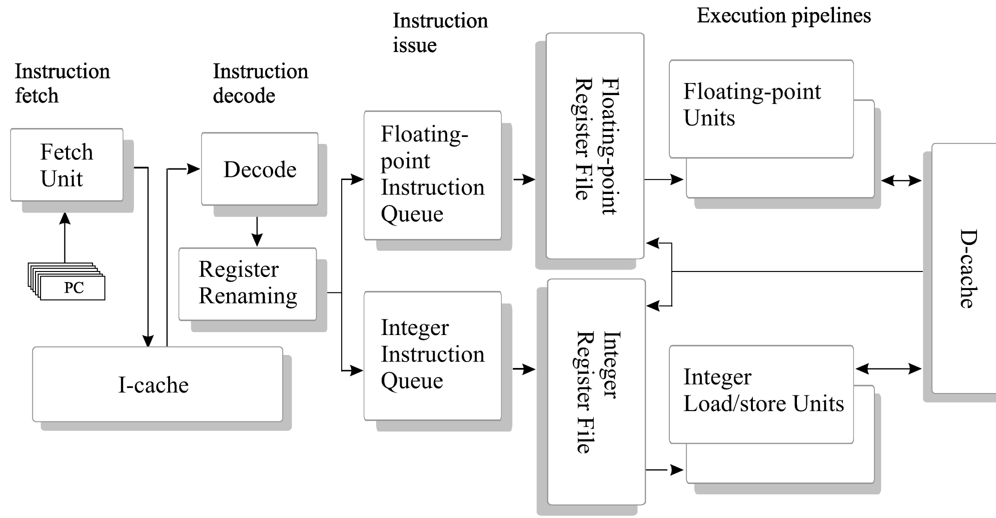

Fig. 12. SMT processor architecture.

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 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 evaluate different fetch and instruction issue strategies.

An RR.2.8 fetching scheme to access multiported I-cache—that is, in each cycle, eight instructions are fetched in round-robin policy from each of 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 the 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 the highest priority to those threads that are least likely to be on a wrong path, and the MISS-COUNT scheme, which gives priority to the threads that have the fewest outstanding D-cache misses. The IQPOSN policy that gives the 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 MISS-COUNT, 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 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, like oldest instructions first, speculative instructions last, and branches first. Issue bandwidth is not a bottleneck on these simulated machines 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 [Lo et al. 1997] and at extracting threads of a single program designed for multithreaded execution on an SMT [Tullsen et al. 1999]. The SMT processor has also been evaluated with database workloads [Lo et al. 1998] achieving roughly a three-fold increase in instruction throughput with an eight-threaded SMT over a single-threaded superscalar with similar resources.

Wallace et al. [1998] presented the *threaded multipath execution* model, which exploits existing hardware on an SMT processor to execute simultaneously alternate paths of a conditional branch in a thread. 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 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 four to eight 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. Wallace et al. [1999] explored instruction recycling on the proposed multipath SMT processor.

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 [Gulati and Bagherzadeh 1996; Loikkanen and Bagherzadeh 1996]. 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.

Pontius and Bagherzadeh [1999] evaluated a multithreaded superscalar processor model 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 iteration to several threads. The relatively low speedup that was reported for multithreading resulted from algorithmic restrictions and from the already high IPC in the single-threaded model. The latter was 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.

SMV processor. The Simultaneous Multithreaded Vector (SMV) architecture [Espasa and Valero 1997], 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 13 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. Inside the queues, the

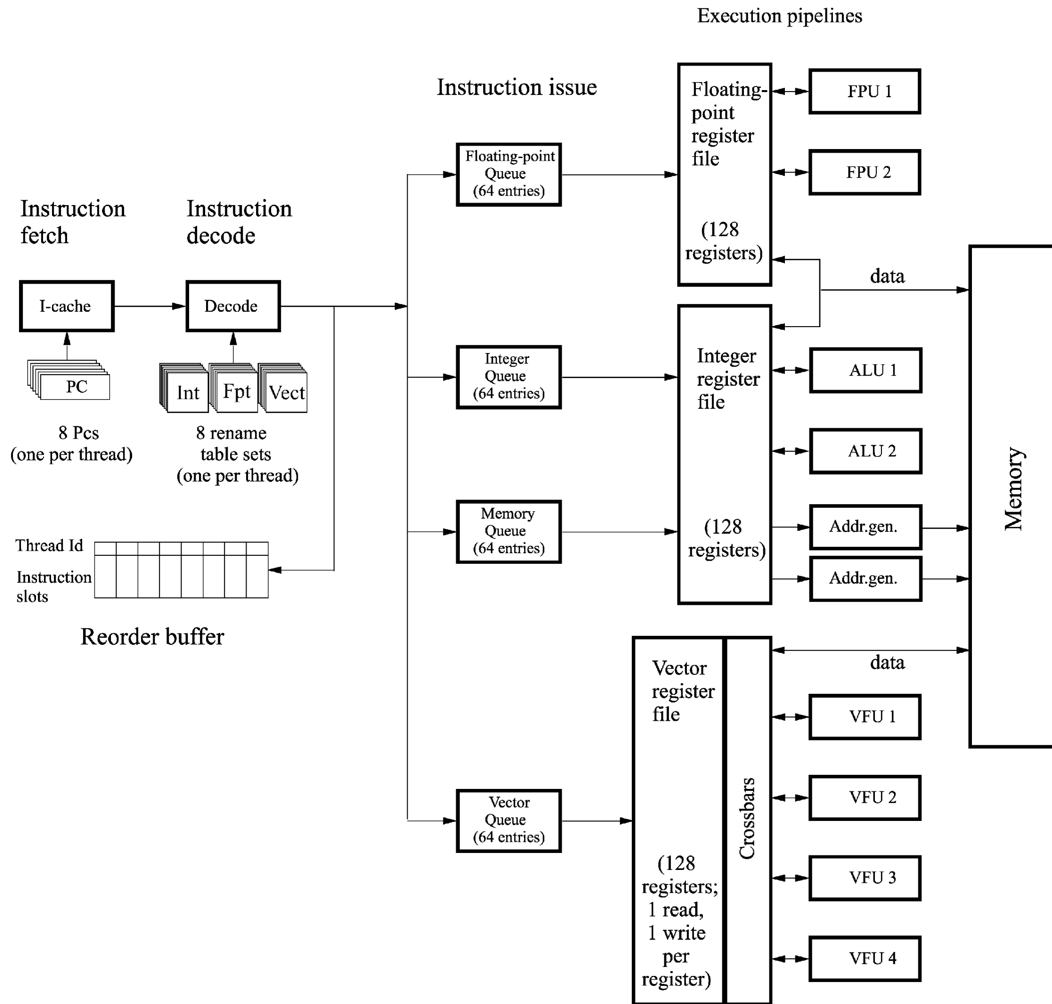

Fig. 13. Simultaneous Multithreaded Vector architecture. FPU = floating-point unit; ALU = arithmetic logic unit; VFU = vector functional unit.

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 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.

Karlsruhe multithreaded superscalar processor. While the SMT processor proposed by Tullsen, Eggers, and Levy [1995] surveys enhancements of the Alpha 21164 processor, the *multithreaded superscalar* processor approach of Sigmund and Ungerer [1996a, 1996b] is based on a simplified PowerPC 604 processor, combining multiple pipelines that were dedicated to different threads with a simultaneous issue unit (resource replication).

Using an instruction mix with 20% load and store instructions, the performance results showed, for an eight-issue processor with four to eight threads and

a single load/store unit, 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 proved the principal bottleneck. The multithreaded superscalar processor (eight-threaded, eight-issue) is able to hide completely latencies caused by 4-2-2-2 burst cache refills.[2] It reaches the maximum throughput of 4.2 IPC that is possible with a single load/store unit.

SMT multimedia processor. Subsequent SMT research at the University of Karlsruhe has explored resource replication-based microarchitecture models for an SMT processor with multimedia enhancements [Oehring et al. 1999a, 1999b]. Related research by Pontius and Bagherzadeh [1999] and Wittenburg et al. [1999] did not use multimedia operations. Hand-coding was applied to transfer a commercial MPEG-2 video decoding algorithm to the SMT multimedia processor model and to program the algorithm in multithreaded fashion. The research processor model assumes a wide-issue superscalar processor, and enhances it by the SMT technique, by multimedia units, and by an additional on-chip RAM storage.

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 the smaller reorder buffers, increased the IPC value for the multithreaded models. Intuition suggests 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 nonspeculative instructions or instructions with low speculation depth are fetched, decoded, issued, and executed in an SMT processor. An abundance of speculative instructions may decrease the performance of an 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 those 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 nonspeculative execution of store and thread control instructions.

Subsequent research by Sigmund et al. [2000] investigated a cost/benefit analysis of various SMT multimedia processor models. Transistor count and chip space estimations (applying the tool described by Steinhaus et al. [2001]) showed that the additional hardware cost of a four-threaded, eight-issue SMT processor over a single-threaded processor is a 2% increase in transistor count and a 9% increase in chip space for a 288 million transistor chip, which yields a threefold speedup. The small scaled four-threaded, eight-issue SMT models with only 24 million transistors require a 9% increase in transistor count and a 27% increase in chip space, resulting in a 1.5-fold speedup over the superscalar base model.

Similar results were reached by Burns and Gaudiot [2002], who estimated the layout area for SMT. They identified which layout blocks are affected by SMT, determined the scaling of chip space requirements, and compared SMT versus single-threaded processor space requirements by scaling a R10000-based layout to 0.18-micron technology.

Simultaneous multithreading for signal processors. Wittenburg et al. [1999] looked at the application of the SMT technique for 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.

[2] 4-2-2-2 assumes that four times 64-bit portions are transmitted over the memory bus, the first portion reaching the processor four cycles after the cache miss indication, the next portion two cycles later, etc.

Simultaneous multithreading and power consumption. Simultaneous multithreading can also be applied for reduction of power consumption. In principle, the same power reduction techniques are applicable for SMT processors as for superscalars. When an SMT processor is simultaneously executing threads, the per-cycle use of the processor resources should noticeably increase, offering less opportunities for power reduction via traditional techniques such as clock gating. However, an SMT processor, specifically designed to execute multiple threads in a power-aware manner, provides additional options for power-aware design [Brooks et al. 2000].

Mispredictions cost energy because the speculatively executed instructions must be squashed. Contemporary superscalar processors discard approximately 60% of the fetched and 30% of the executed instructions due to misspeculations. In case of a power-aware design, the issue slots of an SMT processor could be preferably filled by less speculative instructions of other threads. Harnessing the extra parallelism provided by multiple threads allows the processor to rely much less on speculation. Less resources are utilized by speculative instructions and more parallelism is exploited. Moreover, a simpler branch unit design could be used. This inherently implies a greater power efficiency per thread.

The simulations of Seng et al. [2000] showed that by using an appropriate scheduler up to 22% less power is consumed by an SMT processor than by a comparable superscalar.

7. SIMULTANEOUS MULTITHREADING IN CURRENT MICROPROCESSORS

Alpha 21464. Compaq unveiled its Alpha EV8 21464 proposal in 1999 [Emer 1999], a four-threaded eight-issue SMT processor that closely resembles the SMT processor proposed by Tullsen et al. [1999]. The processor proposal featured out-of-order execution, a large on-chip secondary cache, a direct RAMBUS interface, and an on-chip router for system interconnect of a directory based, cache-coherent NUMA (nonuniform memory access) 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.

Blue Gene. Simultaneous multithreading is mentioned as processor technique for the building block of the IBM Blue Gene system—a 5-year effort to build a petaflops supercomputer started in December 1999 [Allen et al. 2001].

Sun UltraSPARC V. Also the Sun UltraSPARC V processor has been announced to exploit thread-level parallelism by symmetric multithreading—that is, SMT. The ultraSPARC V will be able to switch between two different modes depending on the type of work—one mode for heavy duty calculations and the other for business transactions such as database operations [Lawson and Vance 2002].

Hyper-Threading Technology in the Intel Xeon processor. Intel's Hyper-Threading Technology [Marr et al. 2002] proposes SMT for the Pentium 4-based Intel Xeon processor family to be used in dual and multiprocessor servers. Hyper-Threading Technology makes a single physical processor appear as two logical processors by applying a two-threaded SMT approach. Each logical processor maintains a complete set of the architecture state, which consists of the general-purpose registers, the control registers, the *advanced programmable interrupt controller* (APIC) registers, and some machine state registers. Logical processors share nearly all other resources on the physical processor, such as caches, execution units, branch predictors, control logic, and buses. Each logical processor has its own APIC. Interrupts sent to a specific logical processor are handled only by that logical processor.

When one logical processor is stalled, the other logical processor can continue to make forward progress. A logical processor may be temporarily stalled for a variety of reasons, including servicing cache misses, handling branch mispredictions, or waiting for the results of previous instructions.

Independent forward progress was ensured by managing buffering queues such that no logical processor can use all the entries when two active software threads were executing.

The buffering queues that separate major pipeline logic blocks are either partitioned or duplicated to ensure independent forward progress through each logic block. If only one active software thread is active, the thread should run at the same speed on a processor with Hyper-Threading Technology as on a processor without this capability. This means that partitioned resources should be recombined when only one software thread is active, thus applying a flexible resource-sharing model.

In the Xeon, as in the Pentium 4, instructions generally come from the execution *trace cache* (TC), which replaces the primary I-cache. Two sets of instruction pointers independently track the progress of the two executing software threads. The TC entries are tagged with thread information. The two logical processors arbitrate access to the TC every clock cycle. If both logical processors want access to the TC at the same time, access is granted to one then the other in alternating clock cycles. If one logical processor is stalled or is unable to use the TC, the other logical processor can use the full bandwidth of the TC, every cycle.

If there is a TC miss, instruction bytes need to be fetched from the secondary cache and decoded into μops (microoperations) to be placed in the TC. Each logical processor has its own instruction translation look-aside buffer and its own set of instruction pointers to track the progress of instruction fetch for the two logical processors. The instruction fetch logic in charge of sending requests to the secondary cache arbitrates on a first-come first-served basis, while always reserving at least one request slot for each logical processor. In this way, both logical processors can have fetches pending simultaneously. Each logical processor has its own set of two 64-byte streaming buffers to hold instruction bytes in preparation for the instruction decode stage.

The branch prediction structures are either duplicated or shared. The branch history buffer used to look up the global history array is also tracked independently for each logical processor. However, the large global history array is a shared structure with entries that are tagged with a logical processor ID. The return stack buffer, which predicts the target of return instructions, is duplicated.

When both threads are decoding instructions simultaneously, the streaming buffers alternate between threads so that both threads share the same decoder logic. The decode logic has to keep two copies of all the state needed to decode IA-32 instructions for the two logical processors even though it only decodes instructions for one logical processor at a time. In general, several instructions are decoded for one logical processor before switching to the other logical processor.

The μop queue that decouples the front-end from the out-of-order execution engine is partitioned such that each logical processor has half the entries. The allocator logic takes μops from the μop queue and allocates many of the key machine buffers needed to execute each μop, including the 126 reorder buffer entries, 128 integer and 128 floating-point physical registers, and 48 load and 24 store buffer entries. If there are μops for both logical processors in the μop queue, the allocator will alternate selecting μops from the logical processors every clock cycle to assign resources. Each logical processor can use up to a maximum of 63 reorder buffer entries, 24 load buffers, and 12 store buffer entries.

Since each logical processor must maintain and track its own complete architecture state, there are two register alias tables for register renaming, one for each logical processor. The register renaming process is done in parallel to the allocator logic described above, so the register rename logic works on the same μops to which the allocator is assigning resources.

Once μops have completed the allocation and register rename processes, they are placed into the memory instruction queue or the general instruction queue, respectively. The two sets of queues are

also partitioned such that μops from each logical processor can use at most half the entries. The memory instruction queue and general instruction queues send μops to the five scheduler queues as fast as they can, alternating between μops for the two logical processors every clock cycle, as needed.

Each scheduler has its own scheduler queue of 8 to 12 entries from which it selects μops to send to the execution units. The schedulers choose μops regardless of whether they belong to one logical processor or the other. The schedulers are effectively oblivious to logical processor distinctions. The μops are simply evaluated based on dependent inputs and availability of execution resources. For example, the schedulers could dispatch two μops from one logical processor and two μops from the other logical processor in the same clock cycle. To avoid deadlock and ensure fairness, there is a limit on the number of active entries that a logical processor can have in each scheduler's queue.

The execution core and memory hierarchy are also largely oblivious to logical processors. After execution, the μops are placed in the reorder buffer, which decouples the execution stage from the retirement stage. The reorder buffer is partitioned such that each logical processor can use half the entries.

The retirement logic tracks when μops from the two logical processors are ready to be retired, then retires the μops in program order for each logical processor by alternating between the two logical processors. Retirement logic will retire μops for one logical processor, then the other, alternating back and forth. If one logical processor is not ready to retire any μops then all retirement bandwidth is dedicated to the other logical processor.

The implementation of Hyper-Threading Technology in the Xeon processor adds less than 5% to the relative chip size and maximum power requirements, but can provide performance benefits much greater than that. Initial benchmark tests show up to a 65% performance increase on high-end server applications when comparing the Xeon processor to the previous-generation PentiumIII Xeon processor on four-way server platforms. A significant portion of those gains can be attributed to Hyper-Threading Technology [Marr et al. 2002].

8. CONCLUSIONS

Although the meaning of the term *multithreading* is sometimes used to include all kinds of architectures that are able to concurrently execute multiple instruction streams on a single chip [Sohi 2001], that is, chip multiprocessors, explicit and implicit multithreaded architectures, we clearly distinguish these architectural solutions and focus this survey on explicit multithreaded processors.

Explicit multithreaded processors interleave the execution of instructions of different user-defined threads within the same pipeline, in contrast to *implicit multithreaded processors* that dynamically generate threads from single-threaded programs and execute such speculative threads concurrently with the lead thread.

Superscalar and implicit multithreaded processors aim at a low execution time of a single program, while explicit multithreaded processors (and chip multiprocessors) aim at a low execution time of a multithreaded workload.

Several explicit multithreaded processors as well as chip multiprocessors have currently been announced by industry or are already into production in the areas of high-performance microprocessors, media, and network processors. The basic interleaved and blocked multithreading techniques are applied in VLIW processors, in the network processors, and even in superscalar processors. In particular the success of these techniques in network processors is stunning. In addition, simultaneous multithreading processors and chip multiprocessors have been announced or are already in production by IBM, Intel, and Sun. The additional chip space requirement by a two-threaded processor is reported to be about 5% compared to a single-threaded processor (IBM RS64 IV and Intel Xeon processor), while a

significant throughput increase is reached by multithreading. We therefore expect explicit multithreading techniques, in particular the simultaneous multithreading techniques, to be commonly used in the next generation of high-performance microprocessors.

In contrast, implicit multithreaded processors and the related helper thread approach remain hot research topics and must still prove their efficiency in real processors. We expect that future research will also focus on the deployment of multithreading techniques in the fields of signal processors and microcontrollers, in particular for real-time applications, and for power management. Moreover, instruction scheduling within a multithreaded pipeline as well as system software architectures for multithreaded processors are still open research questions.

This survey should clarify the terminology and demonstrate the state-of-the-art as well as research results achieved for explicit multithreaded processors.

ACKNOWLEDGMENTS

The authors would like to thank anonymous reviewers for many valuable comments.

REFERENCES

AGARWAL, A., BIANCHINI, R., CHAIKEN, D., JOHNSON, K. L., KRANZ, D., KUBIATOWICZ, J., LIM, B. H., MACKENZIE, K., AND YEUNG, D. 1995. The MIT Alewife machine: architecture and performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* (Santa Margherita Ligure, Italy). 2–13.

AGARWAL, A., KUBIATOWICZ, J., KRANZ, D., LIM, B. H., YEOUNG, D., D'SOUZA, G., AND PARKIN, M. 1993. Sparcle: an evolutionary processor design for large-scale multiprocessors. *IEEE Micro 13*, 3, 48–61.

AKKARY, H. AND DRISCOLL, M. A. 1998. A dynamic multithreading processor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture* (Dallas, TX). 226–236.

ALLEN, F., ALMASI, G., ANDREONI, W., BEECE, D., BERNE, B. J., BRIGHT, A., BRUNHEROTO, J., CASCAVAL, C., CASTANOS, J., COTEUS, P., CRUMLEY, P., CURIONI, A., DENNEAU, M., DONATH, W., ELEFTHERIOU, M., FITCH, B., FLEISCHER, B., GEORGIOU, C. J., GERMAIN, R., GIAMPAPA, M., GRESH, D., GUPTA, M., HARING, R., HO, H., HOCHSCHILD, P., HUMMEL, S., JONAS, T., LIEBER, D., MARTYNA, G., MATURU, K., MOREIRA, J., NEWNS, D., NEWTON, M., PHILHOWER, R., PICUNKO, T., PITERA, J., PITMAN, M., RAND, R., ROYYURU, A., SALAPURA, V., SANOMIYA, A., SHAH, R., SHAM, Y., SINGH, S., SNIR, M., SUITS, F., SWETZ, R., SWOPE, W. C., VISHNUMURTHY, N., WARD, T. C. J., WARREN, H., AND ZHOU, R. 2001. Blue Gene: a vision for protein science using a petaflops supercomputer. *IBM Syst. J. 40*, 2, 310–326.

ALMASI, G. S. AND GOTTLIEB, A. 1994. *Highly Parallel Computing*, 2nd ed. Benjamin/Cummings, Menlo Park, CA.

ALVERSON, G., KAHAN, S., KORRY, R., MCCANN, C., AND SMITH, B. J. 1995. Scheduling on the Tera MTA. In Lecture Notes in Computer Science, vol. 949. Springer-Verlag, Heidelberg, Germany. 19–44.

ALVERSON, R., CALLAHAN, D., CUMMINGS, D., KOBLENZ, B., PORTERFIELD, A., AND SMITH, B. J. 1990. The Tera computer system. In *Proceedings of the 4th International Conference on Supercomputing* (Amsterdam, The Netherlands). 1–6.

BACH, P., BRAUN, M., FORMELLA, A., FRIEDRICH, J., GRÜN, T., AND LINCHTENAU, C. 1997. Building the 4 processor SB-PRAM prototype. In *Proceedings of the 30th Hawaii International Conference on System Science* (Maui, HI). 5:14–23.

BARROSO, L. A., GHARACHORLOO, K., MCNAMARA, R., NOWATZYK, A., QADEER, S., SANO, B., SMITH, S., STETS, R., AND VERGHESE, B. 2000. Piranha: a scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture* (Vancouver, B.C., Canada). 282–293.

BOLYCHEVSKY, A., JESSHOPE, C. R., AND MUCHNIK, V. B. 1996. Dynamic scheduling in RISC architectures. *IEE P. Comput. Dig. Tech. 143*, 5, 309–317.

BOOTHE, R. F. 1993. Evaluation of multithreading and caching in large shared memory parallel computers. Tech. Rep. UCB/CSD-93-766. Computer Science Division, University of California, Berkeley, Berkeley, CA.

BOOTHE, R. F. AND RANADE, A. 1992. Improved multithreading techniques for hiding communication latency in multiprocessors. In *Proceedings of the 19th International Symposium on Computer Architecture* (Gold Coast, Australia). 214–223.

BORKENHAGEN, J. M., EICKEMEYER, R. J., KALLA, R. N., AND KUNKEL, S. R. 2000. A multithreaded PowerPC processor for commercial servers. *IBM J. Res. Dev. 44*, 6, 885–898.

BRINKSCHULTE, U., BECHINA, A., PICIOROAGA, F., SCHNEIDER, E., UNGERER, T., KREUZINGER, J., AND PFEFFER, M. 2000. A microkernel middleware architecture for distributed embedded real-time systems. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems* (New Orleans LA). 218–226.

BRINKSCHULTE, U., KRAKOWSKI, C., KREUZINGER, J., AND UNGERER, T. 1999a. A multithreaded Java microcontroller for thread-oriented real-time event-handling. In *Proceedings of the*

International Conference on Parallel Architectures and Compilation Techniques (Newport Beach, CA). 34–39.

BRINKSCHULTE, U., KRAKOWSKI, C., MARSTON, R., KREUZINGER, J., AND UNGERER, T. 1999b. The Komodo project: thread-based event handling supported by a multithreaded Java microcontroller. In *Proceedings of the 25th Euromicro Conference* (Milan, Italy). 122–128.

BRINKSCHULTE, U., KREUZINGER, J., PFEFFER, M., AND UNGERER, T. 2002. A scheduling technique providing a strict isolation of real-time threads. In *Proceedings of the 7th IEEE International Workshop on Object-oriented Real-time Dependable Systems* (San Diego, CA). 169–172.

BROOKS, D. M., BOSE, P., SCHUSTER, S. E., JACOBSON, H., KUDVA, P. N., BUYUKTOSUNOGLU, A., WELLMAN, J. D., ZYUBAN, V., GUPTA, M., AND COOK, P. W. 2000. Power-aware microarchitecture: designing and modeling challenges for next-generation microprocessors. *IEEE Micro 20*, 6, 26–44.

BURNS, J. AND GAUDIOT, J. L. 2002. SMT layout overhead and scalability. *IEEE T. Parall. Distr. Syst. 13*, 2, 142–155.

BUTLER, M., YEH, T. Y., PATT, Y. N., ALSUP, M., SCALES, H., AND SHEBANOW, M. 1991. Single instruction stream parallelism is greater than two. In *Proceedings of the 18th International Symposium on Computer Architecture* (Toronto, Ont., Canada). 276–286.

CHAPPELL, R. S., STARK, J., KIM, S. P., REINHARDT, S. K., AND PATT, Y. N. 1999. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the 26th Annual International Symposium on Computer Architecture* (Atlanta, GA). 186–195.

CHRYSOS, G. Z. AND EMER, J. S. 1998. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* (Barcelona, Spain). 142–153.

CULLER, D. E., SINGH, J. P., AND GUPTA, A. 1998. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, San Francisco, CA.

DALLY, W. J., FISKE, J., KEEN, J., LETHIN, R., NOAKES, M., NUTH, P., DAVISON, R., AND FYLER, G. 1992. The message-driven processor: a multicomputer processing node with efficient mechanisms. *IEEE Micro 12*, 2, 23–39.

DENNIS, J. B. AND GAO, G. R. 1994. Multithreaded architectures: principles, projects, and issues. In *Multithreaded Computer Architecture: A Summary of the State of the Art*, R. A. Iannucci, G. R. Gao, R. Halstead, and B. J. Smith, Eds. Kluwer Boston, MA, Dordrecht, The Netherlands, London, U.K. 1–74.

DOROJEVETS, M. 2000. COOL multithreading in HTMT SPELL-1 processors. *Int. J. High Speed Electron. Sys. 10*, 1, 247–253.

DOROZHEVETS, M. N. AND WOLCOTT, P. 1992. The El'brus-3 and MARS-M: recent advances in Russian high-performance computing. *J. Supercomput. 6*, 1, 5–48.

DUBEY, P. K., O'BRIEN, K., O'BRIEN, K. M., AND BARTON, C. 1995. Single-program speculative multithreading (SPSM) architecture: compiler-assisted fine-grain multithreading. Tech. Rep. RC 19928. IBM, Yorktown Heights, NY.

EGGERS, S. J., EMER, J. S., LEVY, H. M., LO, J. L., STAMM, R. M., AND TULLSEN, D. M. 1997. Simultaneous multithreading: a platform for next-generation processors. *IEEE Micro 17*, 5, 12–19.

EMER, J. S. 1999. Simultaneous multithreading: multiplying Alpha's performance. In *Proceedings of the Microprocessor Forum* (San Jose, CA).

ESPASA, R. AND VALERO, M. 1997. Exploiting instruction- and data-level parallelism. *IEEE Micro 17*, 5, 20–27.

FILLO, M., KECKLER, S. W., DALLY, W. J., CARTER, N. P., CHANG, A., AND GUREVICH, Y. 1995. The M-machine multicomputer. In *Proceedings of the 28th Annual International Symposium on Microarchitecture* (Ann Arbor, MI). 146–156.

FORMELLA, A., KELLER, J., AND WALLE, T. 1996. HPP: A high performance PRAM. In Lecture Notes in Computer Science, vol. 1123. Springer-Verlag, Heidelberg, Germany. 425–434.

FRANKLIN, M. 1993. The multiscalar architecture. Tech. Rep. 1196. Department of Computer Science, University of Wisconsin-Madison, Madison, WI.

FUHRMANN, S., PFEFFER, M., KREUZINGER, J., UNGERER, T., AND BRINKSCHULTE, U. 2001. Real-time garbage collection for a multithreaded Java microcontroller. In *Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* (Magdeburg, Germany). 69–76.

GELINAS, B., HAYS, P., AND KATZMAN, S. 2002. Fine-grained hardware multi-threading: A CPU architecture for high-touch packed processing. Lexra Inc., Waltham, MA. White paper.

GLASKOWSKY, P. N. 2002. Network processors mature in 2001. *Microproc. Report*. February 19, 2002 (online journal).

GRÜNEWALD, W. AND UNGERER, T. 1996. Towards extremely fast context switching in a blockmultithreaded processor. In *Proceedings of the 22nd Euromicro Conference* (Prague, Czech Republic). 592–599.

GRÜNEWALD, W. AND UNGERER, T. 1997. A multithreaded processor designed for distributed shared memory systems. In *Proceedings of the International Conference on Advances in Parallel and Distributed Computing* (Shanghai, China). 206–213.

GULATI, M. AND BAGHERZADEH, N. 1996. Performance study of a multithreaded superscalar microprocessor. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture* (San Jose, CA). 291–301.

GWENNAP, L. 1997. DanSoft develops VLIW design. *Microproc. Report 11*, 2 (Feb. 17), 18–22.

HALSTEAD, R. H. 1985. MULTILISP: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7*, 4, 501–538.

HALSTEAD, R. H. AND FUJITA, T. 1988. MASA: a multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th International Symposium on Computer Architecture* (Honolulu, HI). 443–451.

HAMMOND, L. AND OLUKOTUN, K. 1998. Considerations in the design of Hydra: a multiprocessor-on-chip microarchitecture. Tech. Rep. CSL-TR-98-749. Computer Systems Laboratory, Stanford University, Stanford, CA.

HANSEN, C. 1996. MicroUnity's MediaProcessor architecture. *IEEE Micro 16*, 4, 34–41.

HIRATA, H., KIMURA, K., NAGAMINE, S., MOCHIZUKI, Y., NISHIMURA, A., NAKASE, Y., AND NISHIZAWA, T. 1992. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proceedings of the 19th International Symposium on Computer Architecture* (Gold Coast, Australia). 136–145.

IANNUCCI, R. A., GAO, G. R., HALSTEAD, R., AND SMITH, B. J., Eds. 1994. *Multithreaded Computer Architecture: A Summary of the State of the Art*. Kluwer Boston, MA, Dordrecht, The Netherlands, London, U.K.

IBM CORPORATION. 1999. IBM network processor. Product overview. IBM, Yorktown Heights, NY.

INTEL CORPORATION. 2002. Intel Internet exchange architecture network processors: flexible, wire-speed processing from the customer premises to the network core. White paper. Intel, Santa Clara, CA.

JESSHOPE, C. R. 2001. Implementing an efficient vector instruction set in a chip multi-processor using micro-threaded pipelines. *Aust. Comput. Sci. Commun. 23*, 4, 80–88.

JESSHOPE, C. R. AND LUO, B. 2000. Micro-threading: a new approach to future RISC. In *Proceedings of the Australasian Computer Architecture Conference* (Canbera, Australia). 34–41.

KAVI, K. M., LEVINE, D. L., AND HURSON, A. R. 1997. A non-blocking multithreaded architecture. In *Proceedings of the 5th International Conference on Advanced Computing* (Madras, India). 171–177.

KLAUSER, A., AUSTIN, T., GRUNWALD, D., AND CALDER, B. 1998a. Dynamic hammock predication for non-predicated instruction sets. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Paris, France). 278–285.

KLAUSER, A., PAITHANKAR, A., AND GRUNWALD, D. 1998b. Selective eager execution on the PolyPath architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* (Barcelona, Spain). 250–259.

KREUZINGER, J., SCHULZ, A., PFEFFER, M., UNGERER, T., BRINKSCHULTE, U., AND KRAKOWSKI, C. 2000. Real-time scheduling on multithreaded processors. In *Proceedings of the 7th International Conference on Real-Time Computer Systems and Applications* (Cheju Island, South Korea). 155–159.

KREUZINGER, J. AND UNGERER, T. 1999. Context-switching techniques for decoupled multithreaded processors. In *Proceedings of the 25th Euromicro Conference* (Milan, Italy). 1:248–251.

LAM, M. S. AND WILSON, R. P. 1992. Limits of control flow on parallelism. In *Proceedings of the 18th International Symposium on Computer Architecture* (Toronto, Ont., Canada). 46–57.

LAUDON, J., GUPTA, A., AND HOROWITZ, M. 1994. Interleaving: a multithreading technique targeting multiprocessors and workstations. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA). 308–318.

LAWSON, S. AND VANCE, A. 2002. Sun hints at UltraSparc V and beyond. Available online at PC World.com.

LI, Z., TSAI, J. Y., WANG, X., YEW, P. C., AND ZHENG, B. 1996. Compiler techniques for concurrent multithreading with hardware speculation support. In Lecture Notes in Computer Science, vol. 1239. Springer-Verlag, Heidelberg, Germany. 175–191.

LIPASTI, M. H. AND SHEN, J. P. 1997. The performance potential of value and dependence prediction. In Lecture Notes Computer Science, vol. 1300. Springer-Verlag, Heidelberg, Germany. 1043–1052.

LIPASTI, M. H., WILKERSON, C. B., AND SHEN, J. P. 1996. Value locality and load value prediction. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA). 138–147.

LO, J. L., BARROSO, L. A., EGGERS, S. J., GHARACHORLOO, K., LEVY, H. M., AND PAREKH, S. S. 1998. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* (Barcelona, Spain). 39–50.

LO, J. L., EGGERS, S. J., EMER, J. S., LEVY, H. M., STAMM, R. L., AND TULLSEN, D. M. 1997. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Trans. Comput. Syst. 15*, 3, 322–354.

LOIKKANEN, M. AND BAGHERZADEH, N. 1996. A fine-grain multithreading superscalar architecture. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Boston, MA). 163–168.

LÜTH, K., METZNER, A., PIEKENKAMP, T., AND RISU, J. 1997. The events approach to rapid prototyping for embedded control system. In *Proceedings*

of the Workshop Zielarchitekturen eingebetteter Syststeme (Rostock, Germany). 45–54.

MANKOVIC, T. E., POPESCU, V., AND SULLIVAN, H. 1987. CHoPP priciples of operations. In *Proceedings of the 2nd International Supercomputer Conference* (Mannheim, Germany). 2–10.

MARCUELLO, P., GONZALES, A., AND TUBELLA, J. 1998. Speculative multithreaded processors. In *Proceedings of the 12th International Conference on Supercomputing* (Melbourne, Australia). 77–84.

MARR, D. T., BINNS, F., HILL, D. L., HINTON, G., KOUFATY, D. A., MILLER, J. A., AND UPTON, M. 2002. Hyper-threading technology architecture and microarchitecture: a hypertext history. *Intel Technology J. 6*, 1 (online journal).

METZNER, A. AND NIEHAUS, J. 2000. MSparc: multithreading in real-time architectures. *J. Universal Comput. Sci. 6*, 10, 1034–1051.

MIKSCHL, A. AND DAMM, W. 1996. Msparc: a multithreaded Sparc. In Lecture Notes in Computer Science, vol. 1123. Springer-Verlag, Heidelberg, Germany. 461–469.

OEHRING, H., SIGMUND, U., AND UNGERER, T. 1999a. MPEG-2 video decompression on simultaneous multithreaded multimedia processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Newport Beach, CA). 11–16.

OEHRING, H., SIGMUND, U., AND UNGERER, T. 1999b. Simultaneous multithreading and multimedia. In *Proceedings of the Workshop on Multithreaded Execution, Architecture and Compilation* (Orlando, FL).

PATT, Y. N., PATEL, S. J., EVERS, M., FRIENDLY, D. H., AND STARK, J. 1997. One billion transistors, one uniprocessor, one chip. *Computer 30*, 9, 51–57.

PAUL, W. J., BACH, P., BOSCH, M., FISCHER, J., LICHTENAU, C., AND RÖHRIG, J. 2002. Real PRAM programming. In Lecture Notes in Computer Science, vol. 2400. Springer-Verlag, Heidelberg, Germany. 522–531.

PONTIUS, N. AND BAGHERZADEH, N. 1999. Multithreaded extensions enhance multimedia performance. In *Proceedings of the Workshop on Multithreaded Execution, Architecture and Compilation* (Orlando, FL).

ROTENBERG, E., JACOBSON, Q., SAZEIDES, Y., AND SMITH, J. E. 1997. Trace processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture* (Research Triangle Park, NC). 138–148.

RYCHLIK, B., FAISTL, J., KRUG, B., AND SHEN, J. P. 1998. Efficiency and performance impact of value prediction. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Paris, France). 148–154.

SENG, J. S., TULLSEN, D. M., AND CAI, G. Z. N. 2000. Power-sensitive multithreaded architecture. In *Proceedings of the IEEE International Conference on Computer design: VLSI in Computers and Processors* (Austin, TX). 199–206.

SERRANO, M. J., YAMAMOTO, W., WOOD, R., AND NEMIROVSKY, M. D. 1994. Performance estimation in a multistreamed superscalar processor. In Lecture Notes in Computer Science, vol. 794. Springer-Verlag, Heidelberg, Germany. 213–230.

SIGMUND, U., STEINHAUS, M., AND UNGERER, T. 2000. Transistor count and chip space assessment of multimedia-enhanced simultaneous multithreaded processors. In *Proceedings of the 4th Workshop on Multithreaded Execution, Architecture and Compilation* (Monterrey, CA).

SIGMUND, U. AND UNGERER, T. 1996a. Evaluating a multithreaded superscalar microprocessor versus a multiprocessor chip. In *Proceedings of the 4th PASA Workshop on Parallel Systems and Algorithms* (Jülich, Germany). 147–159.

SIGMUND, U. AND UNGERER, T. 1996b. Identifying bottlenecks in multithreaded superscalar multiprocessors. In Lecture Notes in Computer Science, vol. 1123. Springer-Verlag, Heidelberg, Germany. 797–800.

ŠILC, J., ROBIČ, B., AND UNGERER, T. 1998. Asynchrony in parallel computing: from dataflow to multithreading. *Parall. Distr. Comput. Practices 1*, 1, 57–83.

ŠILC, J., ROBIČ, B., AND UNGERER, T. 1999. *Processor Architecture: From Dataflow to Superscalar and Beyond*. Springer-Verlag, Heidelberg and Berlin, Germany, and New York, NY.

SMITH, B. J. 1981. Architecture and applications of the HEP multiprocessor computer system. *SPIE Real-Time Signal Processing IV 298*, 241–248.

SMITH, B. J. 1985. The architecture of hep. In *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, J. S. Kowalik, Ed. MIT Press, Cambridge, MA, 41–55.

SMITH, J. E. AND VAJAPEYAM, S. 1997. Trace processors: moving to fourth-generation microarchitectures. *Computer 30*, 9, 68–74.

SOHI, G. S. 1997. Multiscalar: another fourth-generation processor. *Computer 30*, 9, 72.

SOHI, G. S. 2001. Microprocessors—10 years back, 10 years ahead. In Lecture Notes in Computer Science, vol. 2000. Heidelberg, Germany. 208–218.

SOHI, G. S., BREACH, S. E., AND VIJAYKUMAR, T. N. 1995. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* (Santa Margherita Ligure, Italy). 414–425.

STEINHAUS, M., KOLLA, R., LARRIBA-PEY, J. L., UNGERER, T., AND VALERO, M. 2001. Transistor count and chip space estimation of simple-scalar-based microprocessor models. In *Proceedings of the Workshop on Complexity-Effective Design* (Göteborg, Sweden).

STERLING, T. 1997. Beyond 100 teraflops through superconductors, holographic storage, and the

data vortex. In *Proceedings of the International Symposium on Supercomputing* (Tokyo, Japan).

TENDLER, J. M., DODSON, J. S., FIELDS, JR., J. S., LE, H., AND SINHAROY, B. 2002. POWER4 system microarchitecture. *IBM J. Res. Dev. 46*, 1, 5–26.

TEXAS INSTRUMENTS. 1994. TMS320C80 Technical brief. Texas Instruments, Dallas, TX.

THISTLE, M. AND SMITH, B. J. 1988. A processor architecture for Horizon. In *Proceedings of the Supercomputing Conference* (Orlando, FL). 35–41.

TREMBLAY, M. 1999. A VLIW convergent multiprocessor system on a chip. In *Proceedings of the Microprocessor Forum* (San Jose, CA).

TREMBLAY, M., CHAN, J., CHAUDHRY, S., CONIGLIARO, A. W., AND TSE, S. S. 2000. The MAJC architecture: a synthesis of parallelism and scalability. *IEEE Micro 20*, 6, 12–25.

TSAI, J. Y. AND YEW, P. C. 1996. The superthreaded architecture: thread pipelining with run-time data dependence checking and control speculation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Boston, MA). 35–46.

TULLSEN, D. M., EGGERS, S. J., EMER, J. S., LEVY, H. M., LO, J. L., AND STAMM, R. L. 1996. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture* (Philadelphia, PA). 191–202.

TULLSEN, D. M., EGGERS, S. J., AND LEVY, H. M. 1995. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* (Santa Margherita Ligure, Italy). 392–403.

TULLSEN, D. M., LO, J. L., EGGERS, S. J., AND LEVY, H. M. 1999. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture* (Orlando, FL). 54–58.

UNGERER, T., ROBIČ, B., AND ŠILC, J. 2002. Multithreaded processors. *Computer J. 45*, 3, 320–348.

VAJAPEYAM, S. AND MITRA, T. 1997. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (Denver, CO). 1–12.

VIJAYKUMAR, T. N. AND SOHI, G. S. 1998. Task selection for a multiscalar processor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture* (Dallas, TX). 81–92.

WALL, D. W. 1991. Limits of instruction-level parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, CA). 176–188.

WALLACE, S., CALDER, B., AND TULLSEN, D. M. 1998. Threaded multiple path execution. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* (Barcelona, Spain). 238–249.

WALLACE, S., TULLSEN, D. M., AND CALDER, B. 1999. Instruction recycling on a multiple-path processor. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture* (Orlando, FL). 44–53.

WITTENBURG, J. P., MEYER, G., AND PIRSCH, P. 1999. Adapting and extending simultaneous multithreading for high performance video signal processing applications. In *Proceedings of the Workshop on Multithreaded Execution, Architecture and Compilation* (Orlando, FL).

YAMAMOTO, W. AND NEMIROVSKY, M. D. 1995. Increasing superscalar performance through multistreaming. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Limassol, Cyprus). 49–58.

Received June 2001; revised September 2002; accepted November 2002