Simultaneous Multithreading: Present Developments and Future Directions

Miquel Pericàs
Email: mpericas@ac.upc.es

June 24, 2003

Contents

Chapter 1

SMT: An Historical Perspective

With the advent of integrated electronics and the miniaturization of the transistor, computer engineers have been offered the possibility to add more and more logic into today's microprocessors. Most of this logic has been used to exploit ever larger quantities of Instruction Level Parallelism (ILP). Wide–issue out–of–order superscalar processors are the ultimate proposal of this tendency. But the complexity in the design and implementation of these chips is starting to yield diminishing returns when going to even wider issue configurations.

ILP has demonstrated to be very effective for the fast execution of numerical applications. The speed–ups for these programs are considerable when going from scalar processors to superscalar processors. Numerical applications have guided the processor market for many years, but with the arrival of personal computing and the server market, numerical applications are increasingly irrelevant to the sector. Commercial workloads have completely different characteristics. They have worse memory behavior and perform a lot of input/output. Exploitable ILP for these applications is very low and continuous stalls are clear signs that a different approach is needed to efficiently use the resources of the processor for these workloads.

One characteristic of many server workloads is that work can be distributed among different parallel threads, serving different requests simultaneously. This can be the case, for example, of a web server that has to send a web page to several clients at the same time. These tasks are independent. Thus, they may proceed concurrently. Similarly, on a personal computer, a user may be running several tasks on top of a multiprogrammed operating system. These tasks may be completely unrelated. Imagine for example a concurrent run of a word processor, a DVD player and a web browser, a very typical workload. These three tasks are also completely independent, so a concurrent execution is possible. These examples expose a different sort of parallelism than Instruction–Level Parallelism. This new sort of parallelism is called Thread–Level Parallelism (TLP), and it is increasingly important.

Architectures that are designed to exploit TLP at the processor level are called multithreaded architectures. While ILP architectures are ultimately limited by instruction dependencies and long–latency operations within the single executing thread, multithreaded architectures employ multiple threads with fast context switch between threads to hide memory and functional unit latencies. The first commercial multithreaded architectures were the HEP, TERA [1], MASA and Alewife, most of them designed in the late 80s. The TERA, for example, implements 128 parallel threads. On every tick of the clock, the processor logic selects a stream that is ready to execute and allows it to issue its next instruction. The TERA design team incorporated this technique primarily for the purpose of hiding memory access latency to the processor.

But fast context switches are not the ultimate solution for extracting TLP from various threads. In a single cycle, these architectures are still limited to execute instructions from a single thread. The limitation of this technique is still the amount of parallelism exploitable from a single thread in a single cycle. A refining of this technique is Simultaneous Multithreading (SMT) which allows the simultaneous execution of various threads in the same cycle. SMT attacks both *vertical* and *horizontal waste*[1].

This report makes a survey of implementations, SMT theory and future directions for Simultaneous Multithreaded Processors. Chapter 2 will explain SMT in more detail, comparing it with superscalar, coarse-grain multithreaded and fine-grain multithreaded execution. Section 3 talks about existing designs such as Intel's HyperThreading or Alpha's 21464 SMT design. Finally, section 4 presents some new directions in the design of SMT processors, concentrating on the topics of Fault Tolerance, Speculation and memory tolerance.

[1]Vertical waste are empty issue slots in consecutive cycles. Horizontal waste are empty issue slots in the same issue group

Chapter 2

Simultaneous Multithreading

Simultaneous Multithreading was introduced in 1995 as an evolution of the multithreading concepts that had already been successfully implemented in systems such as TERA [1] or HEP. In their original paper [14], Henry M. Levy, Dean M. Tullsen and Susan J. Eggers, propose SMT as a technique to substantially increase processor utilization in the face of both long memory latencies and limited available parallelism per thread. Simultaneous Multithreading is proposed as a method to permit several independent threads issue to multiple functional units each cycle.

The introductory chapter commented already on the tendency of increasing exploitable ILP as a means to increase the performance of recent superscalar cores. However, ILP is not an infinite resource of performance, and realizable ILP levels are quite limited compared to the maximal processing capacity of current implementations. Memory conflicts, control hazards, branch mispredictions and other problems are reducing processor efficiency to ridiculous levels. In their study, Levy et al. show how much time superscalar processors spend performing useful work and where all the lost cycles go. Their conclusions are devastating. An average SPEC95 simulation shows that on a typical superscalar core, the processor is busy for less than 20% of the time. The rest is wasted due to several conditions that cause the processor to wait. Figure 2.1 shows these results.

Now, take into account that the SPEC benchmark consists mostly of numerical applications with good practical ILP exploitation. Commercial or desktop workloads will presumably present an even worse picture.

What this means is that realizable ILP is low and that alternate approaches are needed to speed up applications. If ILP is no more available to the rescue, we can either increase processor speed from technology and speculation or we can choose to tackle different sources of parallelism. There are basically two more sources of parallelism available: Thread Level Parallelism (TLP) and Data Level Parallelism (DLP). DLP is normally found in numerical applications and is well exploited by vector architectures. It is not of much interest for general-purpose computing. TLP on the other side is fount in commercial and desktop workloads and is independent of ILP. It is of big interest here.

TLP has been traditionally exploited by multiprocessor systems, where each thread runs on a different CPU. These systems are expensive for single user systems and their efficiency is exactly as bad as the single-processor superscalars which we have just seen. A different way to make use of TLP parallelism is to context switch between threads when a processor stalls. This allows the processor to continue execution as long as any of the active threads has instructions in the READY state.

2.1 From Superscalar to SMT

Several design steps surface when moving from a superscalar out–of–order processor to a SMT processor. Each of these steps has been studied and implemented in several fashions. We now

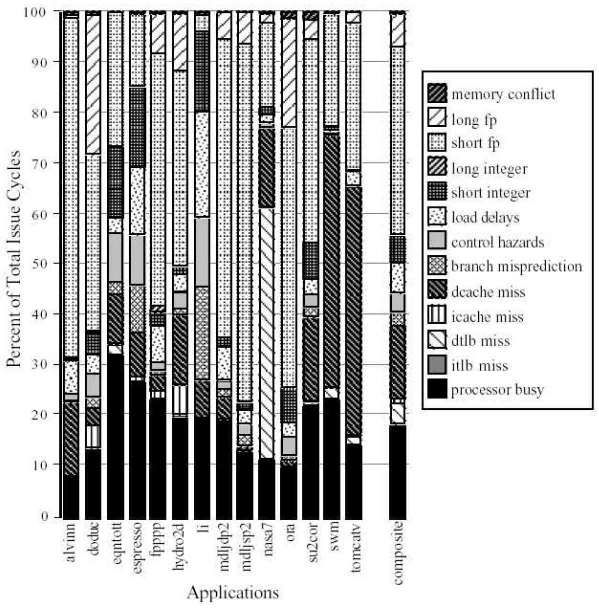

Figure 2.1: Lost issue slots for a typical superscalar processor running the SPEC95 benchmark

present the execution modes for a wide-issue processor using each of these techniques.

2.1.1 Superscalar Processors

The basic execution model is the single–threaded superscalar out–of–order processor. This execution model has been studied extensively and has been in use for many years. In this model we have the memory filled with several processes. The processor fetches instructions from a single process and continues to execute it until the underlying operation system decides that another process should get executed. This model is shown in figure 2.2. Although this model is very popular we see that many slots in the execution core are unused.

2.1.2 Multiprocessors

The simplest extension to this model that will allow the system to profit from multiple threads is to use a different processor for each thread. These systems, called multiprocessors (MPs), have been in use for many years now and many incarnations exist. The most basic model is the symmetric multiprocessor (SMP) whose structure is shown in figure 2.3. Although SMP computers can exploit TLP, the processor resources are used inefficiently due to the fact that each processor is still executing a single thread at each time in the traditional way.

2.1.3 Multithreaded Processors

The next approach to TLP that appeared is traditional multithreading, also called *Superthreading*. In this model the processor is extended with the concept of thread allowing the scheduler to chose instructions from one thread or another at each clock. This form of traditional multithreading comes in two fashions: coarse–grain multithreading and fine–grain multithreading.

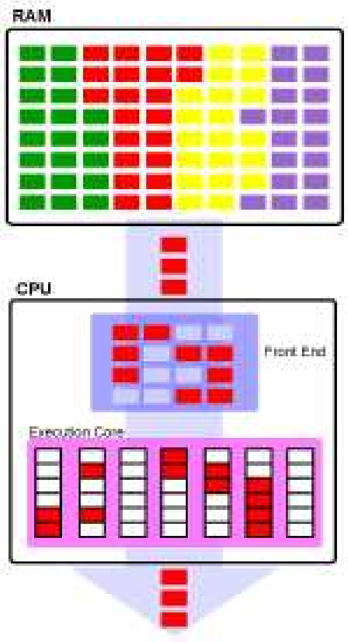

Figure 2.2: SuperScalar Execution

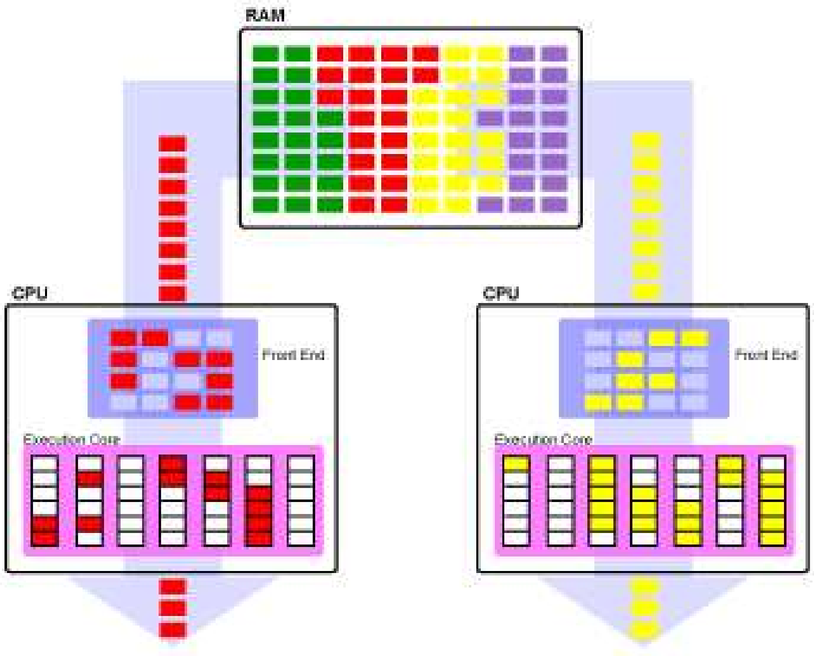

Figure 2.3: Multiprocessor Execution

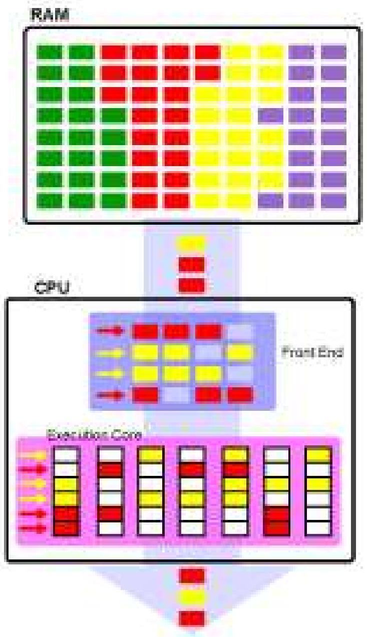

Figure 2.4: Fine–Grained Multithreading Execution

Coarse–Grain Multithreading

When Coarse–Grain Multithreading is being used, the processor executes the current thread normally and chooses to make a context switch only when a long–latency event occurs (such as a cache miss) or when a explicit context switch instruction is encountered. Note that this form of multithreading only hides long–latency events. Recall that only a small percentage of all wasted processor cycles come from dcache or icache misses. This solution has the advantage that it is simple to implement, but it's gains are not huge.

Fine–Grain Multithreading

An alternative approach that has gained momentum is to context switch the threads on every clock cycle. This way, on cycle 1 the scheduler would choose instructions from stream 1; on cycle 2 it would choose instructions from thread 2; and so on. This model is represented in figure 2.4. Occupancy of the execution core is now much higher. This model hides not only long–latency events, but also short latency events. Fine–grain multithreading is the first step forward towards effective TLP exploitation at CPU–level. But it has still some problems. All sources of vertical waste are completely eliminated through this technique, but horizontal waste is not. If a thread has little or no operations to execute issue slots will be wasted.

2.1.4 Simultaneous Multithreading

Simultaneous Multithreading (SMT) is a variation of traditional multithreading schemes that allows the scheduling logic to choose instructions belonging to all threads in each clock cycle. This additional condition allows to hide sources of both vertical and horizontal waste. The resulting scheme, shown in figure 2.5, has much higher execution core occupancy, and thus the processor is more efficient. The biggest penalty that SMT pays is the additional hardware required to implement all this logic. Full simultaneous issue for wide–issue processors with many threads is very expensive. Many SMT implementations choose to reduce the number of threads and limit

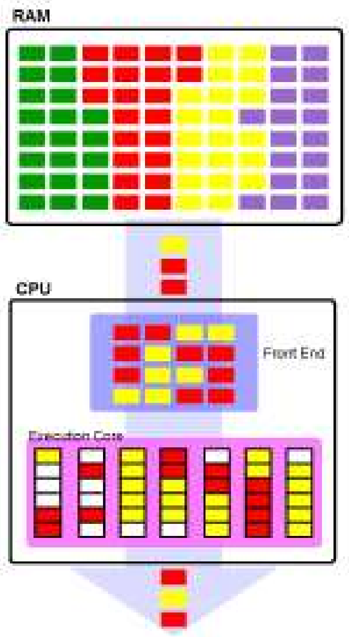

Figure 2.5: Simultaneous Multithreaded Execution

the shared resources so that the logic is not too complex. For example, Intel's HyperThreading implements only two thread contexts, while Alpha's 21464 design allowed for 4 parallel threads. Compare these values with the 128 threads of the fine–grained multithreaded TERA systems.

SMT is the most interesting multithreaded architecture. Effectively what an SMT processor does is transform TLP into ILP and execute various threads on an ILP processor. This is not really true, as some hardware support is needed, but the amount of extra hardware is small and the complexity is not too big. SMT turns out to be a simple and efficient solution for TLP extraction. The rest of this report will concentrate on this type of multithreaded architectures. The next section gives an overview on how an SMT processor works and what hardware is necessary.

2.2 Implementation and Design of SMT

Multithreading allows the overlapped sharing of functional units by multiple threads. To permit this sharing, the independent state of the thread must be duplicated by the processor. This includes, among others, a separate copy of the register file, a separate PC and a separate page table for each thread. Other resources such as the memory do not need to be replicated as the virtual memory mechanism already supports multiprogramming.

Ideally, the issue slot usage in an SMT processor is limited by imbalances in the resource needs and resource availability over multiple threads. In practice, other factors can also restrict how many slots are used. These include the number of active threads, finite limitations on buffers, the ability to fetch enough instructions from multiple threads, and practical limitations of what instruction combinations can issue from one thread and from multiple threads.

Simultaneous Multithreading uses the insight that a dynamically scheduled processor already has many of the hardware mechanisms needed to support exploitation of TLP. In particular, dynamically scheduled superscalars have a large set of virtual registers that can be used to hold the register sets of independent threads (assuming separate rename tables for each thread). Because register renaming provides unique register identifiers, instructions from multiple threads can be mixed in the data path without confusing sources and destinations across the threads. This observation leads to the insight that multithreading can be built on top of an out–of–order processor by adding a per–thread renaming table, keeping separate PCs, and providing the capability for instructions from multiple threads to commit. There are complications in handling instruction commit, since we would like instructions from independent threads to be able to commit independently. The independent commitment of instruction from separate threads can be supported by logically keeping a separate reorder buffer for each thread.

One issue which SMT has to deal with is possible degradation of single–thread performance. This effect can be minimized by having a preferred thread, which still permits multithreading to preserve some of its performance advantage with a smaller compromise in single–thread performance. This approach suffers from the fact that stalls encountered by the preferred thread are likely to reduce performance since the pipeline will probably not have a mix of instructions from several threads. But even though empty slots are more probable with this technique, simulations have shown that use of a preferred thread has only a small performance impact. However, if we want to maximize throughput, this is still best achieved having a combination of independent threads to hide all stalls in any combination of threads.

The preferred thread issue is not the only design issue. There are a lot of issues when designing these processors. Some of these will be covered in the next chapter. These design challenges include the following:

- Dealing with larger register files needed to hold multiple contexts
- Maintaining low overhead on the clock cycle. Two places where this is specially critical are the instruction issue, where more instructions need to be considered, and instruction completion, where choosing which instructions to commit may be challenging.
- Ensuring that cache conflicts generated by the simultaneous execution of multiple threads does not cause significant performance degradation.

2.3 Performance Study of SMT

To finish this section we give performance numbers on SMT. We're interested in how much speed is gained from thread–level parallelism. In this section we present the values obtained by the research of H. Levy in [14].

The evaluation measured the level of Instructions per Cycle (IPC) when going from one thread to eight concurrent threads. This evaluation is performed basically on three architectures:

- a fine–grained multithreaded architecture
- a simultaneous multithreaded architecture with single issue per thread (no more than one issue slot per thread each cycle)
- simultaneous multithreaded architecture with full simultaneous issue (no restrictions in the number of issue slots per thread)

The fine–grained architecture provides the smallest maximum speedup over the single–threaded architecture with only 2.1 when going from 1 thread to 8 threads (from 1.5 IPC to 3.2). Figure 2.6 shows little advantage in using more than 4 threads. Other studies give a similar result for coarse–grained multithreading architectures. With more than 4 threads, no vertical waste remains available to extract more parallelism. To obtain bigger IPC counts the only solution is to attack horizontal waste. Figure 2.7 gives IPC values for all SMT configurations. The speedups achieved by these configurations range from 3.2 to 4.2 over single threaded superscalar execution. Full simultaneous issue is the most aggressive configuration, achieving an IPC of 6.3 with 8 concurrent threads. Further, it is interesting to observe that no saturation effect has taken place, and it seems likely that increasing the number of threads will still be beneficial

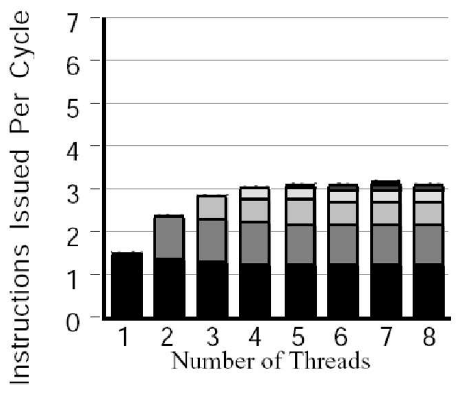

Figure 2.6: Fine–Grain Multithreading

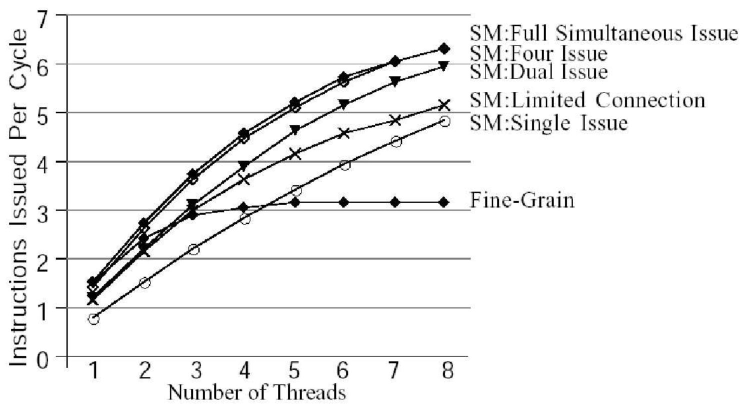

Figure 2.7: Results for all configurations

in terms of IPC. None of the configurations using SMT, including single issue, suffer from saturation before eight threads. At eight threads, the simplest configuration – single issue per thread – achieves an IPC of 4.9. It is also interesting to see that full simultaneous issue is not required to obtain good performance. A four issue configuration achieves almost identical IPC while a two–issue configuration achieves 94% of full simultaneous issue.

The increases in processor utilization are a direct result of threads dynamically sharing processor resources that would otherwise remain idle much of the time. But sharing also has negative effects. The effect of competition for issue slots and functional units in the full simultaneous issue model, where the lowest priority thread (at 8 threads) runs at 55% of the speed of the highest priority thread, can be seen in figure 2.8.

Another penalty is paid in the memory system. Running multiple threads side–by–side reduces locality. Thus, when moving from 1 thread to 8 threads, cache performance degrades:

- Wasted issue cycles due to icache misses grow from 1% at one thread to 14% at eight threads
- Wasted cycles due to dcache misses grow from 12% to 18%.
- data TLB waste also increases, from less than 1% to 6%.

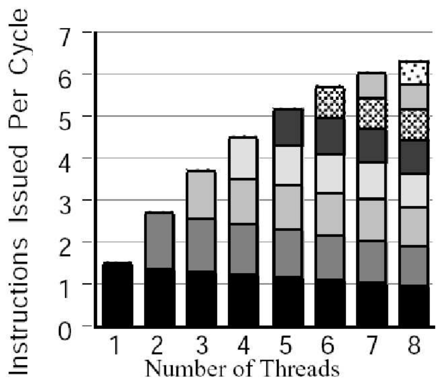

Figure 2.8: Full Simultaneous Issue

The cache hierarchy can be modified to better support SMT by combining private (per–thread) and shared (global) memory caches. The work by Levy suggests that shared caches optimize for a small number of threads will private caches perform better with a large number of threads.

The following chapter will deal with current implementations of this technique in processors such as the already mentioned Alpha 21464 and Intel P4 Xeons. We'll also discuss some other designs.

Chapter 3

Commercial SMT processors

The previous chapter has presented basic theory on Simultaneous Multithreading along with several simulation results. In this chapter we present several commercial implementations of the SMT concept. We will focus on Intel's HyperThreading as it is the only currently commercially available implementation for general purpose processors. Other proposals, such as the Alpha 21464 will also be dealt with.

3.1 Intel's HyperThreading

HyperThreading [3] was first introduced on the Intel Xeon processor in early 2002 for the server market. In November 2002, Intel launched the technology on the Intel Pentium 4, making HyperThreading available the consumer market. HyperThreading is Intel's attempts to bring simultaneous multithreading to the IA-32 architecture.

HyperThreading makes a single physical processor appear to be multiple logical processors. There is one copy of the architectural state for each logical processor, and these processors share a single set of physical execution resources. From a software perspective, this means that the operating systems and user programs can schedule processes or threads to logical processors as they would on conventional physical processors in a multiprocessor system. Note that Intel's current implementations of HyperThreading limit the number of contexts to two.

Each logical processor maintains a copy of the architectural state. The duplicated resources are primarily:

- Next–Instruction Pointer
- Instruction Stream Buffers
- Instruction translation look–aside buffer
- Return stack predictor
- Trace–cache plus local next–instruction pointer
- Trace–cache fill buffers
- Advanced Programmable Interrupt Controller (APIC)
- Register Alias Tables

HyperThreading requires duplication of additional miscellaneous pointers and control logic, but these are too small to point out. Figure 3.1 shows the structures that need to be replicated.

An important issue when adding more logic to a processor core is die size and complexity. HyperThreading is able to deliver a large performance improvement at minimal cost because

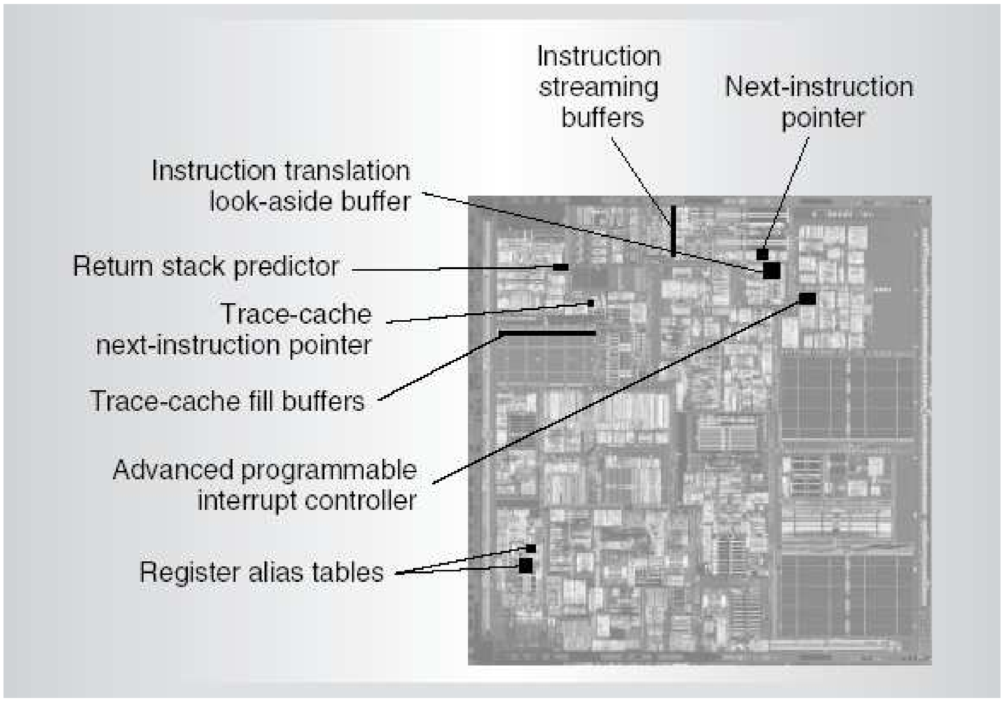
Instruction streaming buffers
Next-instruction pointer
Instruction translation look-aside buffer
Return stack predictor
Trace-cache next-instruction pointer
Trace-cache fill buffers
Advanced programmable interrupt controller
Register alias tables

Figure 3.1: Replicated Structures in HyperThreading

it entails only a small increase in die size. Logical processors share nearly all resources on the physical processor, including caches, execution units, branch predictors, control logic, and buses. The increase in die size is due to a second architectural state, additional control logic, and replication of a few processor resources.

But even though the die area increase is small, as Koufaty claims, the increase in design complexity is substantial. HyperThreading challenges many basic assumptions about single–threaded out–of–order design.

3.1.1 Sharing Policies

Several trade-offs have been tackled in the design of Intel's HyperThreading. The choice of a resource sharing policy for each shared resource is important because it can dramatically impact performance. In determining how to share resources, Intel chose among possible sharing schemes that included:

- *partition*, dedicating equal resources to each logical processor;
- *threshold*, flexible resource sharing with a limit on maximum resource usage, and;
- *full sharing*, flexible resource sharing with no limit on the maximum resource usage.

Choosing which method to use requires to consider carefully throughput versus fairness and potential livelock scenarios, as well as die size and complexity.

Partitioning is a good choice for the major pipeline queues, which provide buffering to avoid pipeline stalls and, ideally, remain full most of the time. Using partitioning for this case is important. If the two logical processors fully shared these queues, a slow thread could gain an unfair share of the resources and prevent a fast thread from making progress. Partitioning resources is simple, entails little implementation complexity, and ensures fairness and progress for both logical processors.

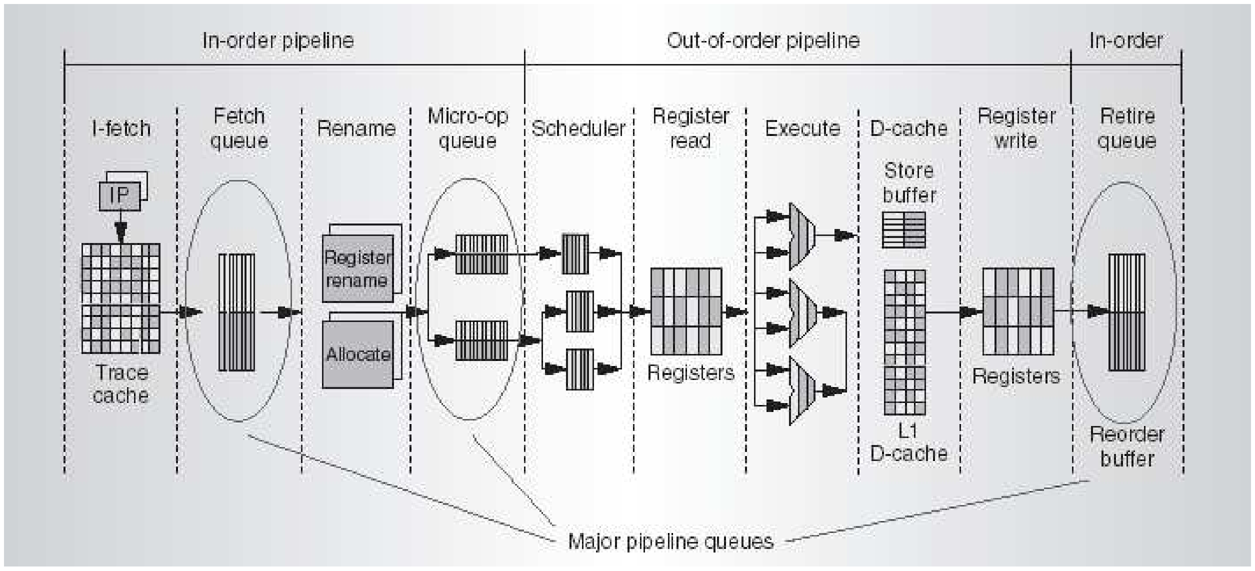

Figure 3.2: Pipeline of a pentium 4 processor with HyperThreading technology

The *threshold* sharing approach is ideally suited for small structures where the resource utilization is bursty, and the length of time a micro–op stays in the structure is short, fairly uniform, and predictable. A structure well suited for this approach are the processor schedulers. A threshold limiting the maximum number of entries each logical processor can use prevents one logical processor from blocking the other's access to the scheduler. The threshold lets the scheduler look for maximum parallelism among micro–ops across both threads, thereby improving execution resource utilization.

Fully shared resources are good for large structures in which working–set sizes are variable, and one logical processor cannot starve the other. A good example of such a structure are processor caches. In the NetBurst architecture, all processor caches are shared.

3.1.2 HyperThreading Performance

HyperThreading technology speeds up performance in two ways. First, it improves performance of applications that are already multithreaded. In this case, each logical processor will run software threads from the same application. Second, it speeds up a workload consisting of multiple applications by multitasking. In this case, each logical processor will likely run threads from different applications.

Figure 3.3 shows the HyperThreading technology performance boost on current multithreaded packages. The technology delivers a 15% to 26% performance boost on these applications. Figure 3.4 shows the performance benefits that HyperThreading delivers for several multitasking workloads. The performance boost, 15% to 27%, is very close to the multithreaded case.

HyperThreading is currently available in Pentium Xeon processors and will soon be included in the popular Pentium 4 processor.

3.2 The Alpha 21464

None of the remaining processors that will be presented during the rest of this chapter exists as a commercial product. The information available is in most cases scarce. However, many basic

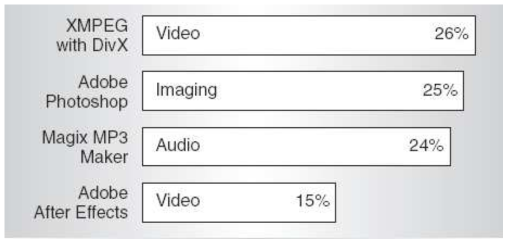

Figure 3.3: Performance of HyperThreading under multithreaded workloads

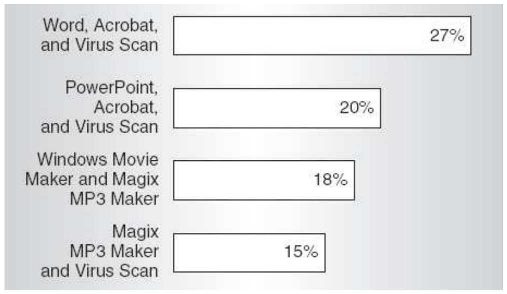

Figure 3.4: Performance of HyperThreading under multitasking

facts that allow for simple comparisons are available. The first such product that I review is the Alpha 21464 processor.

The Alpha 21464 [9], also known as the EV8, was canceled in June 2001 along the whole Alpha line of microprocessors. By then, the design of the next–generation Alpha, the 21464, had already been completed.

One of the new features that this chip was to include was support for simultaneous multithreading. SMT is implemented in the EV8 by adding circuitry to maintain 4 independent program counters and select a single thread each cycle for fetch from the instruction cache. As instructions are fetched, they are tagged with a thread-ID that follows the instructions through the instruction, execution, and memory units. The out–of–order issue queue is free to select issue–ready instructions from any of the 4 simultaneous threads. As with HyperThreading, the added logical complexity for SMT given the existing out–of–order issue policy and register renaming is modest, accounting for only about 6% additional die area. The most significant change experienced by the Alpha 21464 due to SMT support is the inclusion of 32 integer and 32 floating–point registers for each of the 4 threads. The architectural state of the machine therefore requires 256 registers. With additional registers required to support register renaming by covering up to 256 in–flight instructions, a total of 512 registers are implemented. The following figure depicts the pipeline of the Alpha processor.

Alpha 21464 only replicates the program counter and the register map. All other resources are shared, like the register file (although its size is increased), the instruction queue, the first and second level caches, the translation buffers (TLB) and the branch predictor.

Architecturally, the Alpha is visible to the programmer as a single CPU with 4 thread processing units (TPUs) that share hardware resources like the icache, the dcache, the TLB or the L2 cache.

3.2.1 EV8 multithreading performance

EV8 performance has been estimated on a variety of workloads. The first workload is a multiprogrammed workload. Using 4 threads (best case) and SpecInt as the benchmark, the EV8 achieves a speed–up of 2.2. The figure changes drastically with SpecFP, where the maximum speed–up

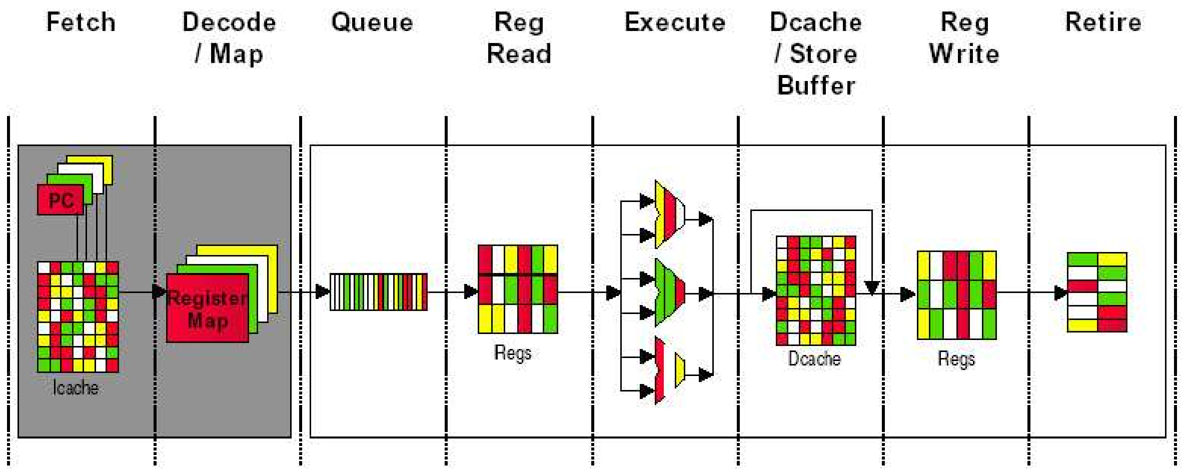

Figure 3.5: The Pipeline of the Alpha 21464 processor

drops to only 1.5 with 4 threads. Using a mix of Integer and Floating Point workloads the speed–up is again around 2.2–2.3. Another interesting case is the evaluation of multithreaded applications. The *Barnes* application obtains a small speed–up of 1.3, but other workloads like *sort* or *TP* show speed–ups around 2.3–2.5.

3.3 MemoryLogix MLX1

The MLX1 [12] project was presented at last years' Microprocessor Forum. With MLX1, Memory Logix aims at building a tiny multithreaded 586 core for smart mobile devices. The smart mobile device market has requirements that are different from the high performance market targeted by Intel and Compaq's Alpha:

- Balanced communication, entertainment and computing performance
- High MIPS / Watt (Power–Performance Efficiency)
- Low cost

The design goals of the MLX1 project are to produce a "synthesis–friendly" x86 core with support for FPU & MMX and 2.5 times the performance of a single–threaded core per MHz. The die size should be 2.5 times the size of an ARM10 core. High MIPS is obtained in this design by using SMT techniques.

In the MLX1 SMT implementation, threads share the register file and the cache. The core has three identical fetch units. Threads share a single–instruction decoder. Thread switch can occur when several conditions appear:

- The thread's decode buffer is not full.
- The thread's issue queue is full.
- A branch or a 4-cycle load has started.
- A serialization instruction has been encountered.
- The 8th consecutive decode happened.

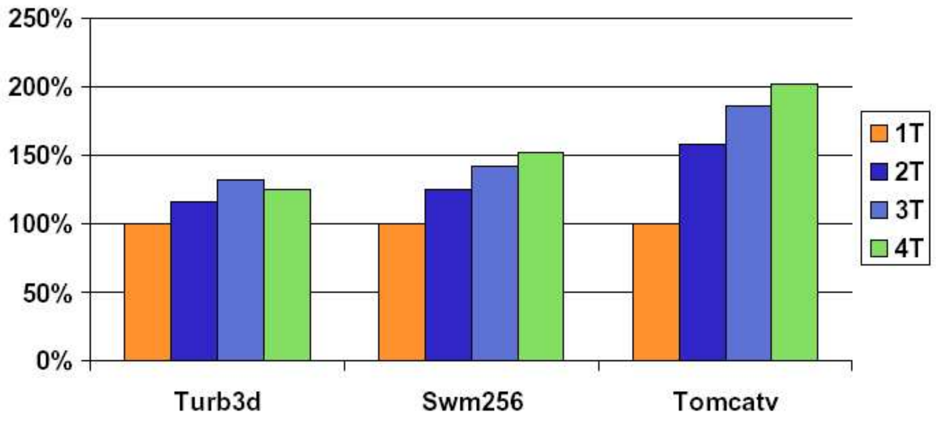

Figure 3.6: Performance of the 21464 using a multitasking workload composed of SPEC95 applications

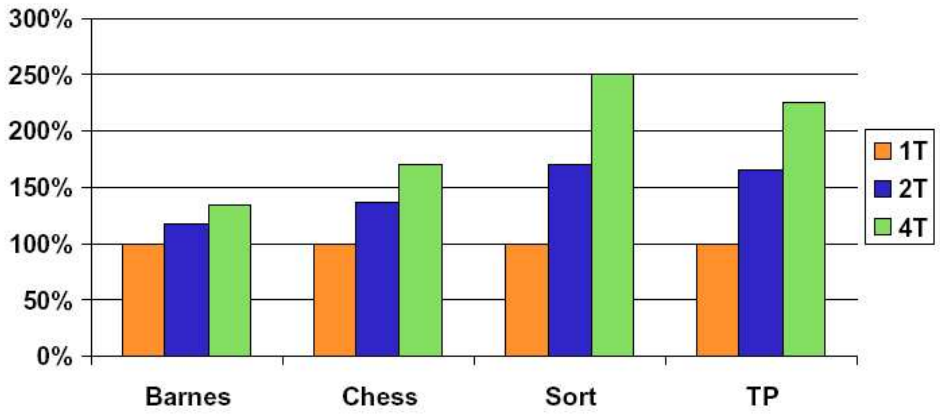

Figure 3.7: Performance of the 21464 using a workload of multithreaded applications

Moreover, all threads share a queue of 64 rename registers (8 GPRs + EIP). Each thread has its own future register map, which is updated at rename. Similarly, each thread has a private architecture register map, updated at completion.

The thread switch conditions listed previously seem to suggest that the MLX1 is a fine–grained multithreading architecture. However, Memory Logix describe this processor as a SMT core, which suggests that the processor may be combining some features of both techniques.

3.4 Clearwater Networks CNP810SP

While the previous processors are implementations of SMT for general purpose computers, the next two examples target the network processor market. None of both designs will see the light as both companies have already shut down, but the designs are still worth to be considered in this survey.

The first of these processors is the CNP810SP [6], a network processor with SMT capabilities. In the CNP810SP, up to eight threads may execute simultaneously. In each cycle anywhere from zero to three instructions can be executed from each of the threads depending on instruction dependencies and availability of resources. The maximum instructions per cycle (IPC) of the entire core is ten. In each cycle two threads are selected for fetch and their respective program counters (PCs) are supplied to the dual-ported instruction cache. Each port supplies eight instructions, so there is a maximum fetch bandwidth of 16 instructions. Each of the eight threads has its own instruction queue (IQ) which can hold up to 16 instructions. The two threads chosen for fetch in each cycle are the two that have the fewest number of instructions in their respective IQs. Each of the eight threads has its own 31 entry register file (RF). Since there is no sharing of registers between threads, the only communication between threads occurs through memory.

The eight threads are divided into two clusters of four for ease of implementation. Thus the dispatch logic is split into two groups where each group dispatches up to six instructions from four different threads. Eight function units are grouped into two sets of four, each set dedicated to a single cluster. There are also two ports to the data cache that are shared by both clusters. A maximum of 10 instructions can be dispatched in each cycle. The functional units are fully bypassed so that dependent instructions can be dispatched in successive cycles. The following picture shows the main structure of a CNP810SP processor.

3.5 FlowStorm Porthos

The FlowStorm Porthos [7] is another SMT network processor designed by the same team who designed the CNP810SP. The company itself, FlowStorm, was dissolved in December 2002 which lead to the cancellation of the *Porthos*.

Porthos is designed for stateful networking applications: i.e. those applications where there is a requirement to support a large amount of state with little locality of access. Stateful applications require a large number of external memory accesses. To satisfy this, a high degree of parallelism is required. Porthos makes use of multiple multithreaded processing engines to support this parallelism in a design that supports 256 simultaneous threads in eight processing engines. Each thread has its own independent register file and executes instructions formatted to a general purpose ISA, while sharing execution resources and memory ports with other threads.

Porthos is optimized to sacrifice single threaded performance for efficiency, so that a design is achieved that is realizable in terms of silicon area and clock frequency. Such an architecture is called *Massive Multithreading*, or MMT. A maximum of one instruction can be executed from each of the threads depending on instruction dependencies and availability of resources. Porthos can achieve a sustained execution rate of 40 instructions per cycle (5 instructions per cycle per tribe). The tribe pipeline structure that allows this is shown below.

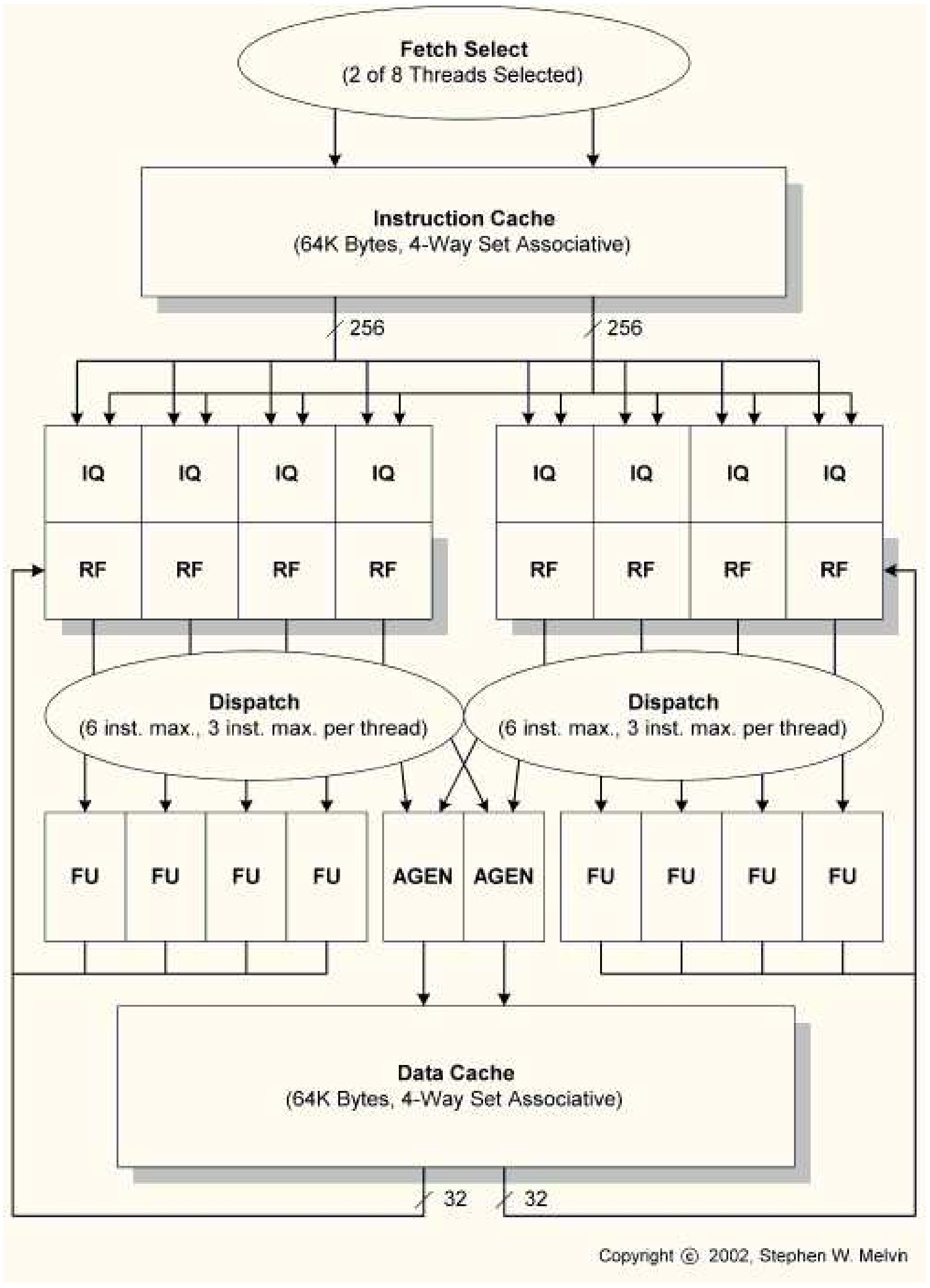
Fetch Select
(2 of 8 Threads Selected)
Instruction Cache
(64K Bytes, 4-Way Set Associative)
256
256
IQ
RF
Dispatch
(6 inst. max., 3 inst. max. per thread)
Dispatch
(6 inst. max., 3 inst. max. per thread)
FU
AGEN
Data Cache
(64K Bytes, 4-Way Set Associative)
32
32
Copyright © 2002, Stephen W. Melvin

Figure 3.8: The CNP810SPS processor

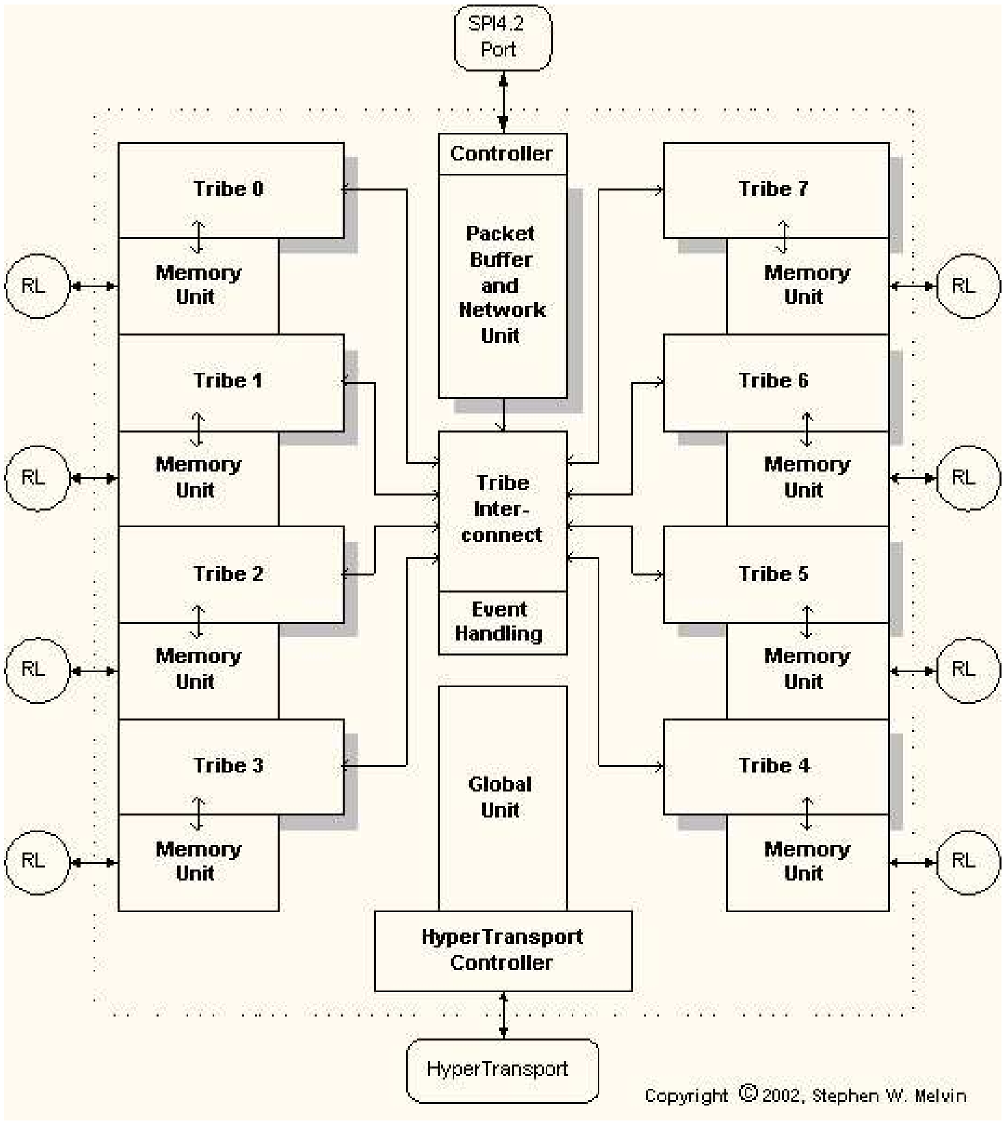

Figure 3.9: The FlowStorm Porthos processor

Each tribe consists of three decoupled pipelines. The first pipeline is the instruction fetch pipeline. In this stage two threads are selected for fetch each cycle. For each thread, up to 4 instructions can be fetched from a shared data cache. This results in a maximum throughput of 8 instructions per cycle per tribe.

FlowStorm uses the term Massive Multithreading (MMT) to refer to architectures that more or less have the following characteristics:

- Execute general purpose ISA (not microengines)
- Hardware support for large numbers of threads
- Single threaded performance has been sacrificed
    - no branch predictor
    - in–order issue
    - no speculative fetch
    - no speculative execution
    - single port to the register file per thread
    - no ALU bypass logic
    - maximum of one operation per thread issued per cycle

The idea behind this structure is that maximum power–efficiency is obtained with processors that are much more conservative than current out–of–order superscalars. The idea is to extract performance using the SMT concept which is very interesting in the area of network processors. This approach is similar to the design of Piranha, a chip multiprocessor designed at Compaq. Piranha was designed with commercial applications in mind such as OLTP or web servers. These applications have little ILP to offer and can achieve more speed–up exploiting TLP. For this reason, each processor within a Piranha chip is a simple in–order Alpha processor with a issue bandwidth of one instruction.

3.6 Other Processors

In this chapter we have presented 5 commercial designs of simultaneous multithreaded processors. Sadly, until now only one of these designs has seen the light. Even though, SMT remains a very complexity–effective choice to extract performance at TLP level. So, I expect that this technique will continue to gain popularity. Sun announced back in 2001 that it was building an SMT processor. I have not discussed its design here because no details are available at this moment. On the long term it is very probable that all major players in the marked will start to make use of this technique. Chip multiprocessors are another cost–efficient way to exploit TLP. The Power4 is a typical CMP implementation. However, scarce resource sharing in CMP (only some caches are shared) make this design option less cost–effective compared to SMT.

The network processor market will probably also see implementations of SMT soon arriving to the market. Although both proposals discussed here will not see the light, multithreading concepts have been used extensively in network processors for years in the form of fine–grain and coarse–grain multithreading. The move towards SMT is probably just a matter of time.

Chapter 4

Future Tendencies

In the previous chapters we have briefly looked at the basic concepts behind SMT and presented several real and proposed implementations. But SMT itself has also spawned a lot of research on new applications of this technique.

4.1 Transient Fault Detection

One of the topics that has generated most interest in relation to simultaneous multithreaded architectures is transient fault detection and recovery in microprocessors. As transistor integration continues to scale, severely reduced design tolerances implied by gigahertz clock rates may result in frequent and arbitrary transient faults. High clock rate designs require (1) small voltage swings for fast switching and power considerations, and (2) widespread use of high performance, but relatively undisciplined, circuit design techniques, e.g. dynamic logic. This combination is problematic because dynamic logic is susceptible to noise and crosstalk and low voltage levels at charged nodes only make it more so.

Another cause for transient hardware faults are cosmic rays. Cosmic rays can alter the voltage levels that represent data values in microprocessor chips. The energy flux of these rays can be reduced to acceptable levels with six feet or more of concrete. However, this width is significantly greater than normal computer room roofs or walls. The frequency of transient faults is still very low today – typically less than one fault per thousand computers per year. This figure makes fault–tolerant computers attractive only for mission–critical applications. OLTP and space programs are examples for applications that require fault–tolerance. But as has been said, future microprocessors are more likely to experience faults due to their smaller feature sizes, reduced voltage levels, higher transistor counts and reduced noise margins. Thus, in the future, even low–end computers may need to implement transient fault detection and even recovery into the chips.

Specialized fault–tolerance techniques, such as error correction codes (ECC) for on–chip memories and *Recomputing with Shifted Operands* (RESO) for ALUs, do not adequately cover arbitrary logic faults characteristic of this environment. Self–checking logic can provide general coverage, but chip–area and performance goals may preclude applying self–checking logic globally. A different approach such as using redundant processors is probably too costly for small general purpose computers.

Rotenberg [10] first suggested using SMT processors for fault detection because they can provide redundancy by running multiple copies of the same program simultaneously. Such a processor, called simultaneous and redundantly threaded processor (SRT), provides three advantages over conventional hardware replication:

- An SRT processor requires less hardware because it can use time and information redundancy in places where space redundancy is not critical. For example, an SRT processor can

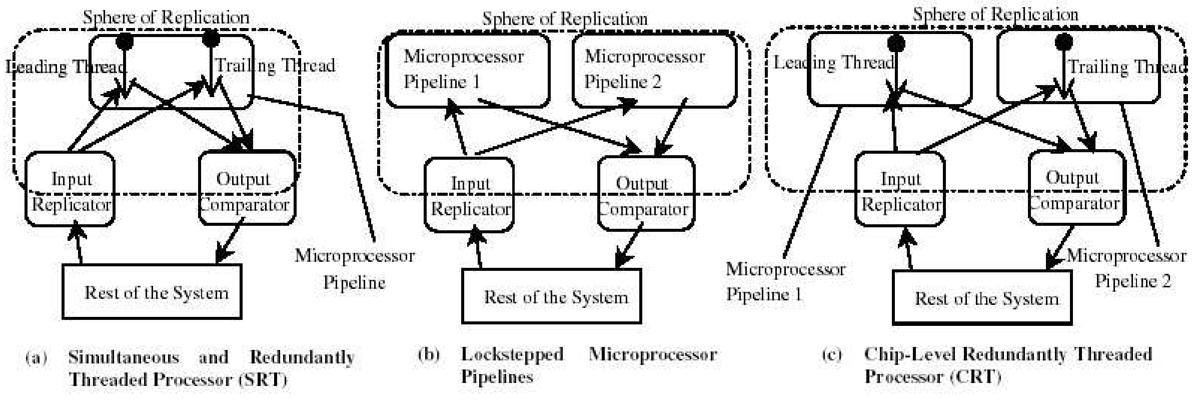

Figure 4.1: Fault Detection Schemes

have an unreplicated datapath protected by ECC or parity and shared between redundant threads.

- An SRT processor can potentially provide performance superior to an equivalently sized hardware–replicated solution because it partitions resources among execution copies dynamically rather than statically
- SRT processors processors may be economically more viable since it should be feasible to design a single processor that can be configured in either SRT or SMT depending on the target system. This merged design would enjoy the combined market volume of both high–reliability and high–throughput processors.

The most advanced proposals in this segment are from ISCA 2002. We will briefly discuss Reinhardt's and Mukherjee's Redundant Multithreading (RMT) and Vijaykumar's Simultaneous and Redundantly Threaded processors with Recovery (SRTR).

4.1.1 Redundant Multithreading (*RMT*)

In a processor with SMT capabilities it's possible to detect transient faults by running two copies of the same program as separate threads, giving them the same inputs and comparing the results. This technique is called Redundant Multithreading (RMT) [8]. Rainhardt's paper studies RMT techniques in the context of both single– and dual–processor simultaneous multithreaded single–chip devices. Their analysis shows that implementing this technique on a commercial SMT processor has subtle implementation complexities. They find that RMT can be a more significant burden for single–processor devices than prior studies indicate. Their solution to this problem is the application of RMT techniques in a dual–processor device, a technique they call chip–level redundant threading (*CRT*). This approach shows higher performance than lockstepping the two cores. The performance gain is more notable in multithreaded workloads.

4.1.2 Simultaneously and Redundantly Threaded processors with Recovery (*SRTR*)

Vijaykumar [15] propose a scheme called Simultaneously and Redundantly Threaded processors with Recovery (SRTR). SRTR is similar in RMT in that it replicates an application into two communicating threads, but in this case, one executes ahead of the other. Both threads perform the same computation and the values they produce are compared. In SRT (the predecessor to SRTR), it is possible that a leading instructions commits before the comparison with the trailing thread occurs. SRT relies in the trailing thread to detect faults. In contrast, SRTR does not allow the

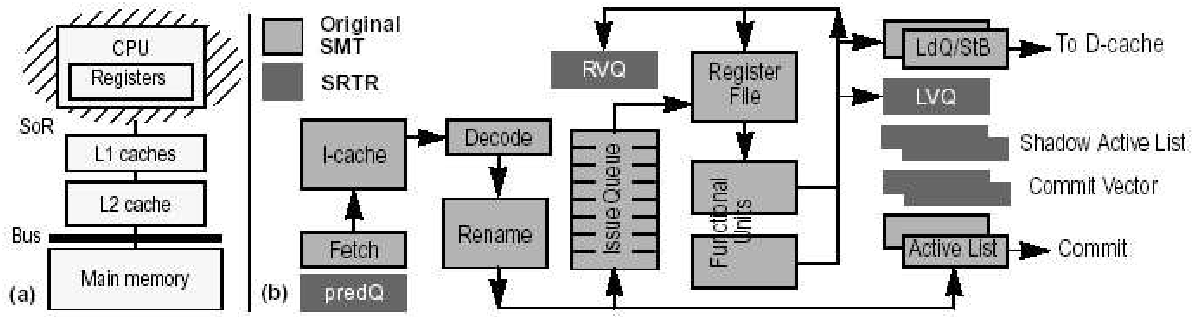

Figure 4.2: Simultaneously and Redundantly Threaded processors with Recovery

leading thread to commit until the trailing thread has checked the result. This may be necessary because it is not possible to undo a faulty instruction once the instruction commit has occurred.

As such, this technique imposes a big penalty because all instructions stall at commit until the trailing thread has checked for detection. Instead of this, SRTR exploits the time between the completion and commit of leading instructions. SRTR compares the leading and trailing values as soon as the trailing instruction completes, typically before the leading instruction reaches the commit point. To reduce pressure on the register file, SRTR makes use of the fact that faults propagate through dependencies so that only the last instruction in a chain makes use of the register value queue. SRTR performs within 1%–7% of SRT for SPEC95 integer and floating–point programs, respectively.

4.2 Memory Tolerance

One of the advantages of SMT is the possibility to continue instruction fetch in the event of long–latency cache misses. However, when little or no TLP is available, as is often the case, this technique doesn't help. However, the additional hardware of SMT processors can be used to hide memory latencies by making use of an interesting idea.

4.2.1 Software Pre–Execution in SMT Processors

Data addresses in many irregular applications are extremely hard to predict. Prefetching techniques are useless for these cases because the only accurate way to predict these addresses is by actually computing them, i.e. the processor needs to execute the code that generates the address. One attractive approach is to use an SMT processor and make use of an idle thread to perform *pre–execution*. Pre–execution [4] is essentially the combined act of speculative address generation and prefetching to accelerate performance of the main thread. Such a scheme was presented by Chi.Keung Luk of the VSSAD/Alpha Development Group in the context of SMT architectures. One of the key points of this proposal is to use a software approach to control pre–execution. This allows the processor to handle some of the most important access patterns that are typically difficult to prefetch. The other key point of the technique is its minimal hardware requirements. This technique does not require integration of pre–execution results, it has no need of shortening programs for pre–execution, and it has no need of special hardware to copy register values upon thread spawns. It's important to note that this is a completely software solution to the problem of prefetching.

The average speedup of this technique is 24% for a set of irregular applications. This represents a 19% speedup over state–of–the–art software controlled prefetching, despite the simplicity of the technique.

4.3 Speculation

The topic of speculation has gained a lot of interest in the computer research area. Most computer activity contains little randomness and is very predictable. Further, limitations in extractable ILP due to cache misses, branch prediction misses and inherent data dependencies impose a very big penalty on the performance of the processor, so that speculation is used aggressively to overcome these limitations. Speculation is basically about executing one path of instructions without full knowledge that the instruction stream or data is correct.

In the context of SMT processors an obvious idea is to exploit TLP hardware to speculate and execute various threads simultaneously, each one speculating in a different direction. In this section we present several ideas of how to make use of simultaneous multithreading to speculate.

4.3.1 Speculative Data–Driven Multithreading (DDMT)

Mispredicted branches and loads that miss in the cache cause most of the stalls experienced by sequential processors. These instructions are called *critical instructions*. Despite their importance, it is difficult for a sequential processor to prioritize critical computations, because all instructions must be fetched sequentially, regardless of their contribution to performance. The technique presented here, *Speculative data–driven multithreading* [11] is a general–purpose mechanism for overcoming this limitation.

In DDMT, critical computations are annotated so that they can execute standalone. When the processor predicts an upcoming instance of a critical instruction, it microarchitecturally forks a copy of its computation as a new kind of speculative thread: a data–driven thread (DDT). The DDT executes in parallel with the main program thread, but typically generates the critical result much faster since it fetches and executes only the critical computation and not the whole program. A DDT "pre–executes" a critical computation and effectively "consumes" its latency on behalf of the main thread. A DDMT component called *integration* incorporates results computed in DDTs directly into the main thread, sparing it from having to repeat the work.

Simulations of an implementation of DDMT on top of a simultaneous multithreading processor have been done using program profiles to create DDTs and annotate them into the executable. The experiments show that DDMT pre–execution of critical loads and branches can improve performance significantly.

4.3.2 Threaded Multiple Path Execution (*TME*)

TME [16] is a technique that exploits hardware present on Simultaneous Multithreading (SMT) processors to speculatively execute multiple paths. When there are fewer threads in an SMT processor than hardware contexts, threaded multi–path execution uses spare contexts to fetch and execute code along the less likely path of hard–to–predict branches.

To enable spawning of speculative threads for threaded multi–path execution, the *Mapping Synchronization Bus* (MSB) is used. The MSB copies the current mapping region between thread contexts. The MSB is a bus which is connected to all the contexts' mapping regions.

Results show that TME increases the single–thread performance of an SMT with eight hardware contexts by 14%–23% on average for programs with high misprediction rate.

4.3.3 Simultaneous Subordinate Microthreading *SSMT*

SSMT [2] is a proposal to use SMT techniques on workloads without TLP. Work on Simultaneous Multithreading provides little benefit to programs that aren't partitioned into threads. SSMT tries to avoid this by spawning subordinate threads that perform optimizations on behalf of the single primary thread. These threads, written in microcode, are issued and executed concurrently with the primary thread. They directly manipulate the microarchitecture to improve the primary

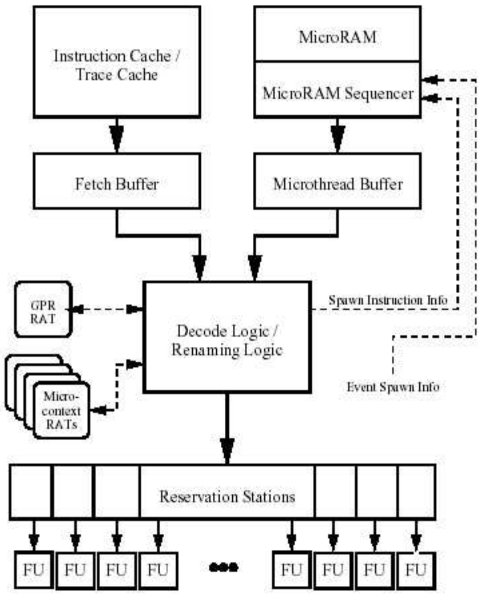

Figure 4.3: Simultaneous Subordinate Microthreading

thread's branch prediction accuracy, cache hit rate and the prefetch effectiveness. All three contribute to the performance of the primary thread.

4.3.4 Thread–Level Speculation (TLS)

Thread–Level Speculation [13] addresses the same problems as SSMT, i.e. the lack of parallel software. But instead of executing a new thread that makes optimizations for the primary thread, TLS is a compiler technique that forks of new threads optimistically, despite the compiler not knowing if these threads are actually independent or not. In their work J. Steffan and T. Mowry present and evaluate a design of TLS, that scales to any machine size because it is a straightforward extension of writeback invalidation–based cache coherence (with itself scales both up and down). This allows the scheme to perform well on both single–chip multiprocessors and on large scale machines where communication latencies are many times larger.

4.3.5 Profile–based Speculative Multithreaded Processors

The techniques exposed in this section deal with the possibility of extracting TLP from programs that are hard to parallelize. The partitioning of programs into threads is mostly accomplished via heuristics. Marcuello and Gonzalez [5] propose to use a profile–based mechanism to divide the programs into threads. For this purpose, a dynamic control flow graph is built where each node represents a basic block and edges represent possible control flows among basic blocks. Using a profile, the probability of execution is assigned to each node. When a pair of nodes has a probability to be executed above a threshold, it is considered as a spawning point. When the processor reaches the spawning point it will start a speculative thread.

This scheme has been evaluated on a Clustered Speculative Multithreaded Processor. Results show almost a 20% performance boost compared with traditional heuristics. The speed–up over a single thread execution is higher than 5x for a 16–thread–unit processor and close to 2x for a 4–thread–unit processor. In this chapter we have presented several trends in research on Simultaneous Multithreaded Processors. Two topics, *Transient Fault Detection* and *Speculative Multithreading* have captured most attention. None of these techniques has been implemented

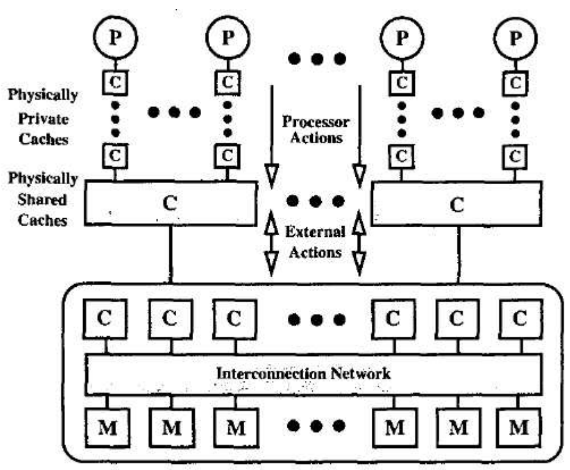

Figure 4.4: Thread–Level Speculation

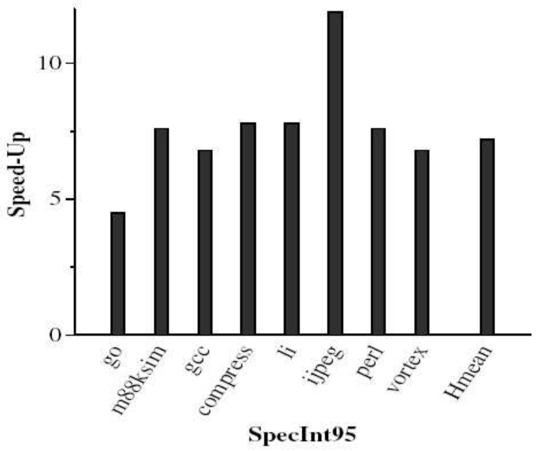

Figure 4.5: Some ideal speedups for Profile–based speculative multithreaded processors

or announced for a commercial system yet. However, both topics are of big importance and both achieve excellent results so it should be a matter of time when this gets implemented.

4.4 Conclusions

Simultaneous Multithreading is a technique that is able to exploit big quantities of TLP with little hardware overhead. Currently only one commercial implementation exist (Intel's *Hyper-Threading*) but many more have been proposed. I believe that most future architectures will have support for TLP extraction at the processor core level. SMT cores do not require modifications to the ISA. From the programmer point of view a SMT core behaves exactly the same as a multiprocessor. This is similar to what happens with Chip–Multiprocessors (CMP).

SMT improve the performance of workloads with TLP. However, many workloads do not benefit from this feature. Thus speculation techniques have been proposed to extract more ILP from these workloads making use of SMT hardware.

Transient Fault Detection is another topic of interest that can benefit from SMT. Currently, only mission–critical applications do require redundancy in the core. SMT's multiple contexts offer a simple solution to the redundancy problem and with little cost. In the future, with reduced feature sizes and increased error rates, some of the techniques proposed in this last chapter may be applied even to desktop systems.

However, SMT also has some drawbacks. As I already commented, single–thread performance decreases due to the constraints that most architectures have. Another interesting topic that has not been discussed here is power. Additional contexts will surely increase the power requirements of SMT processors. In how far this increase will be limiting needs to be studied still. HyperThreading is limited enough (2 contexts) so that it is difficult to extract generic conclusions about SMT.

Bibliography

[1] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6, 1990.

[2] R. Chappel, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous subordinate microthreading (ssmt), May 1999.

[3] David Koufaty and Deborah T. Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro 2003*, pages 56–65, 2003.

[4] Chi-Keung Luk. Tolerating memory latency through Software-Controlled Pre-Execu tion in simultaneous multithreading processors. pages 40–51.

[5] Pedro Marcuello and Antonio Gonzalez. Thread-spawning schemes for speculative multithreading. In *Proceedings of the Eight International Symposium on High-Performance Computer Architecture (HPCA-02)*, 2002.

[6] Stephen Melvin. Clearwater networks cnp810sp simultaneous multithreading (smt) core. *www.zytek.com/~melvin/clearwater.html*, 2000.

[7] Stephen Melvin. Flowstorm porthos massive multithreading (mmt) packet processor. *www.zytek.com/~melvin/flowstorm.html*, 2003.

[8] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *29th International Symposium on Computer Architecture (ISCA'02)*, 2002.

[9] R. et al Preston. Design of an 8-wide superscalar risc microprocessor with simultaneous multithreading. In *IEEE International Solid-State Circuits Conference*, page 344, 2002.

[10] Eric Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Symposium on Fault-Tolerant Computing*, pages 84–91, 1999.

[11] Amir Roth and Gurindar S. Sohi. Speculative data-driven multithreading. In *International Symposium on High Performance Computer Architecture (HPCA'01)*, pages 37–48, 2001.

[12] Peter Song. A tiny multithreaded 586 core for smart mobile devices. In *2002 Microprocessor Forum (MPF)*, 2002.

[13] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Intl. Conf. on Computer Architecture (ISCA'27)*, pages 1–24, 2000.

[14] Dean M. Tullsen, Susan Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the22th Annual International Symposium on Computer Architecture*, 1995.

[15] T.N. Vijaykumar, Irith Pomeranz, and Karl Cheng. Transient-fault recovery using simultaneous multithreading. In *29th International Symposium on Computer Architecture (ISCA'02)*, 2002.

[16] S. Wallace, B. Calder, and D.M. Tullsen. Threaded multiple path execution. In *Intl. Conf. on Computer Architecture (ISCA'27)*, pages 238–249, 1998.