Mitosis: A Speculative Multithreaded Processor Based on Precomputation Slices

Carlos Madriles, Carlos García-Quiñones, Jesús Sánchez, *Member*, *IEEE*, Pedro Marcuello, *Member*, *IEEE Computer Society*, Antonio González, *Member*, *IEEE*, Dean M. Tullsen, *Senior Member*, *IEEE*, Hong Wang, *Member*, *IEEE*, and John P. Shen, *Fellow*, *IEEE*

Abstract—This paper presents the Mitosis framework, which is a combined hardware-software approach to speculative multithreading, even in the presence of frequent dependences among threads. Speculative multithreading increases single-threaded application performance by exploiting thread-level parallelism speculatively, that is, executing code in parallel, even when the compiler or runtime system cannot guarantee that the parallelism exists. The proposed approach is based on predicting/computing thread input values via software through a piece of code that is added at the beginning of each thread (the precomputation slice). A precomputation slice is expected to compute the correct thread input values most of the time but not necessarily always. This allows aggressive optimization techniques to be applied to the slice to make it very short. This paper focuses on the microarchitecture that supports this execution model. The primary novelty of the microarchitecture is the hardware support for the execution and validation of precomputation slices. Additionally, this paper presents new architectures for the register file and the cache memory in order to support multiple versions of each variable and allow for efficient rollback in case of misspeculation. We show that the proposed microarchitecture, together with the compiler support, achieves an average speedup of 2.2 for applications that conventional nonspeculative approaches are not able to parallelize at all.

Index Terms—Speculative thread-level parallelism, precomputation slices, thread partitioning, multicore architecture.

1 Introduction

Most high-performance processor vendors have introduced designs that can execute multiple threads simultaneously on the same chip through simultaneous multithreading [8], [18], multiple cores [29], or a combination of the two [20]. The way that thread-level parallelism (TLP) is currently being exploited in these processors is through nonspeculative parallelism. There are two main sources of nonspeculative TLP: 1) executing different applications in parallel [31] and 2) executing parallel threads generated by the compiler or programmer from a single application. In the former case, running different independent applications in the same processor provides an increase in throughput (the number of jobs finished per time unit) over a single-threaded processor. In the latter case, programs are partitioned into smaller threads that are executed in parallel. This partitioning process may significantly reduce the execution time of the parallelized application over single-threaded execution.

- *C. Madriles, C. García-Quiñones, J. Sánchez, P. Marcuello, and A. González are with the Intel-UPC Barcelona Research Center, Intel Corporation, Jordi Girona 29 Nexus II, 3A Barcelona 08034, Spain. E-mail: {carlos.madriles.gimeno, carlos.garcia.quinones, f.jesus.sanchez, pedro.marcuello, antonio.gonzalez}@intel.com.*
- *D.M. Tullsen is with the Department of Computer Science and Engineering, University of California, San Diego, 9500 Gilman Dr. #0404, La Jolla, CA 92093-0404. E-mail: tullsen@cs.ucsd.edu.*
- *H. Wang is with the Microprocessor Research Lab, Intel Corp., 3600 Juliette Ln., SC-12, Santa Clara, CA 95054. E-mail: hong.wang@intel.com.*
- *J.P. Shen is with the Nokia Research Center, Nokia, 955 Page Mill Rd., Suite 200, Palo Alto, CA 94304-1003. E-mail: John.Shen@nokia.com.*

Manuscript received 2 Nov. 2006; revised 2 Aug. 2007; accepted 3 Oct. 2007; published online 23 Oct. 2007.
Recommended for acceptance by U. Ramachandran.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0383-1106.
Digital Object Identifier no. 10.1109/TPDS.2007.70797.

Executing different applications in parallel increases throughput but provides no gain at all for workloads consisting of just one application. On the other hand, partitioning applications into parallel threads may be a straightforward task in regular applications but becomes much harder for irregular programs, where compilers usually fail to discover sufficient TLP. This is typically because of the necessarily conservative approach taken by the compiler. This results in the compiler including many unnecessary interthread communication/synchronization operations or, more likely, concluding that insufficient parallelism exists. However, although compilers are typically unable to find parallel threads in irregular applications and thus benefit from the hardware support for multiple contexts, this does not mean that those applications do not contain large amounts of parallelism. In many cases, they do.

Recent studies have proposed the use of coarse-grained speculation techniques to reduce the execution time of these applications. This kind of parallelism is usually referred to as *speculative TLP*. This technique reduces the execution time of applications by executing several speculative threads in parallel. These threads are speculative in the sense that they may be data and control dependent on previous threads, and their correct execution and commitment are not guaranteed. Therefore, in this model, we execute sections of the code in parallel, because we suspect that they may be independent (or predictably dependent), and we check whether that was true after the fact. For these architectures, additional hardware/software is required to validate these threads and eventually commit them.

There are two main strategies for speculative TLP: 1) the use of helper threads to reduce the execution time of high-latency instructions by means of side effects [5], [7], [24], [34] and 2) relaxing the parallelization constraints and

parallelizing the applications into speculative threads (for example, [1], [6], [12], [15], [25], [27], [30], and [33], among others).

The Mitosis processor is based on the speculative parallelization approach. Each of the speculative threads executes a different portion of the program. The partitioning process allows the spawning of speculative threads, even where the compiler cannot guarantee correct execution. Once the thread finishes, the speculative decisions are verified. If they were correct, then the application has been accelerated. If a misspeculation (control or data) has occurred, then the speculative work done is discarded, and the processor continues with the correct threads.

It is occasionally possible to partition a program into enough parallel threads (depending on the number of hardware contexts) such that there are few or no dependences between them [27]. For architectures that take this approach, they speculate that there are no dependences between threads, and they recover when there are dependences. However, for most programs, it is necessary to create threads where there are control/data dependences across these partitions to fully exploit available parallelism. The manner in which such dependences are managed critically affects the performance of the speculative multithreaded processor. Previous approaches have used the hardware communication of produced register values [14], explicit synchronization [9], [30], and hardware value prediction [16] to manage these dependences.

In contrast, Mitosis takes a completely new software-based approach to managing both data and control dependences among threads. We find that this produces values earlier than the hardware-assisted value-passing approaches and more accurately than hardware value prediction approaches (due to the use of instructions derived from the original code). Each speculative thread is prepended with a *speculative precomputation slice (p-slice)* that precomputes live-ins (those register and memory values that are consumed by the speculative thread and may be computed by prior threads still in execution). The p-slice typically executes in a fraction of the time of the actual code that produces those live-ins, because it skips over all code that is not specifically computing live-in values.

The Mitosis framework is composed of a compiler that partitions the applications into speculative threads and generates the corresponding p-slice and a speculative multithreaded processor that is able to manage multiple speculative threads. This paper presents the whole Mitosis framework but focuses more heavily on the hardware architecture, which contains a number of novel features. These include hardware support for the execution and validation of p-slices, a new multiversion register file that manages global register state and intercore dependences with virtually no latency overhead, and a multiversion memory subsystem that uses a replication cache (RC) to allow the processor to achieve cache performance similar to what would be seen by the application if it were running in a single core.

Early performance results reported by the Mitosis processor show an average speedup of about 2.2 for a subset of the Olden benchmark suite and 1.75 over a configuration that models perfect L1 data caches.

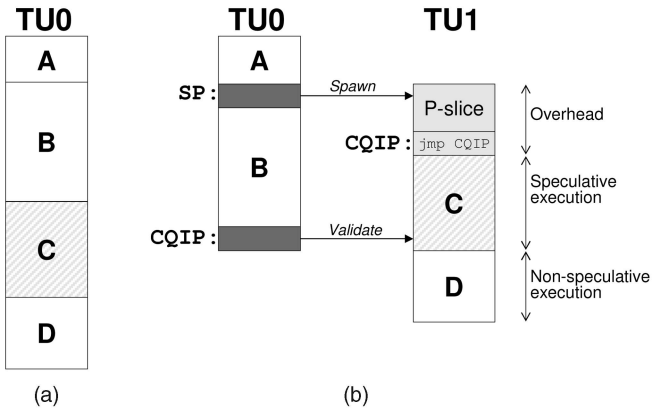


Fig. 1. Execution model of the Mitosis processor. (a) Sequential execution. (b) Parallel Mitosis execution.

The rest of this paper is organized as follows: Section 2 presents the execution model. The Mitosis compiler is briefly described in Section 3. Section 4 details the main features of the microarchitecture. Some performance results are shown in Section 5. Related work is reviewed in Section 6, and finally, Section 7 summarizes the main conclusions of this work.

2 Execution Model of the Mitosis Processor

Fig. 1 shows the Mitosis processor execution model. Programs are partitioned into speculative threads statically with the Mitosis compiler. The partitioning process explores the application to insert spawning pairs. A Spawning pair is a pair of instructions made up of the point where the speculative thread will be spawned (the *spawning point* (SP)) and the point where the speculative thread will start its execution (the *control quasi-independent point* (CQIP)) based on earlier terminology [17]. The Mitosis compiler also computes the p-slice corresponding to each spawning pair. The p-slice is the subset of instructions between the SP and CQIP, which produce values consumed shortly after the CQIP.

In the example shown in Fig. 1, then, the choice of a particular SP and CQIP allow the code in section C (the code following the CQIP) to execute in parallel with the code in B (the code between the SP and CQIP) after the completion of the p-slice. As can be seen from this example, the amount of parallelism exposed by this technique is sensitive to the size of B (the distance between SP and CQIP) and the size of the p-slice (which is derived from B). Thus, if the size of the p-slice is similar to that of B, that is, most of the computation of B is consumed in C, then there will be little or no gain from these techniques. If the p-slice is very small relative to B, then the code will execute as if the regions were completely parallel, despite the existence of dependences.

This new binary produced by the Mitosis compiler is executed on the Mitosis processor. Applications run on a Mitosis processor in the same way as on a conventional superscalar processor. However, when a *spawn* instruction is found, the processor looks for the availability of a free context (or thread unit (TU)). Upon finding a free one, the p-slice of the corresponding speculative thread is assigned to a TU. The p-slice executes on that TU, ending with an unconditional jump to the CQIP, thereby starting the execution of the speculative thread.

If no free TU is available when the spawn instruction is executed, the system looks for a speculative thread that is more speculative (further in sequential time) than the new thread that we want to spawn. If any is found, the most speculative one is canceled, and its TU is assigned to the new thread.

Threads in the Mitosis processor are committed in a sequential program order. The thread executing the oldest instructions in program order is nonspeculative, whereas the rest are speculative. When any running thread reaches the CQIP of any other active thread, it stops fetching instructions until it becomes nonspeculative. Then, a verification process checks that the next speculative thread has been executed with the correct input values. If the speculation has been correct, the nonspeculative thread is committed, and its TU is freed. Moreover, the next speculative thread now becomes the nonspeculative one: it will either continue executing in nonspeculative mode or, if it has already reached another CQIP, immediately proceed to verify its successor. If there is misspeculation, the next speculative thread and all its successors are squashed, and the nonspeculative thread continues executing the instructions beyond the CQIP. Other less conservative recovery mechanisms such as selective reissuing can reduce the impact of squashing the whole thread and its successors but are not considered in this paper due to their higher complexity. Our results indicate that the high accuracy of the slice-based live-in prediction makes the system relatively insensitive to squash overhead.

In the Mitosis processor, the spawning mechanism is highly general. Any speculative or nonspeculative thread can spawn a new speculative thread upon reaching an SP. Moreover, speculative threads can be spawned out of the program order; that is, threads can be created in a different order than they will be committed.

3 Mitosis Compiler

The Mitosis compiler [10] for this architecture performs the following highly coupled tasks: 1) select the best candidate spawning pairs and 2) generate the p-slices for each pair and optimize them to minimize the overhead. We will detail these two steps in the following.

The proposed scheme has been implemented in the code generation phase (after optimizations) of the ORC compiler [13]. The compiler makes use of the information provided by an edge profile. This information includes the probability of going from any basic block to each of its successors and the execution count of each basic block.

3.1 Pair Identification and Selection

A key feature of the proposed compilation tool is its generality in the sense that it can discover a speculative TLP in any region of the program. The tool is not constrained to analyze potential spawning pairs at loop or subroutine boundary, but practically, any pair of basic blocks is considered a candidate spawning pair. To reduce the search space, we first apply the following filters to eliminate candidate pairs that likely have little potential:

1. Spawning pairs in routines whose contribution to the total execution of the program is lower than a threshold are discarded.
2. Both basic blocks of the spawning pair must be located in the same routine and at the same loop level. Notice that this does not constrain the possibility of selecting candidate pairs different from loop iterations and subroutine continuations. For instance, potentially important spawning pairs like those that partition large sequential regions are also eligible.
3. The length of the spawning pair (as the average length of all the paths from the SP to the CQIP) must be higher than a certain minimum size in order to overcome the initialization overhead when a speculative thread is created. It must also be lower than a certain maximum size in order to avoid very large speculative threads and stalls due to the lack of space to store speculative state.
4. The probability of reaching the CQIP from the SP must be higher than a certain threshold.
5. Finally, the ratio between the length of the p-slice and the estimated length of the speculative thread must be lower than a threshold. This ratio is a key factor in determining the potential benefits of the thread.

Selecting the best set of spawning pairs requires assessing the benefit of any given candidate pair. However, determining the benefits of a particular spawning pair is not straightforward and cannot be done on a pair-per-pair basis. The effectiveness of a pair depends not only on the control flow between the SP and the start of the thread, the control flow after the start of the thread, and the accuracy and overhead of its p-slice but also on the number of hardware contexts available to execute speculative threads and interactions with other speculative threads running at the same time.

Once the set of candidate pairs is built, the selection of pairs is performed using a greedy search by means of a cost model that estimates the expected benefit of any set of potential spawning pairs. This model takes into account the aforementioned parameters. The goal of this model is to analyze the behavior and interactions of the set of spawning pairs when the program is executed on the given Mitosis processor. The output of the model is the expected execution time. For the sake of simplicity, we assume in the following that the execution of any instruction takes a unit of time. However, the model can be extended in a straightforward manner to include different execution times for each static instruction (for example, using average memory latencies obtained through profiling).

3.2 P-Slice Generation

For each spawning pair, the compiler identifies its input values and generates the p-slice, which consists of a minimal set of instructions between the SP and CQIP, assuring that the input values are almost always correct. The p-slice is built by traversing the control-flow graph backward from its CQIP until the SP. The identification of data dependences is relatively straightforward for register values but is harder for memory values. To detect memory dependences, the Mitosis compiler uses a memory dependence profiler.

Large p-slices significantly constrain the benefits of speculative threads. However, a key feature of the Mitosis SpMT architecture is that it can detect and recover from misspeculations. This opens the door to new types of

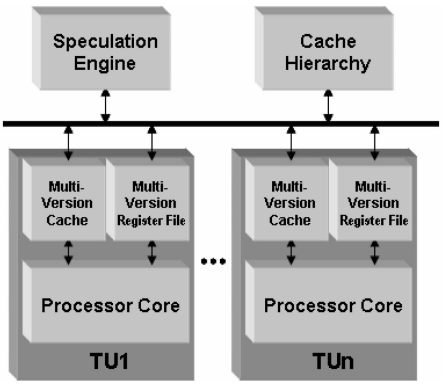


Fig. 2. Mitosis processor microarchitecture.

aggressive/unsafe optimizations that otherwise could not be applied by the compiler and that have the potential of significantly reducing the overhead of p-slices. The following optimizations have been considered so far:

- *Memory dependence speculation.* Based on a memory dependence profile, dependences that never occur in practice or occur very infrequently are discarded.
- *Branch pruning.* Based on an edge profile, paths that exhibit low probability of being taken are ignored. These paths may belong to either the body of the speculative thread or its p-slice. In the first case, live-ins in the pruned path are ignored. In the second case, all the instructions of the pruned path from the p-slice are removed, as well as the predecessors of these removed instructions, if their output is not used elsewhere.

A lot of research remains to be done on the compiler side, mainly in the form of new p-slice optimizations and pair selection techniques. In the Mitosis paradigm, the performance of the architecture is clearly sensitive to the quality of the compiler; thus, we expect the performance of the architecture to improve significantly as the compiler improves.

Full details of the Mitosis compiler infrastructure can be found in [10].

4 The Mitosis Processor

The Mitosis processor has a multicore design similar to an on-chip multiprocessor (CMP), as shown in Fig. 2. Each TU executes a single thread at a time and is similar to a conventional superscalar core. Each speculative thread may have a different version for each logical register and memory location. The ability to have different active versions for the register and memory values is supported by means of a local register file (LRF) and a local cache per TU. In this section, the most relevant components of the microarchitecture are described, including the Multiversion Register File and the Multiversion Memory. These two structures maintain a single global view of register and memory contents while allowing multiple local versions of those values.

The most novel aspect of this architecture is the use of p-slices to precompute live-ins for the speculative threads. This has several implications and challenges for the microarchitecture. Although the p-slice and the thread execute consecutively on the TU, they differ in significant ways. First, the p-slice executes in the speculative domain of the parent (at the SP), whereas the thread executes in a more speculative domain, following the CQIP. Second, register and memory values produced by the p-slice must be validated (usually) but are not committed: they predict the processor state but are not processor states. Conversely, values produced in the thread must be committed but do not need to be validated.

Therefore, the architecture must support the following features for executing p-slices:

- It must be able to distinguish between a p-slice and a regular computation.
- P-slice register and memory writes must be handled differently from regular writes.
- The state of the parent thread at the SP must remain visible to the p-slice, even if values are overwritten by the parent, as long as the p-slice is still executing.

The specific implementation of these features, as well as other novel features of the architecture to support the Mitosis speculative multithreading model, is described in the following sections.

4.1 Spawning Threads

A speculative thread starts when any active thread executes a spawn instruction in the code. The *spawn* instruction triggers the allocation of a TU, the initialization of some registers, and the ordering of the new thread with respect to the other active threads. These tasks are handled by the Speculation Engine.

To initialize the register state, the *spawn* instruction includes a mask that encodes which registers are p-slice live-ins. Registers included in the mask are copied from the parent thread to the spawned one. Our studies show that on the average, just six registers need to be initialized.

Any active thread is allowed to spawn a new thread when it executes a *spawn* instruction. Additionally, speculative threads can be spawned out of the program order; that is, a speculative thread that would be executed later than another thread, if executed sequentially, can be spawned in reverse order in the Mitosis processor. Previous studies have shown that out-of-order spawning schemes have a much higher performance potential than in-order approaches [1], [20], [15].

It is also necessary to properly order the spawned thread with respect to the rest of the active speculative threads. The order among threads will determine where a thread is going to look for values not produced by itself and will be used to check misspeculations. Akkary and Driscoll [1] proposed a simple mechanism in which the new spawned thread is always assumed to be the most speculative. On the other hand, Marcuello [15] proposed an order predictor based on the previous executions of the speculative threads. This latter scheme takes advantage of the fact that thread spawning patterns are typically repeated. This suggests a simple order predictor that predicts for a new spawned thread the same order seen in its previous executions with the same coexecuting threads.

Fig. 3 shows the order predictor used in the Mitosis processor. It is made up of a table that is indexed by the

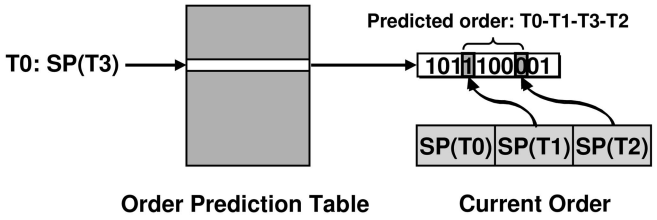


Fig. 3. Mitosis order predictor.

SP address of the newly spawned thread, and it contains all the relationship information of that thread regarding the other active threads. The entries of the table may be tagged or nontagged. In this work, we have considered a nontagged approach. Each entry of the table holds a bitmap. When a thread spawns a new thread, this bitmap is accessed for all the active speculative threads that are within the parent thread and the most speculative thread. Each bit of the bitmap represents whether the new thread would be more (that is, 1) or less (that is, 0) speculative, a thread that hashes to that index in the bitmap. The hash function uses the original spawn point address for each of the active threads.

Fig. 3 gives an example. Thread 0 spawns a new thread (T3). We know that T3 occurs in the program order after T0, but we do not know where it falls relative to threads T1 and T2. Thread T1's hash function indexes to the fourth bit, and T2 indexes to the eighth bit. Those bits indicate that T3 (based on its past history) should fall before T2 but after T1. Thus, when T1 reaches T3's CQIP, it will assume that T3 is its successor.

The bitmaps of the Order Prediction Table may contain single-bit information for each thread (as in Fig. 3) or can also be implemented by means of n-bit saturated counters. In our experiments, we assume that each entry contains eight 2-bit saturated counters, since this provides the best average hit ratios. With this configuration, our studies show that the average hit ratio of the order predictor is higher than 98 percent.

4.2 Multiversion Register File

To achieve correct execution and high performance, the architecture must simultaneously support the following seemingly conflicting goals: a unified view of all committed register state, the coexistence of multiple versions of each register, register dependences that cross TUs, and a latency similar to a single LRF.

This support is provided by the Mitosis *multiversion register file*. As shown in Fig. 4a, the register file has a hierarchical organization. Each TU has its own LRF, and there is a *Global Register File* (GRF) for all the TUs. There is also a table, the *Register Versioning Table* (RVT), that has as many rows as logical registers and as many columns as TUs and tracks which TUs have a copy of that logical register.

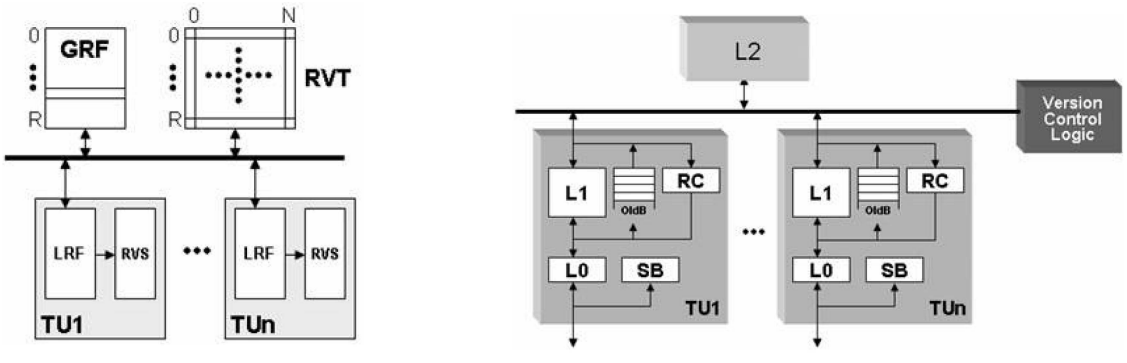
When a speculative thread is spawned in a TU, the p-slice live-in registers are copied from the parent thread to the LRF of the spawned thread. These registers are identified by a mask in the spawn instruction, as it was previously pointed out. Remember that slices have the characteristic that they are neither more nor less speculative than the parent thread. In fact, they execute a subset of instructions of the parent thread, so the speculation degree is the same. Therefore, it needs to access the same register versions, as the parent thread would see at the SP (while executing in a different TU).

The values produced by a speculative thread are always stored in its LRF. On the first write to a register in the LRF, the corresponding entry of the RVT is set to mark that this TU has a local copy of this register.

When a thread requires a register value, the LRF is checked first. If the value is not present, then the RVT is accessed to determine the closest predecessor thread that has a copy of the value. If there are no predecessors that have the requested register, the value is obtained from the GRF.

There is an additional structure per core, the Register Validation Store (RVS), which is used for validation purposes. When a speculative thread reads a register for the first time and this value has not been produced by the thread itself, the value is copied into this structure. Additionally, those register values generated by the p-slice that have been used by the speculative thread are also inserted in the RVS. When this thread is validated, the values in the RVS are compared with the actual values of the corresponding registers in the predecessor thread. By doing so, we ensure that values consumed by the speculative thread are identical to those that would have been seen in a sequential execution. Because we explicitly track the values consumed, incorrect live-ins produced by the slice that are not consumed do not cause misspeculation.

Finally, when the nonspeculative thread commits, all the modified registers in the LRF are copied into the GRF. An evaluation of the performance impact of the Multiversion Register File design has been done. On the average, more than



(a) (b)

Fig. 4. (a) Multiversion register file. (b) Multiversion memory system.

99 percent of the register accesses are satisfied from the LRF for the benchmarks evaluated. Thus, the average perceived latency for register access is essentially equal to the latency of the LRF, meeting the goals for our register file hierarchy. When we miss in the LRF, most of the accesses are to the GRF, and only a few go to a remote register file. This is because most of the register values needed by the speculative thread are usually computed by the p-slice: this is another side benefit of the accuracy of the p-slice mechanism.

4.3 Multiversion Memory System

Similar to the register storage, the memory system of the Mitosis processor provides support for multiversioning; that is, it allows different threads to have different versions for each memory location. As shown in Fig. 4b, each TU has its own L0 and L1 data caches, which are responsible for maintaining all speculative values, since speculative threads are not allowed to modify the main memory. Moreover, three additional structures are needed at each TU: the *Slice Buffer* (SB), the *Old Buffer* (OldB), and the *RC*. Finally, there is a global L2 cache shared among all the TUs that can only be updated by the nonspeculative thread and centralized logic to handle the order list of the different variables, that is, the Version Control Logic (VCL).

The architecture of this memory system is inspired by the Speculative Versioning Cache (SVC) [11], with notable extensions to handle p-slices in our implementation. As a summary, the Mitosis memory subsystem contains the following novel features: support for p-slice execution and the RC. This section focuses on these new features.

A load in a p-slice needs to read the value of that memory location at the SP, whereas a load in the thread needs to see the value at the CQIP, unless the value was produced by a store in the same thread. For this reason, during the execution of a p-slice, the processor needs to have an exact view of the machine state of the parent at the time of the *spawn* instruction. Therefore, when a thread performs a store and any of its children is still executing its p-slice, the value of that memory location needs to be saved before overwriting it, since its children may later require that value. The buffers used for storing the values that are overwritten while a child is executing a p-slice are referred to as OldBs. Each TU has as many OldBs as direct child threads are allowed to be executing a p-slice simultaneously. Thus, when a speculative thread that is executing the p-slice performs a load to a memory location, it first checks for a local version at its local memory. In case of a miss, it checks in its corresponding OldB from the parent thread. If the value is there, then it is forwarded to the speculative thread. Otherwise, it looks for it at any less speculative thread cache. When a speculative thread finishes its slice, it sends a notification to its parent thread to deallocate the corresponding OldB. Finally, it is possible that a thread finishes its execution and some of its children are still executing the p-slice. Then, those OldBs cannot be freed until these threads finish their corresponding p-slices.

The values read from the corresponding OldB during the execution of a p-slice are stored in the local memory but have to be marked in some way to avoid being read by any other thread and mistaken as state expected to be valid beyond the CQIP. To prevent a more speculative thread from incorrectly reading these values, a new bit is added to the SVC protocol, which is referred to as the *Old Bit*. When a thread that is executing the slice performs a load from the parent OldB, the value is inserted into the local cache with the Old Bit set. Then, when a more speculative thread requests this value and if it finds the Old Bit set, it knows that the value stored in that cache may potentially be old and is ignored. Finally, when the slice finishes, all the lines of the local cache with the Old Bit set are invalidated.

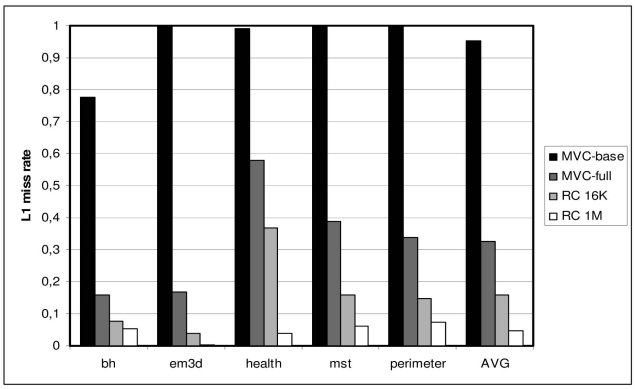


Fig. 5. Multiversion cache (L1) miss ratio for the basic MVC protocol, the full MVC protocol (with the stale bit), and the RC.

SBs are used to store the values computed by the p-slice in order to be later validated. When a thread is allocated to a TU, the SB is reset. All the stores performed during the execution of the p-slice go directly to the SB, bypassing the cache. Each entry of the SB contains an address, a value, a read bit, and a valid bit. When the slice finishes and the speculative thread starts the execution of its body, the SB is checked first whenever a value is read from the memory. If the value is there, the read bit is set, and the value is copied to the cache. When the thread is validated, all the values that have been consumed from the SB have to be checked for their correctness. Thus, those entries that have their read bit set are compared with the value of the corresponding memory locations seen by the previous (terminated) thread.

The speculative multithreading execution model can cause problems for the cache subsystem, especially when implemented on a multicore architecture without shared L1 caches. A sequential execution would pass values from stores to loads within the single cache, whereas with speculative multithreading, it is common that the load executes in a different thread context than the store. Additionally, data that exists in the cache when the thread starts is not necessarily valid for this thread. Since this architecture is capable of exploiting parallelism from relatively small threads, if threads only exploited same-thread memory locality, cache miss rates would be extremely high. However, without modifying the memory subsystem, this is exactly what happens, and preliminary experiments confirmed this result: every newly spawned thread begins with a completely empty cold cache.

The results of this preliminary study for a subset of the Olden benchmarks and a reduced input set (executing about 10 million instructions) are shown in Fig. 5. These results correspond to the miss ratio for a memory system with a 16-Kbyte local L0 data cache and a four-way 1-Mbyte local L1 cache. Complete details of the simulations and the

TABLE 1
Mitosis Processor Configuration (per TU)

Fetch, in-order issue and commit bandwidth	2 bundles (6 instructions)	**Crossfeed latency**	3 cycles
Pipeline Length	14 stages	**Replication Cache**	4-way 16 KB
Reorder Buffer Size	512 instructions	**Local Register File**	1 cycle
I-Cache	64KB	**Global Register File**	6 cycles
L0-Cache	4-way associative 16KB – hit latency: 1 cycle	**Spawn/Validation overhead**	5 cycles/15 cycles
L1-Cache	4-way associative 1MB – hit latency: 4 cycles	**Slice Buffer**	1K-entry – hit latency: 1 cycle
L2-Cache (share)	4 way associative 8 MB – hit latency: 8; miss latency: 250	**Old Buffers**	3 of 128-entry each

benchmarks are described in the next section. The leftmost bar corresponds to the initial implementation of the multiversion memory, as described above. The miss ratio can be reduced by introducing a *stale bit* in the memory protocol, as proposed in [11]. The stale bit marks whether committed data on a local L1 cache, pending to be propagated to main memory, is stale and thus cannot be used on a local cache access. Without the stale bit, every access to the committed data on a local L1 cache has to be treated as a miss, because the data is potentially stale.

The miss ratio for this feature is depicted in the second bar. The stale bit achieves a drastic reduction in miss ratio by allowing sharing between threads that follow each other temporally on the same core, but the miss rate is still high, because there is no locality between threads on different cores.

To further reduce the miss ratio, the Mitosis processor includes an RC. This cache works as follows: When a thread performs a store, it sends a bus request to know whether any more speculative thread has performed a load on that address in order to detect misspeculations due to memory dependence violations. Together with this request, we send the value and store it in the RC of all the threads that are more speculative and all free TUs. Thus, when a thread performs a load, the local L1 and RC are checked simultaneously. If the value requested is not in L1 but in the RC, the value is moved to L1 and supplied to the TU. This simple mechanism prevents the TUs from starting with cold caches and taking advantage of locality. The RC enables write-update coherence. Although write-update (versus write-invalidate) is less common among recent multiprocessor implementations, it is justified for two reasons: 1) the write messages piggyback messages already required in this system and 2) more importantly, in this architecture, implementing a sequential execution model, store-load communication between cores is much higher than on a system executing conventionally parallelized code.

The miss ratio for two configurations of the RC is shown in the rightmost bars. It can be observed that the use of a four-way 16-Kbyte RC significantly reduces the miss ratio to 15 percent on the average. Further reduction in the miss rate can be achieved by a larger 1-Mbyte RC.

4.4 Thread Validation

A thread finishes its execution when it reaches the starting point of any other active thread, that is, the CQIP. At this point, if the thread is nonspeculative, it validates the next thread. Otherwise, it waits until it becomes the nonspeculative one. The first thing to verify is the order. The CQIP found by the terminating thread is compared with the CQIP of the following thread in the thread order (as maintained by the order predictor). If they are not the same, then an order misspeculation has occurred, and the following thread and all its successors are squashed.

If the order is correct, the thread input values used by the speculative thread are verified. These comparisons may take some time, depending on the number of values to validate. We have observed that on the average, in our studies, thread validation results in only checking less than one memory value and about five register values. Note that only memory values produced by the slice and then consumed by the thread (values read from the SB) need to be validated when the previous thread finishes. Other memory values consumed by the thread are dynamically validated through the versioning protocol as soon as they are produced. If no misspeculations are detected, the nonspeculative thread is committed, and the next thread becomes the nonspeculative one. The TU assigned to the finished thread is freed, except when there is a child thread that is still executing the p-slice, since it may require values in the OldB.

5 Evaluation

The performance of the Mitosis processor has been evaluated through a detailed execution-driven simulation. The Mitosis processor simulator models a research Itanium CMP processor with four TUs, and it is based upon SMTSIM [31], configured for multicore execution. The main parameters considered are shown in Table 1. The numbers in the table are per TU.

The simulator executes binaries generated by our Mitosis compiler. In this paper, we present the initial results for the Olden benchmarks, which exhibit the complex interdependence patterns between prospective threads targeted by

TABLE 2
Characterization of the Olden Benchmarks

OLDEN	Spawned Threads	Thread Size	Slice Size	Slice / Thread	Thread Live-ins	Squash %
bh	422	15543	196	1.3%	4.4	0.7%
em3d	396638	422	9	2.1%	1.0	0.3%
health	198497	1112	41	3.7%	2.7	26.9%
mst	1367114	271	5	2.1%	2.3	0.8%
perimeter	493725	576	24	4.2%	3.6	1.0%
MEAN	**491279**	**3585**	**55**	**2.7%**	**2.8**	**6.0%**

these techniques but are small enough to be handled easily by our compiler. Improving the compiler infrastructure to handle larger and more complex programs (for example, SPEC) is ongoing. The benchmarks used are *bh*, *em3d*, *health*, *mst*, and *perimeter*, with an input set that, on the average, executes around 300 million instructions. The rest of the suite has not been considered due to the recursive nature of the programs. Currently, the Mitosis compiler is not able to extract speculative TLP in recursive routines. This feature will be targeted in our future work. Statistics in the next section correspond to the whole execution of the programs. Different input data sets have been used for profiling and simulation.

The Olden suite has been chosen, since automatic parallel compilers are unable to extract TLP. To corroborate this, we have compiled the Olden suite with the Intel C++ production compiler, which produces parallel code. None of the loops in these applications was parallelized by this compiler.

5.1 Performance Figures

Statistics corresponding to the characterization of the speculative threads are shown in Table 2. The last row shows the arithmetic mean for the evaluated benchmarks. The second column shows the number of spawned threads by benchmark, and the third column shows the average number of speculative instructions executed by the speculative threads. It can be observed that bh spawns the fewest number of threads, but the average size is about 30 times larger than for the rest of the benchmarks. On the other hand, mst spawns the most, but the average size is the lowest. The fourth column shows the average dynamic size of the slices, and the fifth column shows the relationship between the sizes of the speculative threads and their corresponding slice. This percentage is consistently quite low for all the studied benchmarks and, on the average, is less than 3 percent. The low slice overhead comes from three sources: the careful choice of SP-CQIP pairs, the elimination of all unnecessary computation, and the aggressive (and sometimes unsafe) minimization of the slice size by the compiler (errors in the slice are manifested as invalid live-ins, which are captured by the Mitosis hardware).

The sixth column shows the average number of thread input values that are computed by the slice; that is, on the average, it is only necessary to compute three values to execute the speculative threads. This supports our hypothesis that irregular programs contain parallel regions of code not easily detectable with conventional compiler techniques. Finally, the rightmost column represents the average percentage of squashed threads. For most of the benchmarks, this percentage is rather low, except for health, where almost one of every four threads is squashed. We have observed that for this benchmark, memory dependences for the profiling and simulated inputs are significantly different, which result in many memory dependence misspeculations.

Fig. 6 shows the speedups of the Mitosis processor over a superscalar in-order processor with about the same resources as one Mitosis TU with no speculative threading. For comparison, we also show the speedup of a more aggressive processor, with twice the amount of resources (functional units), twice the superscalar width and out-of-order issue (with no speculative threading), and a processor with a perfect first-level cache (an aggressive upper limit to the performance of helper threads that target cache misses).

It can be observed that the Mitosis processor achieves an average speedup close to 2.2 over single-threaded execution, whereas the rest of the configurations provide a much lower performance. Perfect memory achieves a speedup of just 1.23, and the more aggressive out-of-order processor only provides a 1.26 speedup. These results show that when the lack of execution resources are the bottleneck (em3d, with high instruction-level parallelism), Mitosis alleviates this problem

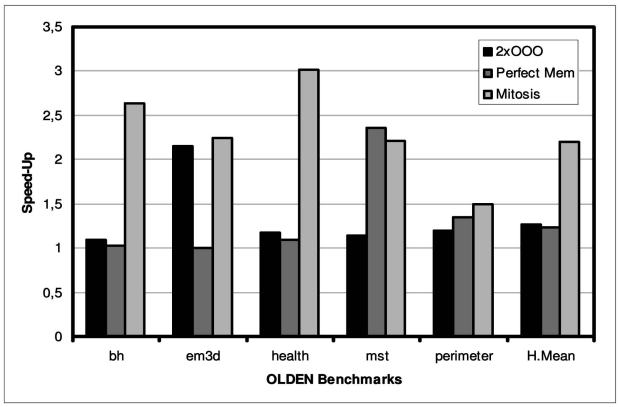
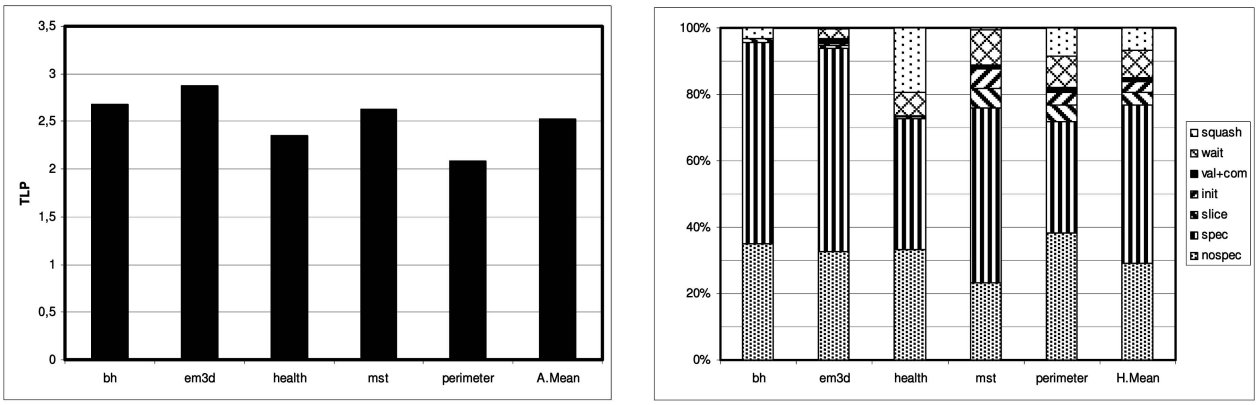


Fig. 6. Speedup over a single-threaded execution.



(a) (b)

Fig. 7. (a) Average number of active threads/cycle. (b) Time breakdown for the Mitosis processor.

as effectively as a much more complex uniprocessor by spreading the computation among multiple cores. When memory latency is the bottleneck (mst, with L1 miss ratio over 70 percent), the ability to hide memory latencies in Mitosis (speculative threads do not stall waiting for load misses on other cores) gives the performance of a perfect cache. In summary, we find Mitosis mirrors an aggressive superscalar when the ILP is high, an unattainable memory subsystem when memory parallelism is high, and outperforms both when neither ILP nor memory parallelism is high.

Fig. 7a shows the degree of speculative TLP that is exploited in the Mitosis processor. It can be observed that even though parallel compilers are unable to find TLP in these benchmarks, there is, in fact, a high degree of TLP. On the average, the number of active threads per cycle that perform useful work is around 2.5.

We have also measured that, on average, 1.2 TUs are idle due to the lack of threads. Fig. 7b shows the active time breakdown for the execution of the different benchmarks in the Mitosis processor. As expected, the most of the time the TUs are executing useful work (the sum of the nonspeculative and the speculative execution). On the average, this is nearly 80 percent of the time that the TUs are working and higher than 90 percent for bh and em3d. The overhead added by this execution model represents less than 20 percent for these benchmarks. The most significant part of it comes from the wait time. This time stands for the time that a TU has finished the execution of a speculative thread but it has to wait until becoming nonspeculative to commit. The other components of the overhead are the slice execution, initialization, validation, and commit overhead. It is worth noting that the overhead of the slices only amounts to 4 percent. Finally, the top of the bars shows the average time that TUs are executing incorrect work, that is, executing instructions for threads that are later squashed. This percentage is only 8 percent overall, mostly due to health, where the overhead is almost 20 percent. In this case, most of the squashes are due to memory violations, as previously pointed out in this section, and the cascading effect of the squashing mechanism. Recall, however, that health still maintains a three-time speedup, despite these squashes.

These results strongly validate the effectiveness of the execution model introduced in this paper (p-slice-based speculative multithreading) in meeting the Mitosis design goals: high performance, resulting from high parallelism (low wait time), high spawn accuracy (very low squash rates), and low spawn and prediction overhead (very low slice overheads).

6 Related Work

Multithreaded architectures have been studied extensively, with the primary focus on the improvement of throughput by executing several independent threads or dependent threads from parallel applications. Examples of architectural paradigms that exploit this kind of nonspeculative TLPs are multiprocessors and simultaneous multithreaded processors [31].

For speculative multithreading, pioneering works were the Expandable Split Window Paradigm [9] and the follow-up work Multiscalar processor [26]. This microarchitecture is made up of several execution units interconnected by means of a unidirectional ring. Speculative threads are created by the compiler based on several heuristics that try minimizing data dependences among threads and maximizing workload balance, among other compiler criteria [32].

Some other examples of speculative multithreaded architectures are the Superthreaded architecture [30], the Dynamic Multithreaded Processors [1], and the Clustered SpMT [15]. In those architectures, speculative threads are assigned to well-known program constructs such as loop iterations, loop continuations, and subroutine continuations.

A more generic scheme for partitioning the program into speculative threads has been recently presented [17]. This work differs in the way speculative thread inputs are predicted. Mitosis uses a software-based approach, whereas a hardware-based scheme is used in that earlier work [17]. This has important implications on the architecture and the required compiler analysis.

There has also been prior research focused on speculative multithreading for CMPs. For instance, the I-ACOMA Group [3], [4], [14], [22], the STAMPede Group [27], [28], and the Hydra Group [12], [19] have proposed compiler-based techniques to speculatively exploit TLP. In these works, loops are considered as the main source of speculative TLP.

A different approach for CMPs is taken by the Atlas multiprocessor [6]. Speculative threads are obtained by means of the MEM-slicing algorithm, which, instead of spawning threads at points of the program with high control independence, spawns threads at memory access instructions. On the other hand, Warg and Stenström [33] also used a CMP to exploit speculative TLP at the module level (procedures, functions, methods, etc.).

With regard to the implementation of multiversion structures, some proposals for register files [2], [14] and memory subsystem [3], [11] can be found in some previous work, but the execution model of the Mitosis processor requires special features not considered in those previous works. Prvulovic et al. [22] proposed some optimizations to increase the scalability of the memory subsystem that are also applicable to the Mitosis processor.

Finally, it is well known that the way interthread data dependences are managed strongly affects the performance of speculative multithreaded processors [15]. Basically, two mechanisms have been studied previously: synchronization and value prediction. Synchronization mechanisms include all those mechanisms where an interthread-dependent instruction has to wait for the computation of the value at the producer thread. There are techniques to reduce these long latencies, such as code reordering [30], identifying the last writer [26], etc. Compared to these techniques, Mitosis can benefit from aggressive speculative optimizations that can reduce the size of the p-slice over the prologue of the Superthreaded or to have the values available earlier than Multiscalar.

Value prediction of dependences can significantly increase parallelism, especially for register values that are quite predictable [16], but memory values are harder to predict [4], [28]. Speculative multithreading architectures are particularly sensitive to value prediction accuracy, because multiple values must typically be predicted correctly for the thread to be useful. In contrast to the hardware approach, Mitosis uses p-slices to predict in the software the interthread data dependent values. This scheme improves on hardware value prediction, since the prediction is potentially more accurate, since the computation of these values is directly derived from the original code. Additionally, it can encapsulate multiple control flows that contribute to the computation of the live-ins. Some hardware value predictors can also do that [16], but they require additional hardware to predict the control flow of the parent thread. Another usage model of p-slices in the area of speculative multithreading has been recently proposed for recovering from interthread dependence violations [25].

The Pinot group [21] has proposed a speculative multithreading architecture with a generic thread partition scheme that is able to exploit TLS over a wide range of granularities as Mitosis does. However, the Pinot architecture does not support out-of-order spawning, and interthread data dependences are handled through code reordering, without including any support for hardware or software prediction.

The use of Helper Threads to reduce the latency of high-cost instructions has been thoroughly studied [7], [24], [34]. This research borrows some high-level concepts from that body of work to create the p-slices for thread live-ins. However, the need of Mitosis to precompute a set of values accurately (as opposed to a single load address or branch result), as well as an increased cost of misspeculation, requires significantly more careful creation of slices, and the inclusion of a more accurate control flow in the slice (previous work on helper threads typically followed only a single control flow path in a slice) makes it quite different from other models.

Finally, Zilles and Sohi present a different scheme to exploit speculative TLP by means of distilled programs [35]. A distilled program executes a small subset of the instructions of a given program to compute the input values of all speculative threads. In that execution model, the distilled program runs as a master thread. When all the input values for a speculative thread are computed, it is spawned on an idle context, whereas the master starts computing new input values for the next thread. The Mitosis execution model differs from that previous work in the fact that the computation of the thread live-in values are done by the speculative threads themselves, which allows the processor to spawn threads out of the program order and to often compute the live-ins for many speculative threads in parallel.

7 Conclusions

In this work, we have presented and evaluated the Mitosis processor, which exploits speculative TLP through a novel software scheme to predict and manage interthread data dependences that leverages the original code to compute thread live-ins. It does so by inserting a piece of code in the binary that speculatively computes the values at the starting point of the speculative thread. This code, referred to as a p-slice, is built from a subset of the code after the spawn instruction and before the beginning of the speculative thread. A key feature of the Mitosis processor is that p-slices do not need to be correct, which allows the compiler to use aggressive optimizations when generating them, to keep the p-slice overhead low.

The key microarchitecture components of the Mitosis processor have been presented: 1) hardware support for the spawning, execution, and validation of p-slices allows the compiler to create slices with minimal overhead, 2) a novel multiversion register file organization supports a unified global register view, multiple versions of register values, transparent communication of register dependences across processor cores, all with no significant latency increase over traditional register files, and 3) the memory system supports multiple versions of memory values and introduces a novel architecture that pushes values toward threads that are executing future code to effectively mimic the temporal locality available to a single-threaded processor with a single cache.

Finally, the results obtained by the Mitosis processor with four TUs for a subset of the Olden benchmarks show a significant performance potential for this architecture. It outperforms the single-threaded execution by 2.2 times and provides more than a 1.75 speedup over a double-sized out-of-order processor. A similar speedup was achieved over a processor with perfect memory. These results confirm that there are large amounts of available TLP for code that is resistant to conventional parallelism techniques. However, this parallelism requires highly accurate dependence prediction and efficient data communication between threads, as provided by the Mitosis architecture.

ACKNOWLEDGMENTS

This work was done while D.M. Tullsen was on a sabbatical leave and was with the Interl-UPC Barcelona Research Center (IBRC).

REFERENCES

[1] H. Akkary and M.A. Driscoll, "A Dynamic Multithreading Processor," *Proc. 31st IEEE/ACM Int'l Symp. Microarchitecture (MICRO),* 1998.

[2] S. Breach, T.N. Vijaykumar, and G.S. Sohi, "The Anatomy of the Register File in a Multiscalar Processor," *Proc. 25th IEEE/ACM Int'l Symp. Microarchitecture (MICRO '94),* pp. 181-190, 1994.

[3] M. Cintra, J.F. Martinez, and J. Torrellas, "Architectural Support for Scalable Speculative Parallelization in Shared-Memory Systems," *Proc. 27th IEEE/ACM Int'l Symp. Microarchitecture (MICRO),* 2000.

[4] M. Cintra and J. Torrellas, "Eliminating Squashes through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors," *Proc. Eighth Int'l Symp. High-Performance Computer Architecture (HPCA),* 2002.

[5] R.S. Chappel, J. Stark, S.P. Kim, S.K. Reinhardt, and Y.N. Patt, "Simultaneous Subordinate Microthreading (SSMT)," *Proc. 26th Int'l Symp. Computer Architecture (ISCA '99),* pp. 186-195, 1999.

[6] L. Codrescu and D. Wills, "On Dynamic Speculative Thread Partitioning and the MEM-Slicing Algorithm," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '99),* pp. 40-46, 1999.

[7] J.D. Collins, H. Wang, D.M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J.P. Shen, "Speculative Precomputation: Long Range Prefetching of Delinquent Loads," *Proc. 28th Int'l Symp. Computer Architecture (ISCA),* 2001.

[8] K. Diekendorff, *Compaq Chooses SMT for Alpha,* microprocessor report, Dec. 1999.

[9] M. Franklin and G.S. Sohi, "The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism," *Proc. 19th Int'l Symp. Computer Architecture (ISCA '92),* pp. 58-67, 1992.

[10] C. Garcia, C. Madriles, J. Sanchez, P. Marcuello, A. Gonzalez, and D.M. Tullsen, "Mitosis Compiler: n Infrastructure for Speculative Threading Based on Pre-Computation Slices," *Proc. ACM Conf. Programming Language Design and Implementation (PLDI '05),* June 2005.

[11] S. Gopal, T.N. Vijaykumar, J.E. Smith, and G.S. Sohi, "Speculative Versioning Cache," *Proc. Fourth Int'l Symp. High-Performance Computer Architecture (HPCA),* 1998.

[12] L. Hammond, M. Willey, and K. Olukotun, "Data Speculation Support for a Chip Multiprocessor," *Proc. Eighth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* 1998.

[13] http://ipf-orc/sourceforge.net, 2007.

[14] V. Krishnan and J. Torrellas, "Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor," *Proc. Int'l Conf. Supercomputing (ICS '98),* pp. 85-92, 1998.

[15] P. Marcuello, "Speculative Multithreaded Processors," PhD dissertation, Universitat Politecnica de Catalunya, 2003.

[16] P. Marcuello, J. Tubella, and A. González, "Value Prediction for Speculative Multithreaded Architectures," *Proc. 32nd Int'l Conf. Microarchitecture (MICRO '99),* pp. 203-236, 1999.

[17] P. Marcuello and A. González, "Thread-Spawning Schemes for Speculative Multithreaded Architectures," *Proc. Eighth Int'l Symp. High-Performance Computer Architecture (HPCA),* 2002.

[18] T. Marr et al., "Hyperthreading Technology Architecture and Microarchitecture," *Intel Technology J.,* vol. 6, no. 1, 2002.

[19] J. Oplinger et al., "Software and Hardware for Exploiting Speculative Parallelism in Multiprocessors," Technical Report CSL-TR-97-715, Stanford Univ., 1997.

[20] A. Mendelson et al., "CMP Implementation in the Intel Core Duo Processor," *Intel Technology J.,* vol. 10, no. 2, 2006.

[21] T. Ohsawa, M. Takagi, S. Kawahara, and S. Matsushita, "Pinot: Speculative Muti-threading Processor Architecture Exploiting Parallelism over a wide Range of Granularities," *Proc. 38th Int'l Symp. Microarchitecture (MICRO),* 2005.

[22] M. Prvulovic, M.J. Garzarán, L. Rauchwerger, and J. Torrellas, "Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization," *Proc. 28th Int'l Symp. Computer Architecture (ISCA),* 2001.

[23] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas, "Tasking with Out-of-Order Spawn in TLS Chip Multiprocessors: Microarchitecture and Compilation," *Proc. 19th ACM Int'l Conf. Supercomputing (ICS),* 2005.

[24] A. Roth and G.S. Sohi, "Speculative Data-Driven Multithreading," *Proc. Seventh Int'l Symp. High-Performance Computer Architecture (HPCA '01),* pp. 37-48, 2001.

[25] S.R. Sarangi, W. Liu, J. Torrellas, and Y. Zhou, "ReSlice: Selective Re-Execution of Long-Retired Misspeculated Instructions Using Forward Slicing," *Proc. 38th Int'l Symp. Microarchitecture (MICRO),* 2005.

[26] G.S. Sohi, S.E. Breach, and T.N. Vijaykumar, "Multiscalar Processors," *Proc. 22nd Int'l Symp. Computer Architecture (ISCA '95),* pp. 414-425, 1995.

[27] J. Steffan and T. Mowry, "The Potential of Using Thread-Level Data Speculation to Facilitate Automatic Parallelization," *Proc. Fourth Int'l Symp. High-Performance Computer Architecture (HPCA '98),* pp. 2-13, 1998.

[28] J. Steffan, C. Colohan, A. Zhai, and T. Mowry, "Improving Value Communication for Thread-Level Speculation," *Proc. Eighth Int'l Symp. High-Performance Computer Architecture (HPCA '98),* pp. 58-62, 1998.

[29] S. Storino and D.J. Borkenhagen, "A Multithreaded 64-bit PowerPC Commercial RISC Processor Design," *Proc. 11th Int'l Conf. High-Performance Chips,* 1999.

[30] J.Y. Tsai and P.-C. Yew, "The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT),* 1995.

[31] D.M. Tullsen, S.J. Eggers, and H.M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. 22nd Int'l Symp. Computer Architecture (ISCA '95),* pp. 392-403, 1995.

[32] T.N. Vijaykumar, "Compiling for the Multiscalar Architecture," PhD dissertation, Univ. of Wisconsin, Madison, 1998.

[33] F. Warg and P. Stenström, "Limits on Speculative Module-Level Parallelism in Imperative and Object-Oriented Programs on CMP Platforms," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT),* 2001.

[34] C.B. Zilles and G.S. Sohi, "Execution-Based Prediction Using Speculative Slices," *Proc. 28th Int'l Symp. Computer Architecture (ISCA),* 2001.

[35] C.B. Zilles and G.S. Sohi, "Master/Slave Speculative Parallelization," *Proc. 35th Int'l Symp. Microarchitecture (MICRO),* 2002.

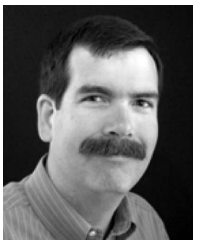
Carlos Madriles received the MS degree in computer engineering in 2002 from the Universitat Politècnica de Catalunya (UPC), Barcelona, where he is currently working toward the PhD degree in speculative thread-level parallelism. He joined the Department of Computer Architecture, UPC, in 2001 as a research assistant. Since May 2002, he has been a research scientist at the Intel-UPC Barcelona Research Center. His research interests include multicore architectures and compilation techniques, in particular speculative multithreading and transactional memory.

Carlos García-Quiñones received the MS degree in computer science in 2003 from the Universitat Politècnica de Catalunya (UPC), Barcelona. From 2002 to 2006, he was with the Intel-UPC Barcelona Research Center as a researcher in computer architecture, in particular compilers for speculative architectures. His research interests include massively distributed computation and reconfigurable hardware.

Jesús Sánchez received the MS and PhD degrees in computer engineering from the Universitat Politècnica de Catalunya (UPC), Barcelona, in 1995 and 2001, respectively. He joined the Department of Computer Architecture, UPC, in 1995 as a research assistant and was an assistant professor from 1998 to 2002. Since March 2002, he has been with the Intel-UPC Barcelona Research Center, which he joined as a senior research scientist. His interests include processor microarchitecture and compilation techniques, in particular memory hierarchy, instruction-level parallelism, clustered architectures, instruction scheduling, and speculative multithreading. He has more than 25 publications on these topics. He is currently working on speculative multithreading techniques and FPGA-based prototypes. He is a member of the IEEE.

Pedro Marcuello received the bachelor's and PhD degrees in computer science from the Universitat Politècnica de Catalunya (UPC), Barcelona, in 1995 and 2003, respectively. Since 2003, he has been with the Intel-UPC Barcelona Research Center as a research scientist. From 1997 to 2003, he was with the Department of Computer Architecture, UPC, as a full-time teaching assistant. His research interests include speculative thread-level parallelism and multicore architectures. He is a member of the IEEE Computer Society.

Antonio González received the MS and PhD degrees from the Universitat Politècnica de Catalunya (UPC), Barcelona. He joined the faculty of the Department of Computer Architecture, UPC, in 1986 and became a full professor in 2002. He is the founding director of the Intel-UPC Barcelona Research Center, which started in 2002 and whose research focuses on new microarchitecture paradigms and code generation techniques for future microprocessors. He has given over more than invited talks, is the holder of more than 40 patents, and has advised 13 PhD dissertations in the areas of computer architecture and compilers. He is an associate editor for the *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, *ACM Transactions on Architecture and Code Optimization*, and *Journal of Embedded Computing*. He has served on more than 100 program committees for international symposia in the field of computer architecture, including the International Symposium on Computer Architecture (ISCA), Annual International Symposium on Microarchitecture (MICRO), International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), IEEE International Symposium on High Performance Computer Architecture (HPCA), International Conference on Parallel Architectures and Compilation Techniques (PACT), ACM International Conference on Supercomputing (ICS), IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), and International Parallel and Distributed Processing Symposium (IPDPS). He has been the program chair or cochair of ICS 2003, ISPASS 2003, MICRO 2004, and HPCA 2008, among other symposia. He has published more than 250 papers. He is a member of the IEEE.



Dean M. Tullsen received the BS and MS degrees from the University of California, Los Angeles (UCLA) and the PhD degree in August 1996 from the University of Washington. He spent four years as a computer architect at AT&T Computer Systems/AT&T Bell Labs and a year teaching in the Department of Computer Science, Shantou University, Shantou, China. He is currently a professor in the Department of Computer Science and Engineering, University of California, San Diego. His research interests include computer architecture. He is a senior member of the IEEE.

Hong Wang received the PhD degree in electrical engineering from the University of Rhode Island in 1996. He joined Intel in 1995, where he is currently a senior principal engineer and the director of the Microarchitecture Research Labs (uAL). His work involves research on future processor architecture and microarchitecture. He is a member of the IEEE.

John P. Shen received the BS degree in electrical engineering from the University of Michigan and the MS and PhD degrees in electrical engineering from the University of Southern California. He is the head of the Nokia Research Center, Nokia, Palo Alto, which is a newly established research laboratory focusing on mobile Internet systems, applications, and services. Prior to joining Nokia in 2006, he was the director of the Microarchitecture Research Labs, Intel Corp., which was responsible for developing processor and system architecture innovations. Prior to joining Intel in 2000, he was a professor at Carnegie Mellon University, where he supervised a total of 17 PhD students and numerous MS students and received multiple teaching awards. He is currently an adjunct professor at Carnegie Mellon West. He is the author or coauthor of numerous published articles and books, including *Modern Processor Design: Fundamentals of Superscalar Processors* (McGraw-Hill, 2005). He is a fellow of the IEEE.