Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors*

Marcelo Cintra, José F. Martínez, and Josep Torrellas
Department of Computer Science
University of Illinois at Urbana-Champaign
{cintra,martinez,torrellas}@cs.uiuc.edu

ABSTRACT

Speculative parallelization aggressively executes in parallel codes that cannot be fully parallelized by the compiler. Past proposals of hardware schemes have mostly focused on single-chip multiprocessors (CMPs), whose effectiveness is necessarily limited by their small size. Very few schemes have attempted this technique in the context of scalable shared-memory systems.

In this paper, we present and evaluate a new hardware scheme for scalable speculative parallelization. This design needs relatively simple hardware and is efficiently integrated into a cache-coherent NUMA system. We have designed the scheme in a hierarchical manner that largely abstracts away the internals of the node. We effectively utilize a speculative CMP as the building block for our scheme.

Simulations show that the architecture proposed delivers good speedups at a modest hardware cost. For a set of important non-analyzable scientific loops, we report average speedups of 4.2 for 16 processors. We show that support for per-word speculative state is required by our applications, or else the performance suffers greatly.

1 INTRODUCTION

Despite advances in compiler technology [2, 6], there is still a large set of codes that compilers fail to parallelize to an acceptable degree. Complex data dependence structures caused by non-linear subscripts, double indirections, pointers, or function calls within code sections often lead the compiler to conservatively abstain from parallelizing the code. Many of these codes, particularly in the scientific domain, may still turn out to have a large amount of parallelism.

Software transformations are a possible way of extracting some parallelism from these codes. Some software schemes analyze the dependence structure of the code at run time and try to run parts of it in parallel protected by synchronization (for example [13]). Other software schemes speculatively run the code in parallel and later recover if a dependence violation is detected [5, 15]. While these techniques are certainly promising, they all have various amounts of software overhead, which may limit their scalability.

On the hardware side, there have been several proposals for single-chip speculative multithreaded or multiprocessor architectures [4, 7, 11, 14, 16, 17, 20]. In these systems, the hardware detects dependence violations across threads at run time. The code is speculatively run in parallel and, when a violation is detected, a corrective action is taken that involves thread squash and parallel execution resumption. Already companies like Intel and Sun are seriously involved in on-chip speculative parallelization [1, 19]; the latter has even announced a chip multiprocessor (CMP) with such support (*MAJC*). Most of these small-scale designs are conceived for standalone operation, and are thus not tailored at being integrated into large system configurations. Only [17] is designed for multichip configurations.

One possible way to address this limitation is to extend a scalable cache coherence protocol to support speculative parallelization. There have been two proposals in this direction [17, 18, 22, 23, 24]. Both schemes extend an invalidation-based cache coherence protocol. They both yield a flat view of their speculation threads. Neither of these proposals is fleshed out enough to show how, if speculative CMPs were used as building blocks, it would reconcile its single layer protocol with many of the self-contained speculation protocols of these CMPs.

The main contribution of our paper is the design and evaluation of a new scheme for scalable speculative parallelization that requires relatively simple hardware and is efficiently integrated next to the cache coherence protocol of a conventional NUMA multiprocessor. We have taken a hierarchical approach that largely abstracts away the internals of the node architecture. In particular, we are able to utilize a self-contained speculative CMP as building block, with minimal additions to interface with the rest of the system. The integration of speculative CMPs into scalable systems seems to offer great potential.

Simulations show that the architecture that we propose delivers good speedups at a modest hardware cost. For a set of important non-analyzable scientific loops, we report average speedups of 4.2 for 16 processors. We also show that support for per-word speculative state is required by our applications, or otherwise the performance suffers greatly.

This paper is organized as follows: Section 2 introduces speculative parallelization and the base speculative CMP that we use as building block; Section 3 describes our scalable scheme built out of speculative CMPs; Sections 4 and 5 present the experimental setup and the evaluation of the scheme, respectively; Section 6 analyzes related work; finally, Section 7 concludes the paper.

2 SPECULATIVE PARALLELIZATION

2.1 Basic Concepts

Speculative parallelization extracts threads from sequential code and runs them in parallel, hoping not to violate any sequential semantics. The control flow of the sequential code imposes an order on the threads and, therefore, we can use the terms predecessor and successor to qualify the relation between any given pair of threads. The control flow also yields a data dependence relation on the memory operations. In loop-level speculative parallelization, threads are typically formed every some number of consecutive iterations, and the thread order is total. In general, loop-level speculative parallelization is mostly concerned with not violating cross-thread data dependences. Of course, in loops whose upper limit is not known,

*This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP-9457436, ASC-9612099, MIP-9619351, and CCR-9970488, DARPA Contract DABT63-95-C-0097, and gifts from IBM and Intel.

2000 ACM Intl. Symp. on Computer Architecture.

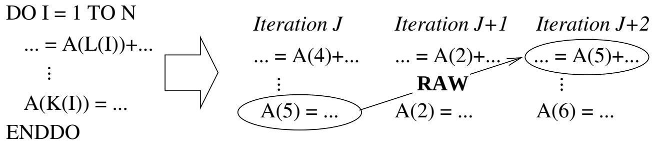

Figure 1: Example of speculative parallel execution with a RAW dependence violation.

		Word 0								Word 1							
Valid	Tag	L0	L1	L2	L3	S0	S1	S2	S3	L0	L1	L2	L3	S0	S1	S2	S3
1	0x2234	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0x0000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 2: Organization of the MDT. In this example, memory lines have 2 words, there are 4 processors, and processors 2 and 1 have performed a load and a store respectively on word 0 of line 0x2234.

speculative parallelization should also be concerned about not violating control flow. This is because the exact number of threads needed to cover the actual number of iterations must be produced. In what follows, we will focus on the aspects of speculative parallelization related to cross-thread data dependences.

A memory operation issued by a thread is *speculative* when it may have data dependences with others issued, or to be issued, by predecessor threads; otherwise it is *non-speculative*. Likewise, a thread is speculative when it has issued, or may issue, speculative memory operations. If all operations issued or to be issued by a thread are known to be non-speculative, the thread itself is non-speculative.

A thread is said to retire or *commit* when it has finished its execution and it is non-speculative. In general, a speculative thread will become non-speculative only when all its predecessors have committed, which guarantees that all its memory operations are non-speculative. If a thread reaches its end and is still speculative, it must wait to acquire non-speculative status in order to commit. Hence, the thread sequence can be split into two consecutive segments of committed and uncommitted threads. Among the uncommitted threads, only the earliest one is non-speculative.

Speculative stores generate speculative *versions*, and speculative loads that do not find a local version try to get it from the closest predecessor that holds one. If no speculative version exists, the load fetches the non-speculative one.

As speculative threads execute in parallel, the system must track memory references to identify any cross-thread data dependence violation. WAR and WAW dependence violations do not induce errors in systems that support multiple versions for the same data. RAW dependence violations, however, typically cause problems. A RAW dependence violation occurs whenever a speculative thread has loaded a version of data that is subsequently modified by a predecessor. We then say that the load was premature, since it tried to consume a value that had not yet been produced. Figure 1 shows an example of such a violation. When a RAW dependence violation is detected, the thread that performed the premature load must be *squashed*. Ordinarily, all its successors are also squashed at this time because they may have consumed versions generated by the squashed thread. While it is possible to resolve reference chains and selectively squash only threads at fault, it involves extra complexity. In any case, when a thread is squashed, all the data produced speculatively must be purged. Then, the thread restarts execution from the beginning.

The approach that we use for speculative parallelization within a chip is that of the Memory Disambiguation Table (MDT) scheme. In the following sections, we give an overview of the speculation protocol used in a MDT-based CMP. Further details can be found in [11].

2.2 Speculative CMP and the MDT

In a MDT-based CMP, each processor has a private L1 cache that can hold speculative versions of data. The memory beyond the chip is accessible by all processors and does not see any such speculative versions. All updates that a speculative thread makes must remain in its L1 cache until the thread becomes non-speculative.

When a thread reaches the end of its execution, the processor on which it runs stalls until the thread becomes non-speculative. Then, the thread commits by writing back all the dirty data in its L1, therefore leaving the L1 cache consistent with memory for a future thread. At this point, a new, speculative thread can start. The decisions to stall the processor until the thread becomes non-speculative, and not to keep dirty data in L1 when a speculative thread is spawn, were made to simplify the hardware and general complexity of the speculation protocol in the chip. With this design, if the threads have load imbalance, a thread may finish some cycles before its predecessor has finished and committed. During this time, the processor that was running the thread remains idle. We call this time *intrinsic chip imbalance*.

This scheme includes a multi-ported table called MDT that records speculative accesses to data. Specifically, as memory lines are being accessed, the MDT allocates a *Load* and a *Store* bit per word in the line and per processor. The Store bit is set whenever a processor writes to a word, while the Load bit is set only if the processor reads a word without first writing to it. Later, when another processor loads a word, the Store bits are used to identify the most up-to-date version among the predecessors. On the other hand, when a store operation is performed, both bits are used to detect premature loads by successor threads.

The MDT is placed between the L1 and memory. It is organized as an 8-way set-associative structure that maintains state corresponding to a number of memory lines (Figure 2). In a naive design, the table would be searched every time that a memory operation occurs to data that can be speculatively accessed. If an entry matching the line requested is found, the corresponding bit is updated; otherwise a new entry is created.

The MDT is of limited size and cannot evict entries while in use. A speculative thread stalls if it finds the MDT full upon trying to allocate a new entry. When a thread commits, all its MDT bits are cleared. If, as a result, all the bits in one MDT entry become zero, the entry is deallocated.

As said before, to simplify the protocol, the scheme does not allow dirty data to remain in L1 across speculative thread initiations. This restriction, which could be eliminated with further support, implies that a thread has to write back all its L1 dirty data when it commits. This is accomplished gradually in two steps. First, when the thread becomes non-speculative, its L1 is switched to work in a write-through mode. This support additionally implies that the non-speculative thread does not need to set any MDT bits and, as a result, cannot stall due to lack of free MDT entries – which means that the code is guaranteed to make progress. Second, when the thread finishes, it writes back all the remaining dirty words.

The actual load and store operations proceed as follows. When a thread issues a load, whether it hits or misses in L1, the MDT is informed. If the access resulted in a L1 miss, the line is fetched from memory in parallel. Meanwhile, the MDT Store bits of its predecessors are checked in reverse order to search for the closest version. If a non-zero Store bit is found in the MDT, the word is read from the predecessor (Figure 3a) and merged with the rest of the line in the requestor's L1. Note that we must check the Store and not the Load bits. A set Load bit does not guarantee that the corresponding L1 has the word: lines that have only been speculatively loaded can be displaced.

When a thread issues a store, the MDT Load and Store bits of its successors are scanned, starting from the immediate successor. The search terminates when a successor is found with either bit set. If the Load bit is set, a RAW dependence has been violated: the consumer and its successors are squashed, their speculatively updated words invalidated, and their MDT state cleared. If, instead, the Load bit is not set but the Store bit is, a WAW dependence has been violated. However, this is harmless because the system correctly supports multiple versions, thus no squashing occurs. In all cases, however, we send an invalidation message to the caches of the threads between the one issuing the store and the first successor with the Load or Store bits set (non-inclusive), as well as to the

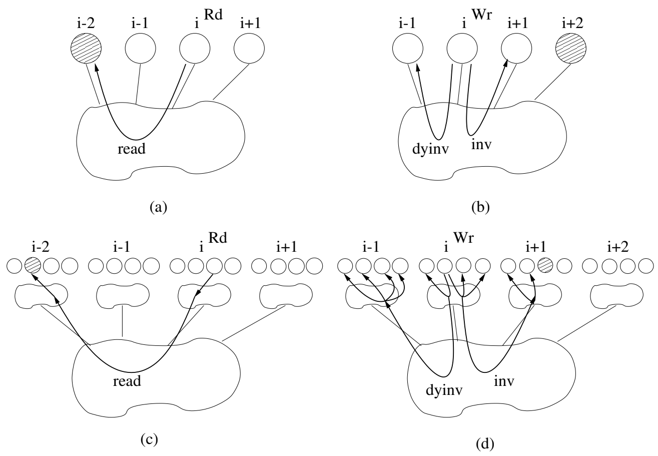

Figure 3: Handling reads and writes in a single CMP ((a) and (b)) and in a hierarchical scalable system ((c) and (d)). In a read, the shaded node is the closest predecessor that wrote the word while, in a write, it is the closest successor that redefined the word before reading it.

caches of any squashed threads. This invalidation ensures that the obsolete word is not used in successor caches. Furthermore, in all cases, we also send a *delayed-invalidation* message (*dyinv*) to the caches of all the predecessor threads. This message marks the data as stale for any new thread that is later started on those processors. Such new threads will necessarily be successors to the writing one (Figure 3b).

In addition to MDT state, this scheme also needs, for each L1 line, one *Dirty*, *Stale*, and *Forwarded* bit, plus per-word *Invalid*, *Safe Load*, and *Safe Store* bits[1]. The Dirty bit is set when a processor writes to any one of the words in the line, while the Invalid bit is set when the word is invalidated. The Safe Load and Safe Store bits are used to avoid having to access the MDT on every load and store. If all loads and stores had to be made visible to the MDT, we would in practice render the L1 caches read- and write-through, causing much traffic on the bus. Consequently, we use the Safe Load and Safe Store bits to indicate whether it is possible to perform the operation in the L1 cache without checking the MDT. The Safe Load bit is set by the thread's first load to the word, so that subsequent loads to the word by the same thread do not need to inform the MDT. Similarly, both the Safe Load and Safe Store bits are set by the thread's first store to the word, so that subsequent loads or stores to the word by the same thread do not need to check the MDT. However, the Safe Store bit in a thread is reset when a successor reads the thread's version of the word. This way, when the thread later performs a store to the word again, it will correctly check the MDT and squash the successor. A load that finds the Safe Load bit unset and a store that finds the Safe Store bit unset are said to be *exposed*.

The Stale bit is set for a L1 line when the cache receives a delayed-invalidation message for one of the words in the line: although still usable by the current thread, the word has become obsolete for future threads and, therefore, must be invalidated before a new thread starts. Consequently, at commit time, after the thread has written back to memory all the valid words in dirty L1 lines, we set the Invalid bit of all the words in lines with the Stale bit set. Then, we reset the Stale bit.

Finally, the Forwarded bit identifies L1 lines with words forwarded from a predecessor, as opposed to loaded from memory. The Forwarded bit is used when a thread is squashed. In this case, all the words in L1 lines with the Forwarded bit set are invalidated to prevent keeping erroneous versions in the cache. The Forwarded bit is cleared at commit time.

[1] In the paper that introduced the MDT [11], we called the Stale, Safe Load, and Safe Store bits the *Flush*, *Safe Read*, and *Safe Write* bits, respectively. We feel that the new names are better. That paper also used the Stale and Forwarded bits on a per-word basis. Subsequent performance evaluation has shown that these bits can be kept on a per-line basis without any noticeable performance degradation.

Dep.	*Order*	*First Access*	*Second Access*
WAW	In Order	PRE generates version Vp.	SUC generates version Vs; mark Vp stale at PRE (send dyinv). When PRE commits, write Vp back to memory and self-invalidate it.
	Out of Order	SUC generates version Vs.	PRE generates version Vp; mark it stale. When PRE commits, write Vp back to memory and self-invalidate it.
WAR	In Order	PRE reads most up-to-date version Vp.	SUC generates version Vs; mark Vp stale at PRE (send dyinv). When PRE commits, self-invalidate Vp.
	Out of Order	SUC generates version Vs.	PRE reads most recent version Vp; mark it stale. When PRE commits, self-invalidate Vp.
RAW	In Order	PRE generates version Vp.	SUC reads most recent version Vp.
	Out of Order	SUC reads most up-to-date version Vs.	PRE generates version Vp; invalidate Vs and squash SUC.

Table 1: Handling all types of data dependences due to accesses to a single word. In the table, *PRE* and *SUC* stand for predecessor and successor thread respectively, and V_p and V_s refer to two versions of the same word created by the predecessor and successor thread respectively.

2.3 Multiple Versions and Per-Word State

The protocol described allows different versions of the same datum to reside in different caches simultaneously, and keeps per-word Load and Store bits in the MDT. Both features are targeted at tolerating situations that would otherwise cause unnecessary squashes. As a result, in our protocol, only out-of-order RAW dependences involving the same word create squashes. Table 1 shows the operations performed by our speculation protocol in the presence of every possible data dependence. To gain further insight, let us now consider other protocols where one of the two supports is missing and see what additional dependences cause squashes.

In a protocol where per-word speculative state is supported but multiple versions are not, out-of-order WAR and WAW dependences also cause thread squashes. Indeed, in these cases, a thread creates a version of a datum and, later, a predecessor tries to use (WAR) or create (WAW) an older version of the same datum. If only one speculative version were supported by the system, the youngest of the two threads would have to be squashed to accommodate the older version that its predecessor is trying to access. If, however, the system can host both versions simultaneously in such a way that each thread manipulates its own copy of the datum, the squash operation becomes unnecessary. Needless to say, the system must be able to reconcile these multiple versions into a meaningful final state at the end of the execution.

Consider now a protocol where multiple versions are supported but per-word speculative state is not. In this case, the system cannot determine whether two accesses to the same cache line target the same or different words. Specifically, all out-of-order RAW and WAW accesses to a cache line must now trigger squashes. For example, consider that a thread reads a datum and, later on, a predecessor writes to the same memory line. The system regards it as a RAW dependence violation and squashes the successor. Consider, instead, that the first thread writes a datum and, later on, the predecessor writes to that line (WAW). Since the system cannot eventually combine the line versions, it squashes the successor.

To see that both types of support are indeed useful, Table 2 shows whether they are necessary in five important data access patterns. Each pattern is illustrated with an example in which several contiguous 4-word memory lines are accessed by three loop iterations (*i0*, *i1*, and *i2*). Accesses are represented by arrows: straight arrows indicate that we can foresee what accesses the iterations will be performing with some accuracy; curved arrows mean that the locations being accessed cannot be predicted in general. Each pattern lists the

Access Pattern	*Multiple Versions*	*Per-Word State*	*Appl.*
1 Random, unpredictable accesses, clustered to some extent. E.g. small arrays with random accesses, arrays with somewhat sequential irregular patterns.	Yes	Yes	Track DSMC3D
2 Important subcase of case 1. Iterations often write and then read a word. Different iterations may access the same word or not. E.g. privatizable or quasi-privatizable data.	Yes	Yes	APSI BDNA
3 Random, sparse access pattern.	No, but may suffer occasional squashes if more than 1 access/line		DSMC3D Euler
4 Predictable accesses in a regular, monotonic manner. E.g. strided array access.	No, but may need alignment and unrolling to group all accesses to the same line.		APSI
5 All iterations read and/or write to a single location (the compiler guarantees such behavior) E.g. privatizable or shared data.	Yes	No	Not observed

Table 2: Access patterns and the protocol support required.

applications from Section 4.1 in which we found instances of it.

From the table, we see that the most general case, namely 1, in which nothing is known about the accesses, requires both types of support. Case 2, a very important subcase of 1, corresponds to loops that can be fully (or to a large degree) parallelized through privatization – although the compiler is unable to determine this. This case also needs both types of support.

When accesses are very sparse (case 3), the system can probably do without these two types of support, since access collisions are occasional. Case 4 can also do without them provided that the iterations access different cache lines, which would typically be achieved through smart data alignment and loop unrolling. However, if different iterations do access the same cache lines, then both types of support may be needed.

Finally, case 5 does not need to hold per-word state if the compiler guarantees that no more than one word in the line is touched. It does need support for multiple versions. This corresponds to a shared variable, or a variable that looks privatizable except that some accesses are inside an `if` statement. Overall, based on all this discussion, we see that the two supports are very useful for many types of patterns.

3 SCALABLE SPECULATION

Supporting speculative parallelization in a scalable multiprocessor is more challenging than in a CMP. The speculative state is necessarily distributed across the nodes. We need a distributed speculative protocol that works over a general interconnect and does not adversely affect conventional cache-coherence support. To this end, our scheme for speculative parallelization is designed to operate largely in parallel with a conventional invalidation-based cache-coherence protocol and directory structure of a CC-NUMA system. Furthermore, we choose simplicity over all-out performance in many aspects of the design. Finally, we want the resulting system to be flexible enough to use a speculative CMP as building block. Speculative CMPs are specially attractive since such chips already provide some support for speculation and using them at each node may leverage costs.

This section describes the following aspects of our scheme: the hierarchical approach that we take (Section 3.1), different aspects of the protocol operation (Section 3.2), the interaction with the CC-NUMA directory (Section 3.3), the support for multiprogramming (Section 3.4), the overall benefits of the approach (Section 3.5) and, briefly, the compiler support (Section 3.6).

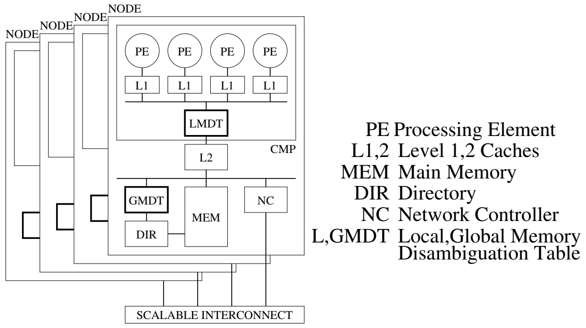

Figure 4: Hierarchical organization for speculative parallelization.

3.1 Hierarchical Organization

To build a flexible scheme, we design our speculative parallelization protocol in a hierarchical manner. Specifically, we add one new MDT in the machine, for which each speculative CMP node is abstracted away as a single entity. This new MDT relates to the L2 caches in the nodes in a similar way as, within a CMP, the on-chip MDT relates to the L1 caches. Internally, each speculative CMP continues to function largely like before. The on-chip MDTs are now called *Local* MDTs (LMDTs), while the external MDT is called *Global* MDT (GMDT). The GMDT is coupled with the directory of the CC-NUMA machine and, like such, it is physically distributed across the different nodes of the machine based on address ranges. Consequently, we will use the term *home* GMDT to refer to the section of the GMDT in the home node of the data. Overall, this organization stores speculative state like a clustered CC-NUMA holds coherence state: per-processor state within the node, and per-node state across the system. Figure 4 shows a block diagram of the envisioned organization.

A hierarchical approach allows us to abstract away the internals of the speculative CMP and construct a system-wide speculative protocol that can work with different classes of nodes. Although the system can easily adapt to many different types of nodes including single processors, CMPs, or SMP clusters, in this work we focus on using speculative CMPs as building blocks. With this approach, we hope to leverage emerging speculative CMP technology like the one commercialized by Sun Microsystems in the MAJC chip [19]. Consequently, in our design, we try to minimize the modifications required to a speculative CMP like the one presented in Section 2.2 to incorporate it into a scalable system.

If we use CMPs as nodes and want the GMDT to keep state on a per-node basis, it is simplest if the threads are assigned to chips in batches, where a batch or *chunk* contains a set of consecutive threads. This way, the threads running on a speculative CMP can be made to look to the external system as a single execution thread. Off-chip predecessors and successors of a given thread will also be predecessors and successors of the thread's on-chip peers. This approach allows each chip to operate as a largely independent speculative system and reduces to a minimum the modifications required to an existing speculative CMP.

Overall, at any time, there is an ordering of the threads and also an ordering of the chips. At the chip level, chips relate to other chips through predecessor and successor ordering. The chip hosting the non-speculative thread, which we call the non-speculative chip, is the earliest in the sequence. Within each chip, the least speculative thread is regarded, for all internal operations, as being non-speculative. This means that such a thread writes through its L1 cache and does not create LMDT entries. Such a thread can behave as non-speculative within its CMP but be speculative across the system.

The per-node L2 caches across the system play the same role that per-processor L1 caches play in the CMP. In the speculative nodes, they keep speculative data from being written back to memory until it is safe to do so. The only L2 cache that does not need to hold any speculative state and, therefore, could be write-through, is the one at the non-speculative node. For performance reasons, however, we keep all these large L2 caches in write-back mode.

3.2 Protocol Operation

To understand the workings of our scheme, we examine several aspects of the protocol, including handling loads and stores, cache line replacements, thread mapping, L2 and GMDT support, and commits and squashes. We mention the modifications required to the CMP as we describe the operations. As indicated before, we have preferred simplicity over higher performance in many aspects of the design.

3.2.1 Loads and Stores

Loads and stores proceed much as in the CMP (Section 2.2), except that we now have another level in the memory hierarchy. Upon a load by a processor, the CMP first tries to resolve the access locally, by seeking versions in its L1 and in predecessors through the LMDT. If it fails, the system repeats the search in L2 and in predecessor nodes via the GMDT module at the home of the memory line. Like in the CMP protocol, the most recent version is located at the closest predecessor whose Store bit is set (possibly the requestor itself). If the version is to be fetched from a predecessor, since all threads in that node are predecessors of the requestor, all we need is for its LMDT to locate the latest version in the chip and to return it. For this scheme to work, of course, the LMDT of the speculative CMP must be enhanced to allow external requests of versions.

A store by a thread proceeds like in the CMP protocol: copies in the successor threads up until, but not including, the first redefinition are invalidated, while predecessors are sent delayed-invalidation messages. These two actions involve, in the general case, threads inside and also outside the chip (Figure 3d). While the LMDT takes care of the threads within the chip, the GMDT is responsible for sending point-to-point messages to the appropriate predecessor and successor chips. Again the LMDT must be augmented to support such external messages.

We use the existing directory and GMDT states to reduce the number of delayed-invalidations and invalidations that we send. How the directory helps is discussed in Section 3.3. As for the GMDT, note that, if another chip already has its Store bit set in the GMDT, it means that it has already sent delayed-invalidations and invalidations to its predecessors and successors respectively. Therefore, upon a store operation, we need to send invalidations only if no predecessor Store bit is set, and then only up to the first successor whose Store bit is set, non-inclusive[2]. Symmetrically, we need to send delayed-invalidations only if no successor Store bit is set, and then only down to the first predecessor whose Store bit set.

L2 caches contain a Stale and a Forwarded bit per line. These bits are used in a similar way as the Stale and Forwarded bits in L1 (Section 2.2). However, unlike L1 caches, L2 caches do not contain Safe Load or Safe Store bits. Recall that these bits are used to save a trip to the LMDT on an L1 hit: if a load finds the Safe Load or a store finds the Safe Store bit set, the LMDT is guaranteed to be up-to-date and, in addition, the store is guaranteed not to generate squash, invalidation, or delayed-invalidation messages. We do not include these bits in L2 because there are few scenarios in which they would actually save trips to the GMDT on an L2 hit. More specifically, the sole presence of the Load bits in the LMDT makes the Safe Load bit in L2 redundant. As for sparing the Safe Store bits in L2, it can be easily shown that most of the unnecessary GMDT accesses in stores are filtered out by the Safe Store bits in the L1 caches.

One change that we make to the speculative CMP is related to exposed loads, namely loads that hit in L1 but find that the Safe Load bit is zero. These loads require performing a check for dependences in the LMDT and, potentially, in the GMDT as well. Since such a check may be time-consuming in scalable machines, our change consists of returning the data to the processor immediately. In the background, a protocol message checks the MDTs. If the check concludes that the load has used a stale version, a squash is generated. We expect, however, that the data in the cache will be usually up-to-date. Such is the case when data that is not modified by any processor remains in the caches across thread executions. Note that this optimization introduces the possibility of loads generating squashes. However, this technique, which we call *aggressive exposed loads*, is attractive due to its latency-hiding ability.

Finally, because all L2 caches are write-back, even actions from the non-speculative thread in the system must generate and update GMDT entries like any other thread in any chip. As a result, we could run the risk of stalling the non-speculative thread due to lack of GMDT space. To avoid this problem, we organize the GMDT as a cache, backing up its data in main memory in a manner very similar to a directory cache.

3.2.2 Cache Line Replacements

Similarly to the CMP case, our scheme maintains versions in the caches and imposes restrictions as to which levels of the memory hierarchy these versions can propagate to. Following the speculative CMP protocol of Section 2.2, only the least speculative thread in a chip is allowed to modify the L2 state. Any other thread in the chip stalls if it tries to displace an updated line from its L1. Likewise, only the non-speculative thread in the system is allowed to displace updated lines from the local L2 into memory. Any other thread in any chip stalls if it tries to displace an updated line from its local L2. Stalled threads resume execution when they reach non-speculative status.

Lines that have been speculatively loaded but not modified can always be displaced from any cache. The reason is that no information is lost: MDTs record what words are being speculatively loaded by what threads or chips.

3.2.3 Mapping of Threads

The requirement to minimize the modifications to a speculative CMP like the one of Section 2.2 introduces some constraints to thread mapping. As indicated in Section 3.1, to maintain the hierarchical abstraction, it is simplest if we assign threads to chips in chunks of consecutive threads. Each chunk contains as many threads as there are processors in the CMP. Furthermore, since the CMP protocol tightly couples the on-chip threads, allowing different processors within the chip to execute threads from different chunks at the same time would require significant CMP changes. As a result, we only assign a new chunk to a CMP when all its processors have completed the execution of (although not necessarily committed) the previous chunk.

This restriction of waiting until all the threads in a chunk have finished before starting a new chunk may lead to some idle time if the different threads in a chunk have load imbalance. Recall from Section 2.2 that the speculative CMP design that we adopted requires that a processor stall when the thread that it is running finishes and is still speculative. The resulting idle time we called intrinsic chip imbalance. Now, in addition to this, assigning threads in whole chunks implies that each processor must also wait for its successor processors in the chip to finish their threads before starting to work on a new chunk. We call this second source of idle time *induced chip imbalance*. The combination of intrinsic and induced chip imbalance we call *chip imbalance*.

The equivalent to intrinsic chip imbalance in a scalable system would be not to start a new chunk on a chip until all the predecessor chunks had finished and committed. This approach, which effectively implies assigning thread chunks to chips statically, would lead to poor performance in the presence of load imbalance. Further, its impact would increase with the number of chips in the system. Consequently, we do not impose this restriction.

Instead, when a chip finishes a chunk of threads, it asks for another chunk and starts executing it, irrespective of whether the predecessor chunks have finished. With this approach, chunks of threads are dynamically assigned to chips, therefore reducing idle time due to load imbalance. The set of contiguous uncommitted

[2] Recall that if the Load bit of that successor is also set, we trigger a squash operation and, as part of it, invalidate its copy of the word.

thread chunks that have been assigned to chips to execute, we call the *active window*. The active window always starts with the non-speculative chunk. The active window size changes dynamically and, at any given time, is likely to be larger than the number of chips in the machine. Its maximum size is set to a hardware-predefined limit. In general, the only situation in which load imbalance would become apparent at the system level would be when the active window has reached its maximum size and we are unable to assign new chunks. This effect we call *system imbalance*.

Overall, our dynamic chunk scheduling approach is supported with practically no modification to the speculative CMP hardware. Most of the modifications are performed on the node's L2 and local GMDT module. Of course, not modifying the CMP hardware means that the system may potentially suffer from chip imbalance. Fortunately, however, the magnitude of chip imbalance depends on the number of processors per chip, and not on the total number of nodes in the machine. Therefore, its impact stays largely constant instead of increasing as we scale up the machine to more nodes.

3.2.4 L2 Support

The speculative protocol within the CMP keeps working as described in Section 2.2: threads in a chunk commit within the chip in order and, when they do, they write back all the updated lines in their L1 to L2. These updates must remain in L2 until the chunk commits at the system level.

Before the chunk commits, however, the chip may be given another chunk to execute. As chunks execute and complete, L2 will accumulate speculatively updated lines from several chunks. The state of each chunk must be carefully buffered and kept separated from that of the others. When one chunk finally commits at the system level, only its updated lines must be written back from L2 to main memory.

The problem of buffering data from several threads in the context of speculative parallelization has been addressed in two ways in the literature: using separate write buffers to hold versions from different threads [7] or extending the cache tags to identify the thread that owns the version in each cache line [17]. Our scheme uses an approach similar to the second one, adding chunk IDs to the L2 cache tags. We will see later that chunk IDs are encoded with numbers going from 0 to $w - 1$, therefore using $\log_2 w$ bits, where w is the hardware-defined maximum active window size. These bits are on a per-line as opposed to a per-word basis to reduce the overhead. Consequently, when two different chunks update the same line, they create two line versions even if they update different words.

Since the L2 cache may potentially hold much state from different chunks, it may suffer conflicts. As indicated in Section 3.2.2, the processor is stalled when a cache conflict is about to displace a line modified by a speculative chunk. To reduce the chances of stalls, we extend L2 by adding a set-associative buffer that acts somewhat like a victim cache [8]. This buffer stores speculatively-modified lines as they are displaced from L2. However, the buffer does not need to store speculatively-loaded yet clean lines that are evicted from L2. The reason is that the GMDT already has a record of them.

3.2.5 GMDT Support

Since there can be more thread chunks currently assigned to nodes than nodes in the machine, the GMDT really keeps the Load and Store bits on a per-assigned-chunk basis as opposed to a per-node basis. In addition, to locate the nodes to which the chunks have been assigned, the GMDT keeps a data structure with the mapping from the chunks in the active window to the nodes. Both the number of Load and Store bits per word in the table and the number of entries in the chunk-to-node mapping structure are equal to w, which is a limit set in hardware for the number of chunks that can be assigned at a time.

Figure 5 shows the chunk-to-node mapping structure. Each entry in the structure keeps the ID of the node to which a chunk has been assigned. The structure is a circular queue. Two pointers point to the non-speculative chunk and to the most-speculative one, marking the boundaries of the current active window. The first pointer moves as the GMDT is informed that a chunk has committed, while the second one moves as the GMDT assigns new chunks for execution

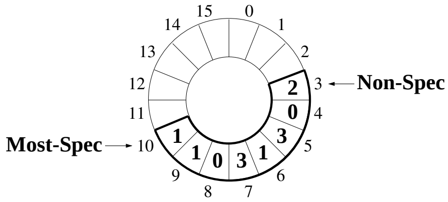

Figure 5: Chunk-to-node mapping structure in the GMDT. The figure corresponds to a 4-node machine where the non-speculative chunk (chunk ID 3) has been assigned to node 2, the next one (chunk ID 4) to node 0, and so forth. Node 1 is currently assigned 3 chunks.

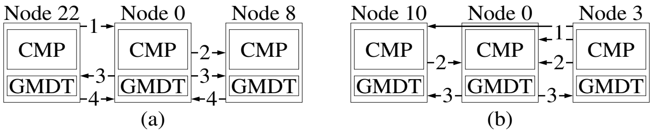

Figure 6: Messages involved in a commit (a) and a squash (b) operation. In the commit, a thread chunk in node 22 passes non-speculative status to one in node 8. In the squash, the triggering violation occurs in node 3.

or as chunks get squashed. All transactions between the GMDT and the nodes are logged with the corresponding chunk ID, which uniquely identifies the chunk at that moment. In the figure, the chunk ID can go from 0 to 15. The application must be able to figure out what work to do once it is given a chunk ID. This information can be computed locally by each node or through a shared data structure, depending on the nature of the application.

Since the GMDT is distributed, this chunk-to-node mapping structure is replicated in all the GMDT modules. All modifications take place in a chunk commit, squash, or assignment. They are initially recorded in one GMDT module, namely the one in node 0 (GMDT0). The updates are then propagated to the other GMDT modules in the background. We now consider these situations.

3.2.6 Commits

Our protocol attempts to speed up the critical path in commit and squash operations. In a commit, the committing thread passes the non-speculative status to its successor thread as fast as possible; in a squash, the threads to be squashed are detected, squashed, and restarted in parallel.

We consider the commit of a chunk first. Within a chunk, thread commit is always in order. A thread commits exactly like in the single CMP protocol of Section 2.2. The effects of the operation do not go past the L2 cache in the node, where all the updates of the on-chip threads committed so far are accumulated.

When the last thread of the chunk has committed, the chip signals a chunk completion event locally. If the chunk is non-speculative, the node proceeds with a chunk commit. Following the simplified protocol of Section 2.2, which does not allow dirty data to remain in a cache across speculative thread initiation, chunk commit involves writing back the dirty L2 lines to memory. It also involves updating the Stale, Forwarded, Invalid, and Dirty bits in the L2 tags. We expect these L2 write back and tag manipulation operations to be performed efficiently in hardware by the L2 cache controller. The CMP is oblivious to the whole chunk commit operation.

When all these operations complete or (if the chunk is speculative) as soon as the chunk finishes, the node sends a message to GMDT0 asking for a new chunk to execute. In addition, if the completed chunk committed, the message also informs of the commit. In the latter case, GMDT0 immediately sends a message to the node with the next chunk ID in the sequence, passing the non-speculative status. We can see, therefore, that passing the non-speculative status from a chunk in one node to a second chunk in another node can be implemented very efficiently with two hardware messages. This is shown with messages 1 and 2 in Figure 6a.

After this, GMDT0 assigns the next available chunk to execute to the node that just finished the chunk, and sends all these changes to all nodes. This is done through messages 3 in Figure 6a. These messages occur in the background, are not in any critical path, and do not use up any processor cycles. In each node, on reception of message 3, the GMDT module records the assignment of the new chunk and, if applicable, the commit. In addition, the node that has just finished the chunk can now start running the new chunk. The messages labeled 4 are acknowledgments of 3.

Consider now the node that receives message 2 from GMDT0 indicating that one of its chunks is now non-speculative. Two cases are possible, neither of which requires any involvement of the local CMP. If, at message arrival, the node is still executing the chunk, no action is taken beyond recording the new status of the chunk. Later, when the chunk completes, the node will proceed with a chunk commit as explained before.

A second case occurs when, at message arrival, the chip has already finished executing that chunk. In fact, the chip may have even executed and completed several successor chunks since then. In this case, the node immediately starts a chunk commit. As usual, the local CMP is not involved. The operation includes writing back dirty L2 data and manipulating the L2 tag bits. However, this operation applies only to the L2 lines that are tagged as belonging to the non-speculative chunk. The other L2 lines are left untouched. At the end, the node informs GMDT0 of the completion of the commit operation but does not ask for a new chunk.

The fact that GMDT modules are updated with some delay may cause minor races that our protocol handles gracefully. For example, a newly-assigned chunk may send a request to a GMDT module before the latter knows of its existence. In this case, the request is bounced back to the sender for retry. Similarly, a new non-speculative chunk may send a request to a GMDT module before the latter knows of its non-speculative status. In this case, loads and stores are processed as usual. Write backs, however, are bounced because speculative chunks are not allowed to write back.

3.2.7 Squashes

The violation that triggers a squash may be detected at a GMDT module or at a LMDT. If the latter, the local GMDT module is informed. In any case, this GMDT module, which we call the *initiating* one, decides which chunks must be squashed: the one that read prematurely and its successors. In general, the GMDT does not have enough information to distinguish between the different threads in a chunk. Hence, squashing is done at the chunk level. It involves clearing the corresponding Load and Store GMDT bits, invalidating the dirty and forwarded L2 lines and, if the chunk is still running, clearing the whole LMDT and invalidating the dirty and forwarded L1 lines.

The exception is when the violation is detected at the LMDT of the chip running the non-speculative chunk. In such a case, in this chunk, only the violating thread and its successors are squashed. For the squashed threads, we clear their LMDT bits and invalidate their dirty and forwarded L1 lines. The L2 and GMDT bits are left unmodified[3].

The actual squash proceeds as follows. When the violation is detected, the initiating GMDT module sends a message to all the other nodes (message 1 in Figure 6b) specifying which chunks to squash. Those nodes with chunks to squash do so. Note that, for a chunk to be squashed, all its pending memory accesses have to necessarily complete or be aborted. In addition, all nodes update their GMDT modules by clearing the state corresponding to the squashed chunks.

Finally, all the GMDT modules synchronize in a barrier: they send a message to GMDT0 and receive an acknowledgment when all have done so (messages 2 and 3, respectively, in Figure 6b). The acknowledgment from GMDT0 specifies the new assignment of chunks to nodes or, possibly, that the assignment of chunks is as before. With this information, the GMDT modules are updated and the chunks restarted in parallel. Note that the barrier is needed to ensure that requests before and after the squash are clearly separated. Indeed, all requests arriving at a GMDT module from squashed chunks between messages 1 and 3 in Figure 6b should be bounced.

Overall, squashing does not affect the processors running non-squashed threads. Furthermore, our protocol uses the GMDT to identify, squash, and restart potentially many threads in parallel. The operation requires a synchronization of the GMDT modules only, not of the processors.

3.3 Interaction with the Directory

The GMDT is distributed across the nodes like the directory and is connected to it. For the ordinary data, the GMDT is unused. For the data marked in the code as speculatively accessed, in principle, the GMDT replaces the directory functionality. In practice, the GMDT does all the work but it can also use the directory for optimization purposes. In this case, the directory needs only small modifications because the baseline coherence protocol largely runs unaffected by the speculative protocol.

It is easy to see why the GMDT replaces the directory for data marked as speculatively accessed: on a read transaction, the GMDT identifies the correct version of the variable to supply; on a write transaction, the GMDT identifies the versions to squash. These two operations cannot be performed by a conventional directory because the latter does not support the notions of multiple versions or thread order.

However, under certain conditions, the GMDT may not be so efficient when it has to send invalidation and delayed-invalidation messages to remove outdated versions from caches after a write. The reason is that the GMDT only keeps correct sharer information for the data accessed by the currently-active chunks. For that data, we can use the GMDT state to reduce the messages, as discussed in Section 3.2.1. However, no record of sharers is kept for the other data because, when a chunk commits, its Load and Store bits in the GMDT are reset. The result is that, for that data, the GMDT may have to conservative send unnecessary invalidation and delayed-invalidation messages.

To send fewer such messages, the GMDT can use sharer information kept by the directory. This requires that the GMDT keep the directory largely up-to-date on processor reads and writes to data marked as speculatively accessed.

Specifically, when one such read or write reaches the GMDT, the directory marks the requesting node as a sharer of the line. Irrespective of the type of access, the dirty bit is not set. The reason is that there may be several different modified versions of the same variable in different caches, and the directory has only one dirty bit. Still, pretending that the nodes are read-only sharers is safe because the directory is never queried to distinguish between versions. Furthermore, as a chunk commits, it writes back its updated version of the variable to memory, keeping the cache in clean state. At that time also, if the version in the cache was marked with the Stale bit, it is automatically invalidated. Overall, cache entries eventually become consistent with memory or invalid.

In reality, for a given memory line, the directory must conservatively keep a superset of all the nodes that currently cache it. This is because invalidations cannot remove directory entries: the directory keeps information only at a memory-line grain size, whereas invalidations are sent to single words. Furthermore, delayed-invalidations cannot remove directory entries either: they take effect with some delay. Despite these considerations, however, the directory information is still useful to reduce the number of invalidations and delayed-invalidations sent on a write.

Finally, there is another reason why the directory should keep at least a superset of all the nodes that currently cache the line: speculative data may be accessed in conventional coherent mode after the speculative section completes. As a result, remembering the sharers smoothens the transition out of the speculative section.

Overall, ordinary shared data and potentially speculative data can remain in L1 and L2 as we enter and exit a speculative section of code. Before we start a speculative section, our simple protocol re-

[3] Although the GMDT may temporarily keep some outdated Store bits set for the non-speculative chunk, correctness is still guaranteed: if a read from a successor is directed to the chip running the non-speculative chunk due to one of these outdated Store bits, the request will be bounced back to memory as if the requested line had just been displaced from the caches. At that point or when the non-speculative chunk commits, the outdated Store bit is cleared.

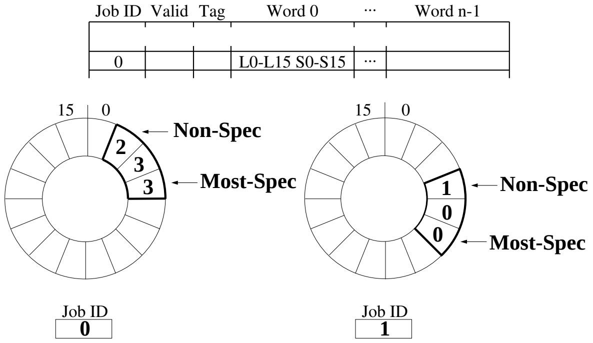

Figure 7: Support required in a GMDT module to run multiprogrammed loads. In the example, two jobs can locally allocate potentially speculative data. The machine has four nodes and a maximum active window size of 16. Job 0 runs on nodes 2 and 3, while job 1 runs on nodes 0 and 1.

quires that we write back the dirty data from all the caches to make them consistent with memory. Before leaving the speculative section, memory and caches are consistent again: cache write-backs at commit points have updated memory, and invalidations and self-invalidations have purged outdated versions from caches. The directory may be left with a superset of the sharers for each line with speculative data. The final operation that we perform before leaving the section is to invalidate from the caches any (necessarily clean) line that has some valid and some invalid words, so that the conventional, per-line coherence protocol can take over correctly. This can be done with a hardware signal that invalidates all such lines in parallel.

3.4 Support for Multiprogramming

With simple extensions, our scheme supports multiprogrammed loads where jobs may or may not use speculative parallelization. Recall that, in our design, we want to use the unmodified speculative CMP of Section 2.2. This necessarily implies that, once a thread chunk that uses speculative parallelization starts running on a CMP, it has to run to completion before releasing it. Completion implies that the speculative state is moved to L2, not that the chunk commits. If the chunk is preempted before completing, it has to be squashed, since the speculative state in the L1 caches will be lost.

All hardware changes needed are limited to the GMDT and, optionally, the L2 caches. In each GMDT module, we now keep a chunk-to-node mapping structure for each job that has locally allocated pages of potentially speculative data (Figure 7). In addition, we allow all these jobs to insert entries in the GMDT table that keeps the Load and Store bits. To identify the entries belonging to each job, the table is extended with an extra field. This field, called *Job ID*, keeps the ID of the job that owns the entry. Note that there is no overlap between the address tags of entries belonging to different jobs. Furthermore, since this table is backed up in memory (Section 3.2.1), letting several jobs use the same table can at most cause more overflows into memory. Finally, associated with each chunk-to-node mapping structure, we have a software structure with the list of nodes that are currently running chunks belonging to this job (*Node List*).

Two approaches to multiprogramming the machine are possible. In the most aggressive one, a free node can be assigned a chunk from any job. In a second, more conservative approach, a free node, if it still has uncommitted chunks from a job in its L2 cache, can only be assigned a chunk from that same job. Since the GMDT knows the current assignment of chunks and their status for all the jobs, it can enforce the second policy if required. Note that, in the conservative scheme, the Node List for a given job may change with time but the Node Lists for two jobs may not overlap at any time. For the aggressive scheme, the Node List for a job may change with time and the Node Lists for two jobs may overlap at any time.

The two schemes differ in the support required and the performance expected. The aggressive scheme needs more support because a L2 cache may end up keeping speculative state belonging to different jobs. As a result, to identify which lines belong to which job, we need to extend the chunk ID field in the L2 cache tags. The ID must include, in its most-significant bit positions, the job ID. Such longer tags are not needed in the conservative scheme.

The performance is likely to be higher in the aggressive scheme because of its higher flexibility: an idle node will always get a chunk if one is available in any job. This is not true in the conservative scheme. However, recall that when a chunk running on a node is about to displace from L2 an uncommitted updated line from another chunk, the processor stalls. Consequently, a job running under the aggressive scheme may stall due to another job. In the conservative scheme, instead, a job in a node can only be stalled due to itself. In neither case, however, deadlock is possible because at least one non-speculative chunk is making progress, and chunk commit does not involve the local CMP.

3.5 Overall GMDT Benefits

The previous discussions have uncovered the pros and cons of the GMDT. The main attraction of the GMDT scheme is that it efficiently implements a protocol that supports multiple versions of the data. Such a protocol uses few messages and is fast. For example, a load finds the correct version in the machine with a fast, one-lookup transaction of a single message. In addition, squash signals are sent in parallel to all the threads that need it and, after the squash, the threads are restarted in parallel. Finally, by exploiting the state in the directory (Section 3.3) and GMDT (Section 3.2.1), the protocol ends up sending only few unnecessary invalidations and delayed-invalidations.

A second attraction of the scheme is that, by keeping a Load bit in the GMDT for each speculatively-loaded datum, it allows processors to displace speculatively-loaded, unmodified data from their caches without causing stalls.

The main disadvantage of the GMDT scheme is that we need to keep its distributed state consistent. Fortunately, although keeping the state consistent involves extra messages at chunk commit, squash, and assignment points, this activity occurs in the background without using processor cycles.

3.6 Compiler and ISA Support

We expect that the code is annotated, typically by the compiler, to mark the sections where speculative parallelization should be enabled. Within these sections, it is more efficient if we mark the data that will be accessed speculatively. Note that it is perfectly possible to access all data speculatively. However, it is probably inefficient due to the extra overheads involved in dealing with speculative data and may also hurt performance if there are cache or LMDT overflows. In any case, our system allows data to change roles across sections. The ability for data to be accessed speculatively in one section and non-speculatively in another allows better use of the resources.

A good way to differentiate speculative from plain accesses is to extend the ISA to have speculative memory instructions. Of course, a given memory line cannot have both kinds of data. An alternative approach is to mark virtual memory pages to be dealt with as speculative or as plain shared data. However, this approach needs careful data placement to make sure that speculative and plain data do not share the same page.

4 EXPERIMENTAL SETUP

4.1 Applications

To evaluate our scheme, we choose two Perfect Club applications, namely *Track* and *BDNA*, one SPECfp95 application, *APSI*, and two HPF-2 applications, namely *Euler* and *DSMC3D*. The input sets used for these applications are the standard ones provided with the suites. The exception is *APSI* which uses a 512x1x64 grid size. These applications are representative of sequential scientific workloads. We choose them because they all spend a large fraction of

Application	*Loop to Parallelize*	*% of Seq. Time*	*Avg. Iterations per Invocation*	*Speculative Data (KB)*
Track	nfilt_300	41	502	240
APSI	run_20	21	64	40
DSMC3D	move3_200	33	758972	24767
Euler	dflux_100	90	2494	686
BDNA	actfor_240	32	1499	7

Table 3: Characteristics of the applications studied. The fraction of time spent in the loop to parallelize speculatively is given as a percentage of the total sequential time of the application on a SGI server.

Application	*Parameter (Average)*	*RAW*		*WAR*		*WAW*	
		Same Word	*False*	*Same Word*	*False*	*Same Word*	*False*
Track	*Number*	0.1	4869	0.1	47	0	4880
	Distance	1.0	1.6	1.0	3.1	0	1.6
APSI	*Number*	0	0	95232	333312	95232	333312
	Distance	0	0	1.0	1.0	1.0	1.0
DSMC3D	*Number*	147390	9350766	102912	509315	85343	8939798
	Distance	2640.3	224.9	260050.8	228047.3	2608.2	89.2
Euler	*Number*	0	104066	0	0	0	104066
	Distance	0	415.0	0	0	0	415.0
BDNA	*Number*	0	0	32422	48518	998500	1492510
	Distance	0	0	1.0	1.0	1.0	1.0

Table 4: Static cross-iteration data dependences exhibited by the loops that we parallelize speculatively.

their sequential execution time on loops that cannot be parallelized by state-of-the-art compilers. These loops have dependence structures that are either too complicated to be analyzed at compile time, or unknown because they depend on input data. For example, many of them have doubly-subscripted accesses to arrays. As a result, the Polaris compiler [2] is unable to parallelize them. In our evaluation, we perform speculative parallelization on these loops and analyze how they are sped up.

Table 3 shows, for each application, the loop that we attempt to parallelize speculatively, the fraction of the sequential execution time taken by this loop on a SGI server, the average number of iterations executed in the loop per loop invocation, and the size of the data accessed through speculative references (*speculative data*). From the table, we see that these loops account for 21-90% of the total execution time. In a parallel system, their weight will likely be even higher if a parallelizing compiler is used to reduce the execution time of the other parts of the code. Note that speculative threads must buffer in their caches not just the speculatively modified data but all the data that they modify.

Table 4 shows the static cross-iteration dependence structure of the loops that we parallelize speculatively. RAW, WAR, and WAW dependences are classified based on whether the two dependent references access the same word (*Same Word*) or different words (*False*) of a memory line. The memory line size used is 16 words. For each type of dependence, the table shows the average number of dependences found for each invocation of the loop, and the average dependence distance in number of iterations.

From the table, we see that same-word RAW dependences are relatively scarce, except for *DSMC3D*. This is unlike same-word WAR and WAW, of which there is plenty. As for false dependences, all three kinds manifest themselves more abundantly. We observe that, many times, all these dependences have very short average distances, which means that they can easily occur out of order. Even large average distances like in *DSMC3D* hide short instances. Recall that our protocol handles all types of in-order and out-of-order dependences without triggering a thread squash except for out-of-order, same-word RAW dependences.

In our experiments, we proceed as follows. The Polaris compiler identifies and instruments the loops in Table 3. The instrumentation

Processor Param.	*Value*
Issue width	4
Instruction window size	64
No. functional units (Int,FP,Ld/St)	3,2,2
No. renaming registers (Int,FP)	32,32
No. pending memory ops. (Ld,St)	8,16

Memory Param.	*Value*
L1,L2,VC size	32KB,1MB, 64KB
L1,L2,VC assoc.	2-way,4-way, 8-way
L1,L2,VC line size	64B,64B,64B
L1,L2,VC latency	1,12,12 cycles
L1,L2,VC banks	2,3,2
Local memory latency	75 cycles
2-hop memory latency	290 cycles
3-hop memory latency	360 cycles
LMDT,GMDT size	512,2K entries
LMDT,GMDT assoc.	8-way,8-way
LMDT,GMDT lookup	4,20 cycles
L1-to-LMDT latency	3 cycles
LMDT-to-L2 latency	8 cycles
Max. active window	8 chunks

Table 5: Parameters of the processor and memory system models.

consists of classifying each variable into one of three classes: variable allocated in a writable memory line that is possibly accessed in two or more iterations; variable allocated in a memory line that is at most accessed in one iteration or is read only; and private or privatizable variable. We mark the first class as speculative, so that they trigger our protocol in our simulations. This class includes the data with potential cross-iteration dependences. In addition, it also includes situations where the compiler is sure that no two iterations access the same word of a line. In this case, however, since different words of the line may end up being buffered in different L1 caches by speculative threads, we cannot use the ordinary, line-based cache coherence protocol. Variables are privatized whenever the compiler finds it safe and convenient, so that the loop runs more efficiently in parallel. Once the loop is instrumented, we perform detailed execution-driven simulations. In our base experiments, each thread is composed of a single loop iteration.

4.2 Simulation Environment

Our execution-driven simulation environment is based on an extension to MINT [21] that includes a superscalar processor model with non-blocking memory operations [10], and supports dynamic spawn, squash, restart, and retire of light-weight threads. We use these threads to attempt speculative parallelization on the loops in Table 3.

The processor model is that of a 4-issue dynamic superscalar with register renaming, branch prediction, and non-blocking memory operations. The left section of Table 5 shows some of the parameters used in the processor model. Processors are grouped into 4-processor chips.

The memory system models the organization in Figure 4: a CC-NUMA multiprocessor whose nodes include speculative CMPs. The architecture of the speculative CMP is as described in Section 2.2. Each CMP includes 4 processors with their private L1 caches and a LMDT. Each node in the machine has a CMP, an L2 cache shared by all the local processors, a portion of the global memory and directory, a network controller, and a GMDT module. The machine is equipped with a directory-based cache coherence protocol in the lines of DASH [12]. In addition, our memory system implements the speculation protocol outlined in Section 3.2. The system simulated has 4 CMPs, for a total of 16 processors.

The right section of Table 5 lists the main parameters of the memory system model. In the table, L1, L2, VC, and memory latencies are round-trip times from the processor. VC stands for Victim Cache. All latencies are in processor cycles and do not include contention effects. We accurately model contention everywhere except in the scalable interconnect, where a fixed time is assumed for each hop.

In our evaluation, we apply speculative parallelization only to the loops of Table 3. To assess the impact of our scheme on the whole application, our resulting speedup numbers should be weighted using the fraction of the execution time taken by the loops as shown in

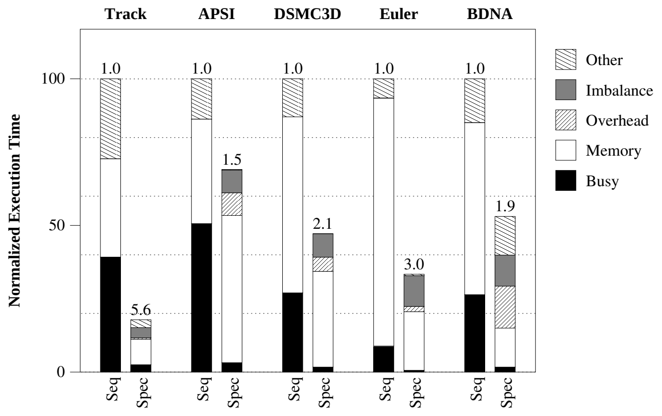

Figure 8: Execution time breakdown and speedups. The numbers on top of the bars are the speedups of the loops over the sequential execution.

Table 3.

Since we want to measure loop speedups in a scenario in which other parts of the code have been parallelized by the compiler, the data must be necessarily distributed across the nodes even in the experiments that run the loop sequentially. In order to keep the same data distribution for all experiments, we use a static round-robin page allocation policy across the nodes.

5 EVALUATION

5.1 Overall Performance

We start by quantifying the speedups delivered by our speculative parallelization scheme. Figure 8 compares the execution times of the loops under sequential execution (*Seq*) and under our speculative parallelization for 16 processors (*Spec*). On top of each bar we quantify the speedups of the loops over the sequential execution.

For each application, the bars are normalized to sequential execution and broken down into the following components of the execution time: execution of instructions (*Busy*); stall due to memory accesses (*Memory*); overhead associated with speculative parallelization, including thread commit time, thread squash time, and stalls caused by L1, L2, VC, and LMDT overflows (*Overhead*); idle time waiting for other threads to complete (*Imbalance*); and conventional pipeline hazards (*Other*). The contribution of each category is measured at the grain size of issue slots [10].

From the figure, we see that our scheme delivers speedups that range from 1.5 to 5.6. Over the five applications, the average speedup is 2.8. Interestingly, the results show that, in all applications but one (*BDNA*), the main obstacle to better speedups is not related to the speculative aspect of the parallelization, but simply to the stall time due to memory accesses (*Memory*).

The overheads associated with speculative parallelization (*Overhead*) account for a quarter of the time in *BDNA* and, less importantly, for about 10% in *APSI* and *DSMC3D*. In *BDNA*, practically all this overhead is caused by stall due to overflows in L1. The dirty working set of this application is too large for the 32KB L1. Recall that our protocol does not allow displacements of dirty data and any attempt to do so causes a processor stall. L1 overflow also causes most of the overhead in *APSI*. Our experimental results reveal that no application suffers from LMDT overflows. Also, the victim cache in L2 successfully absorbs most L2 overflows.

In *DSMC3D*, practically all overhead is caused by thread squashes. This includes the time required to drain all the pending transactions in squashed threads, synchronize all the GMDT modules, and restart the threads. As shown in Table 4, *DSMC3D* has same-word RAW dependences, the only ones that cause squashes in our protocol if they happen out of order. Despite the long average distance that these dependences have for *DSMC3D* (Table 4), some of them have short distances and actually occur out of order, resulting in squashes. Fortunately, no other application wastes much time due to squashes.

Commit time, which is also part of *Overhead*, has a very minor role across all the applications. The reason is two-fold. On the one hand, these loops have relatively large iterations, which result in threads spending much more time executing than committing. On the other hand, only the on-chip commit of threads has a significant effect on processor time; chunk commit is done in the background, except in the case when the CMP has not already started working on a new speculative chunk (Section 3.2.6).

Overall, among the different components of *Overhead*, the only significant ones are L1 overflow and, if the application has short-distance, same-word RAW dependences, thread squashes. The simplest improvement toward diminishing overhead would be to provide storage mechanisms to capture L1 overflows. However, this would require changing the architecture of the speculative CMP, which is not an option in our approach.

Finally, we consider load imbalance. *Imbalance* accounts for a significant 10-30% of the execution time in the parallelized loops. Practically all of this time is due to what we called chip imbalance in Section 3.2.3, which comes from the way threads are scheduled within CMPs. As pointed out before, the only way to reduce the impact of chip imbalance is to significantly change the architecture of the speculative CMP. Nevertheless, as indicated in Section 3.2.3, we expect the impact of chip imbalance to remain roughly constant as we scale up the machine to large numbers of nodes.

A second source of imbalance is related to idle time while the node, although finished, cannot start working on a new speculative chunk. This system imbalance can only be caused by the overflow of the active window, which happens very rarely in our case.

5.2 Effect of Loop Unrolling

In the previous section, speculative parallelization was performed using single-iteration threads. Unrolling the loops and giving one of these bigger iterations to each thread may improve the performance of our architecture. Specifically, loop unrolling may enable a higher reuse of the cached data if consecutive iterations access similar sets of memory lines. Loop unrolling may also reduce cross-thread data dependences and even reduce chip imbalance. Finally, observe that loop bodies smaller than ours could also benefit from exposing more ILP.

On the other hand, loop unrolling may also degrade performance. For example, having fewer threads reduces the amount of parallelism that can be exploited. Also, the resulting bigger iterations may be more imbalanced and cause a higher chip imbalance. Squashes, too, may have a more severe impact on performance, as the average work to be redone is greater. Finally, loop unrolling increases the amount of dirty state that must be buffered in the caches, which increases the probability of overflow-induced stalls.

In this section we evaluate the impact of loop unrolling in our loops. Figure 9 compares the result from the previous section (*Spec* is now relabeled *Blk1*) to configurations where threads are composed of blocks of 2 or 4 base iterations (*Blk2* and *Blk4*, respectively). For each application, the bars are normalized to *Blk1* and broken down as in Figure 8. As before, the numbers on top of the bars are the speedups of the loops over the sequential execution. Note that we do not attempt *Blk4* for *APSI* because, as shown in Table 3, its loop has only 64 iterations.

The results show that loop unrolling improves the performance of our architecture. The speedups improve significantly: if we use 2 and 4 base iterations per thread, speculative parallelization delivers average speedups over the sequential execution of 3.4 and 4.2, respectively. Furthermore, some applications perform very well. In particular, *Track* reaches a speedup of 8.7.

The reasons for the improvement are a reduction in the memory time and, to a lesser extent, a reduction in chip imbalance. The memory time decreases to various degrees in all applications, sometimes significantly as in, for example, *APSI*. Indeed, cached data is being reused more effectively. Chip imbalance also decreases in most applications thanks to the load averaging effect of performing loop unrolling. The exception is *APSI*, where *Imbalance* increases. The

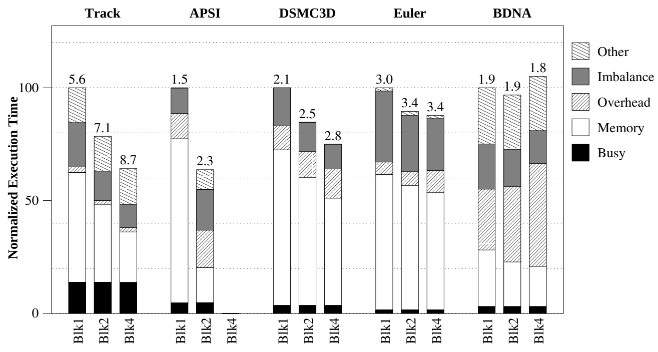

Figure 9: Effect of using multiple loop iterations per thread. The numbers on top of the bars are the speedups of the loops over the sequential execution.

reason is that the loop has so few iterations after blocking that, at the end of the loop, some nodes have to wait a long time for the others to finish. For example, in *Blk2*, each node only executes 2 chunks.

The only category that increases with loop unrolling is *Overhead*. This effect, which takes place in practically all applications, is largely due to an increase in the stall time due to L1 overflows. Loop unrolling has increased too much the amount of dirty state that must be buffered in the caches. This effect is specially noticeable in *BDNA*.

Overall, the data shows that loop unrolling can be a very useful technique. However, care must be taken to estimate the increase of the dirty working set and its corresponding pressure on the caches.

5.3 Granularity of Speculative State

Our protocol keeps speculative state on a per-word basis in the MDTs and, to some extent, in the caches. It is interesting, however, to consider what happens if, in the whole machine, we only kept state at the grain size of a memory line. The protocol is otherwise mostly unchanged and, in particular, still has support for multiple versions of a given memory line.

In general, if the state is kept per line, we need to perform squashes on every out-of-order RAW and WAW dependence, both false and due to same-word accesses. This is in contrast to our scheme which, by keeping the state per word, only suffers squashes in case of same-word out-of-order RAW dependences. The per-line protocol needs to set both the Load and the Store bits upon creating a version, so that the two cases mentioned are handled correctly. Also, Safe Store bits in L1 need to be turned off.

As for in-order RAW and WAW dependences in the line-based protocol, the thread performing the second access is forwarded the entire line from the predecessor writer. This way we can reconcile versions correctly.

If we look at Table 4, we see that the number of WAW and RAW dependences, both same-word and false, is much higher than the number of same-word RAW dependences in all loops. Again, dependence distances are often short, which implies that these dependences can easily occur out of order. Consequently, the data suggests that the protocol with the per-line state will suffer more squashes than the one with per-word state and, therefore, have lower performance.

Figure 10 compares the execution times of our loops using our protocol (*Word*) and a variation with per-line state only (*Line*). Recall that the line size is 16 words. For each application, we use the best degree of unrolling observed in Section 5.2: 4 base iterations per thread in *Track*, *DSMC3D*, and *Euler*, and 2 base iterations per thread in *APSI* and *BDNA*. The bars are normalized and broken down as usual. Again, the number on top of the bars shows the speedup of the loops over sequential execution.

Figure 10 shows that maintaining the speculative state on a per-line basis results in poor performance. The numbers on top of the bars show that the loops now run even slower than the se-

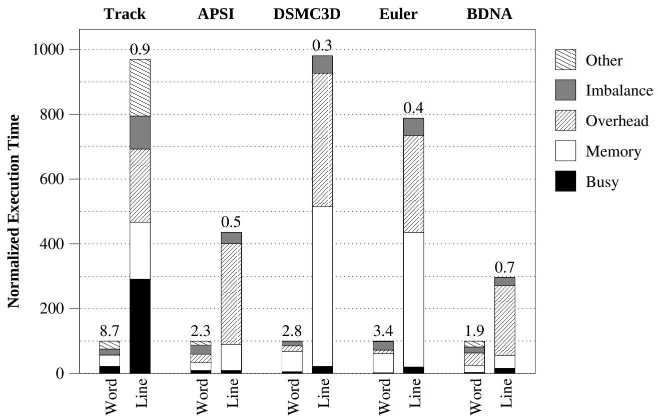

Figure 10: Effect of keeping the speculative state on a per-line basis as opposed to on a per-word basis. The numbers on top of the bars are the speedups over the sequential execution.

Application	*Grain*	*Squashes*	*Squashes by Loads (%)*
Track	*Word*	0.04	0
	Line	1279	28
APSI	*Word*	0	0
	Line	4737	0
DSMC3D	*Word*	13164	2.2
	Line	650473	7.3
Euler	*Word*	0	0
	Line	4429	24
BDNA	*Word*	0	0
	Line	118410	0

Table 6: Number of squash events per loop invocation.

quential case. The main reason for the slowdown is the many squashes suffered by the *Line* protocol. They are responsible for the large increase in *Busy*, *Memory*, and *Overhead* time relative to the *Word* protocol. The *Busy* and *Memory* time increase because of the instructions and memory accesses repeatedly performed by the squashed threads. The *Overhead* time increase is mostly due to thread squash overhead, including draining pending transactions in squashed threads, synchronizing GMDT modules, and restarting threads. Even if squashes did not increase *Overhead*, the repeated work would render the *Line* protocol much slower than the *Word* one.

To gain further insight, we can compare the number of squash events recorded in the two protocols. The column labeled *Squashes* in Table 6 shows the average number of squash events per loop invocation for each application in the *Line* and *Word* protocols. In *Word*, only *DSMC3D* and, to a much lower degree *Track* suffer squashes. These were the only two applications with same-word RAW dependences in Table 4. In *Line*, however, all applications suffer many squashes. This was expected from the many false RAW and both same-word and false WAW dependences in Table 4. Note that, in all but one loop, the number of squashes is higher than the number of iterations in Table 3. We conclude, therefore, that maintaining per-line speculative state is undesirable for these applications.

Finally, the last column of Table 6 shows what fraction of the squashes are triggered by loads. Recall from Section 3.2.1 that the loads in our protocol, if they hit in L1, return the data to the processor immediately, even if the Safe Load bit is zero. In the latter case, to hide latency, the MDTs are checked in the background. If the check shows that a stale version was used, a squash is generated. While the latency hiding effect of this approach can be beneficial, if it is to help, it should not cause too many squashes. Fortunately, we can see from Table 6 that only a small fraction of the squashes in our protocol (*Word*) are due to loads. The fraction is higher in *Line*.

6 RELATED WORK

An early proposal for hardware support for a form of speculative parallelization was made in [9] in the context of functional languages. More recently, the Multiscalar processor [16] was the first major work to use speculation within a single-chip multithreaded architecture, initially with the Address Resolution Buffer [3] and later with the Speculative Versioning Cache [4]. Other related designs have also been proposed [11, 14, 17, 20]. In most cases, the systems are designed with a tightly-coupled architecture in mind and do not scale beyond a small number of processors. Only [17] is designed purposedly for scalability. More recently, speculative CMPs have caught the attention of chip designers [1, 19].

The MDT-based CMP [11] is one example of such a speculative CMP. We borrow the MDT concept in our study and use this speculative CMP as the building block. Our scalable scheme, however, is not dependent on the MDT-based CMP and could easily accommodate the other speculative CMP proposals mentioned above.

The work in [17, 18] and in [22, 23, 24] presents extensions to a cache coherence protocol to accommodate speculation in scalable systems. Both designs yield a flat view of their speculation threads. Neither of these proposals is fleshed out enough to show how, if speculative CMPs were used as building blocks, it would reconcile its single layer protocol with many of the self-contained speculation protocols of these CMPs. Our work, instead, takes a hierarchical approach that largely abstracts away the internals of the node architecture. We have worked out a complete system that uses a self-contained speculative CMP as building block with minimal additions to interface with the rest of the system.

One difference with [17, 18] is that their work has taken the path of extending an existing cache-coherence protocol to handle speculation, while we have opted for the approach of adding a speculation protocol with minimal overlap over the existing cache coherence protocol. Finally, our protocol supports multiple versions and full per-word speculative state, which makes it less susceptible to squashing. In particular, out-of-order false dependences, whether RAW, WAR, or WAW, never cause a squash in our system. Their applications do not appear to require similar support.

Compared to the other scalable schemes, the work in [22, 23, 24] takes a slightly different approach. It has been specially designed to effectively handle workloads of threads that have much load imbalance and large working sets that overflow caches. In addition, it also provides support for parallel reduction operations. All this is done at the expense of utilizing more complex hardware than the other systems.

7 CONCLUSIONS

In this paper, we addressed the problem of extending speculative parallelization to scalable shared-memory systems. We presented a new scheme that requires relatively simple hardware and is efficiently integrated next to the cache coherence protocol of a conventional NUMA multiprocessor. We used a hierarchical approach to largely abstract away the internals of the node architecture. We were able to utilize a speculative CMP as building block with minimal additions to its interface with the rest of the system.

Detailed simulations of our scheme showed good overall performance. For a set of important non-analyzable scientific loops, we obtained average speedups of 4.2 for 16 processors. We found that assigning groups of iterations to each thread can improve performance. Finally, we showed that support for per-word speculative state is required, at least by our applications, to avoid excessive squashes. Overall, we feel that our design lies at a good complexity-performance design point.

REFERENCES

[1] H. Akkary and M. A. Driscoll. "A Dynamic Multithreading Processor." *Intl. Symp. on Microarchitecture*, pages 226-236, December 1998.

[2] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. "Advanced Program Restructuring for High-Performance Computers with Polaris." *IEEE Computer*, Vol. 29, No. 12, pages 78-82, December 1996.

[3] M. Franklin and G. Sohi. "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References." *IEEE Trans. on Computers*, Vol. 45, No. 5, pages 552-571, May 1996.

[4] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. "Speculative Versioning Cache." *Intl. Symp. on High Performance Computer Architecture*, pages 195-205, February 1998.

[5] M. Gupta and R. Nim. "Techniques for Run-Time Parallelization of Loops." *Supercomputing*, November 1998.

[6] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S-W. Liao, E. Bugnion, and M. Lam. "Maximizing Multiprocessor Performance with the SUIF Compiler." *IEEE Computer*, Vol. 29, No. 12, pages 84-89, December 1996.

[7] L. Hammond, M. Wiley, and K. Olukotun. "Data Speculation Support for a Chip Multiprocessor." *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 58-69, October 1998.

[8] N. P. Jouppi. "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers." *Intl. Symp. on Computer Architecture*, pages 364-373, May 1990.

[9] T. Knight. "An Architecture for Mostly Functional Languages." *ACM Lisp and Functional Programming Conference*, pages 500-519, August 1986.

[10] V. Krishnan and J. Torrellas. "A Direct-Execution Framework for Fast and Accurate Simulation of Superscalar Processors." *Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 286-293, October 1998.

[11] V. Krishnan and J. Torrellas. "A Chip-Multiprocessor Architecture with Speculative Multithreading." *IEEE Trans. on Computers, Special Issue on Multithreaded Architectures*, pages 866-880, September 1999.

[12] D. Lenoski, J. Laudon, K. Guarachorloo, A. Gupta, and J. Hennessy. "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor." *Intl. Symp. on Computer Architecture*, pages 148-159, May 1990.

[13] S.-T. Leung and J. Zahorjan. "Improving the Performance of Runtime Parallelization." *Symp. on Principles and Practice of Parallel Programming*, pages 83-91, May 1993.

[14] P. Marcuello and A. González. "Clustered Speculative Multithreaded Processors." *Intl. Conf. on Supercomputing*, pages 365-372, June 1999.

[15] L. Rauchwerger and D. Padua. "The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization." *SIGPLAN Conf. on Programming Language Design and Implementation*, pages 218-232, June 1995.

[16] G. Sohi, S. Breach, and T. Vijaykumar. "Multiscalar Processors." *Intl. Symp. on Computer Architecture*, pages 414-425, June 1995.

[17] J. G. Steffan and T. C. Mowry. "Architectural Support for Thread-Level Data Speculation." *Tech. Rep. CMU-CS-97-188, Carnegie Mellon University*, November 1997.

[18] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. "A Scalable Approach to Thread-Level Speculation." *Intl. Symp. on Computer Architecture*, June 2000.

[19] M. Tremblay. "MAJC: Microprocessor Architecture for Java Computing." Presentation at *Hot Chips*, August 1999.

[20] J.-Y. Tsai, J. Huang, C. Amlo, D. Lilja, and P.-C. Yew. "The Superthreaded Processor Architecture." *IEEE Trans. on Computers, Special Issue on Multithreaded Architectures*, Vol. 48, No. 9, pages 881-902, September 1999.

[21] J. Veenstra and R. Fowler. "A Front End for Efficient Simulation of Shared-Memory Multiprocessors." *Intl. Wksp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 201-207, January 1994.

[22] Y. Zhang, L. Rauchwerger, and J. Torrellas. "Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors." *Intl. Symp. on High-Performance Computer Architecture*, pages 161-173, February 1998.

[23] Y. Zhang, L. Rauchwerger, and J. Torrellas. "Hardware for Speculative Parallelization of Partially-Parallel Loops in DSM Multiprocessors." *Intl. Symp. on High-Performance Computer Architecture*, pages 135-139, January 1999.

[24] Y. Zhang. "Hardware for Speculative Parallelization in DSM Multiprocessors." *Ph.D. Thesis, University of Illinois at Urbana-Champaign, Department of Electrical and Computer Engineering*, May 1999.