Tradeoffs in Buffering Speculative Memory State for Thread-Level Speculation in Multiprocessors

MARíA JESÚS GARZARÁN
University of Illinois at Urbana-Champaign
MILOS PRVULOVIC
Georgia Institute of Technology
JOSÉ MARíA LLABERíA
Universitat Politècnica de Catalunya, Spain
VíCTOR VIÑALS
Universidad de Zaragoza, Spain
LAWRENCE RAUCHWERGER
Texas A&M University
and
JOSEP TORRELLAS
University of Illinois at Urbana-Champaign

Thread-Level Speculation (TLS) provides architectural support to aggressively run hard-to-analyze code in parallel. As speculative tasks run concurrently, they generate unsafe or speculative *memory state* that needs to be separately buffered and managed in the presence of distributed caches and buffers. Such a state may contain multiple versions of the same variable. In this paper, we introduce a novel taxonomy of approaches to buffer and manage multiversion speculative memory state in multiprocessors. We also present a detailed complexity-benefit tradeoff analysis of the different approaches. Finally, we use numerical applications to evaluate the performance of the approaches under a single architectural framework. Our key insights are that support for buffering the state of multiple speculative tasks and versions per processor is more complexity-effective than support

This paper extends an earlier version that appeared in the 9th International Symposium on High Performance Computer Architecture (HPCA), February 2003.
This work was supported in part by the National Science Foundation under grants EIA-0081307, EIA-0072102, and EIA-0103741; by DARPA under grant F30602-01-C-0078; by the Ministry of Education of Spain under grant TIN 2004-07739-C02-01|02; by the Diputacion General de Aragon of Spain BOA 20-4-2005; and by gifts from IBM and Intel.
Contact author's address: María Jesús Garzarán, University of Illinois at Urbana-Champaign, Department of Computer Science, 201 North Goodwin Avenue, Urbana, IL 61801; email: garzaran@cs.uiuc.edu.

for lazily merging the state of tasks with main memory. Moreover, both supports can be gainfully combined and, in large machines, their effect is nearly fully additive. Finally, the more complex support for storing future state in the main memory can boost performance when buffers are under pressure, but hurts performance when squashes are frequent.

Categories and Subject Descriptors: C.1.4 [**Processor Architectures**]: Parallel Architectures

General Terms: Design, Performance

Additional Key Words and Phrases: Caching and buffering support, coherence protocol, memory hierarchies, thread-level speculation, shared-memory multiprocessors

1. INTRODUCTION

Although parallelizing compilers have made significant advances, they still often fail to parallelize codes with accesses through pointers or subscripted subscripts, possible interprocedural dependences, or input-dependent access patterns. To parallelize these codes, researchers have proposed architectural support for Thread-Level Speculation (TLS). The approach is to build tasks from the code and speculatively run them in parallel, hoping not to violate sequential semantics. As tasks execute, special support checks that no cross-task dependence is violated. If any is, the offending tasks are squashed, the polluted state is repaired, and the tasks are re-executed. Many different schemes have been proposed, ranging from hardware-based [e.g. Akkary and Driscoll 1998; Cintra et al. 2000; Figueiredo and Fortes 2001; Franklin and Sohi 1996; Garzarán et al. 2003; Gopal et al. 1998; Hammond et al. 1998; Knight 1986; Krishnan and Torrellas 1999; Marcuello and González 1999; Prvulovic et al. 2001; Sohi et al. 1995; Steffan et al. 2000; Tremblay 1999; Tsai et al. 1999; Zhang et al. 1999] to software-based schemes [e.g. Frank et al. 2001; Gupta and Nim 1998; Rauchwerger and Padua 1995; Rundberg and Stenström 2000].

Each scheme for TLS has to solve two major problems: (1) detection of violations and, (2) if a violation occurs, state repair. Most schemes detect violations in a similar way: data that is speculatively accessed (e.g., read) is marked with some ordering tag, so that we can detect a later conflicting access (e.g., a write from another task) that should have preceded the first access in sequential order.

As for state repair, a key support is to buffer the *unsafe memory state* that speculative tasks generate as they execute. Typically, this buffered state is merged with main memory when speculation is proved successful, or is discarded when a violation is detected. In some programs, different speculative tasks running concurrently may generate different versions of the same variable. In other cases, multiple tasks that run on the same processor in sequence may create multiple versions of the same variable. In all cases, the TLS system must keep the state generated by each speculative task separate from each other. Moreover, it must organize such state such that reader tasks obtain the correct versions. Finally, versions must eventually merge into the safe state in order. All this is challenging in multiprocessors, given their distributed caches and buffers.

A variety of approaches to buffer and manage speculative memory state have been proposed. In some proposals, tasks buffer unsafe state dynamically

in caches [Cintra et al. 2000; Figueiredo and Fortes 2001; Gopal et al. 1998; Krishnan and Torrellas 1999; Steffan et al. 2000], write buffers [Hammond et al. 1998; Tsai et al. 1999] or special buffers [Franklin and Sohi 1996; Prvulovic et al. 2001] to avoid corrupting main memory. In other proposals, tasks generate a log of updates that allow them to backtrack execution in case of a violation [Frank et al. 2001; Garzarán et al. 2003; Zhang 1999; Zhang et al. 1999]. Often, there are large differences in the way caches, buffers, and logs are used in different schemes. Unfortunately, there is no study that systematically breaks down the design space of buffering approaches by identifying major design decisions and tradeoffs and provides a performance and complexity comparison of important design points.

This paper makes three contributions:

1. It performs a systematic analysis of the design space of buffering approaches in TLS, identifying major design decisions. We introduce a novel taxonomy of approaches to buffer and manage multiversion *speculative memory state* in multiprocessors.
2. It presents a detailed complexity-benefit tradeoff analysis of the different approaches.
3. It uses numerical applications to evaluate the performance of the approaches under a single architectural framework. In the evaluation, we examine both single- and multi-chip multiprocessors.

Our analysis shows that buffering the state of multiple speculative tasks and versions per processor is more complexity-effective than lazily merging the state of tasks with main memory. Moreover, both supports can be gainfully combined and, in large machines, their effect is nearly fully additive. Finally, the more complex support for storing future state in main memory can boost performance when buffers are under pressure, but hurts performance when squashes are frequent.

This paper is organized as follows: Section 2 introduces the challenges of buffering; Section 3 presents our taxonomy and the detailed complexity-benefit tradeoff analysis; Section 4 describes our evaluation methodology; Section 5 evaluates the different buffering approaches; and Section 6 concludes.

2. BUFFERING MEMORY STATE

2.1 Basics of Thread-Level Speculation

Thread-Level Speculation (TLS) extracts tasks from sequential codes and executes them in parallel hoping not to violate any sequential semantics. Under TLS, potentially dependent tasks execute in parallel. At run time, references from a task may have data dependences with references from other tasks. Thus, special software or hardware must ensure that such data dependences are handled properly, enforcing the sequential semantics of the original code.

At all times, tasks have a relative order imposed by the sequential code they come from. Consequently, we use the terms predecessor and successor tasks. If we give increasing IDs to successor tasks, the lowest-ID task still running is

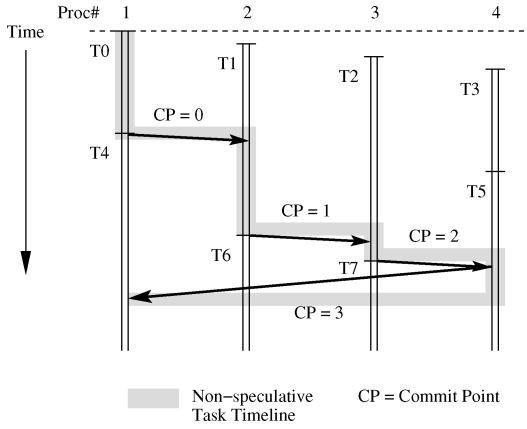


Fig. 1. A set of tasks executing on four processors. The commit point (CP) advance is shown in the nonspeculative task timeline.

nonspeculative, while the others are *speculative*. When the nonspeculative task finishes execution, it commits. Conceptually, commiting involves making the state generated by the task to be part of the safe state of the program. As part of committing, tasks update a variable called the *Commit Point* (*CP*), which identifies the committed task with the highest ID

When a speculative task finishes, it cannot commit until all its predecessors have finished and committed. However, to better tolerate load imbalance, the processor that ran it can start to execute another speculative task. At any time, the system contains many speculative tasks, either finished or unfinished, and one unfinished nonspeculative task. The set of all these tasks is called the *window of uncommitted tasks*.

Figure 1 shows an example of several tasks running on four processors. In this example, when task *T3* executing in processor 4 finishes the execution, it cannot commit. It has to wait until its predecessor tasks *T1* and *T2* also finish and commit. However, processor 4 may be able to start to execute task *T5*. The example also shows how the nonspeculative task status changes as tasks finish and commit (nonspeculative task timeline).

As speculative tasks execute in parallel, special software or hardware checks for cross-task data dependence violations. This is done by tracking memory references. The possible data dependences are WAR (Write-After-Read), WAW (Write-After-Write), and RAW (Read-After-Write), and they can occur in order or out of order. Typically, a violation will occur if these dependences execute out-of-order. However, out of order WAR and WAW dependences can be handled at run-time and not induce violations in systems with support for multiple versions of the same datum. In these systems, as tasks execute they generate versions that are usually tagged with the ID of the producer task. WAR and WAW dependences in these systems are handled by effectively renaming the versions in hardware.

RAW dependences are harder to deal with. In an in-order RAW, the speculation protocol must find the correct version and supply it to the reader task. In an out-of-order RAW, a task has prematurely loaded a variable that is

subsequently modified by a predecessor task. The speculation protocol must detect this error and the reader task needs to be squashed. Ordinarily, all the successors tasks are also squashed because they may have consumed data produced by the squashed task. While it is possible to resolve reference chains and selectively squash only tasks at fault, it involves extra complexity that we avoid. In any case, when a task is squashed, all its speculatively produced data must be purged. The task can then be restarted.

Many different schemes have been proposed for TLS, ranging from hardware-based to software-only, and targeting machines that range from chip multiprocessors to larger shared-memory machines. In hardware-based approaches, dependence checks are piggybacked on top of the coherence protocol messages. Moreover, coherence messages contain the ID of the task that issued the operation in addition to the type of operation, and the referenced memory address. If the message is a write and a successor task has already read the same data (out-of-order RAW), the reader task and its successors are squashed. If the message is a read and the data was previously written by another task (in-order RAW), the speculation protocol finds the correct data version.

2.2 Challenges in Buffering State in TLS

We identify five main difficulties in buffering state in a TLS multiprocessor memory system.

2.2.1 *Separation of Task State.* The state produced by a speculative task is unsafe, since the task may be squashed. Therefore, such state is typically kept separate from that of other tasks and main memory by buffering it in caches [Cintra et al. 2000; Figueiredo and Fortes 2001; Gopal et al. 1998; Krishnan and Torrellas 1999; Steffan et al. 2000] or special buffers [Franklin and Sohi 1996; Hammond et al. 1998; Prvulovic et al. 2001; Tsai et al. 1999]. Alternatively, the task state is merged with memory, but the memory overwritten in the process is saved in an undo log [Frank et al. 2001; Garzarán et al. 2003; Zhang 1999; Zhang et al. 1999].

2.2.2 *Multiple Versions of the Same Variable in the System.* A *new version* of a variable appears in the system when a task writes for the first time to that variable. Thus, two speculative tasks running on different processors may create two different versions of the same variable [Cintra et al. 2000; Gopal et al. 1998]. These versions need to be buffered separately and special actions need to be designed so that a reader task can find the correct version out of the several coexisting in the system. Such version will be the version created by the task with the largest ID that is still a predecessor of the reader task. Figure 2 shows an example where tasks *T0*, *T1*, and *T3* create three different versions of the variable x and task *T2* reads variable x. For correctness, the version produced by task *T1* must be provided.

Notice that a task has, at most, a single version of any given variable. A task can write several times to the same variable, but the processor only needs to keep the latest version produced by that task. The reason is that, on a dependence violation, the whole task is undone. Therefore, there is no need to keep

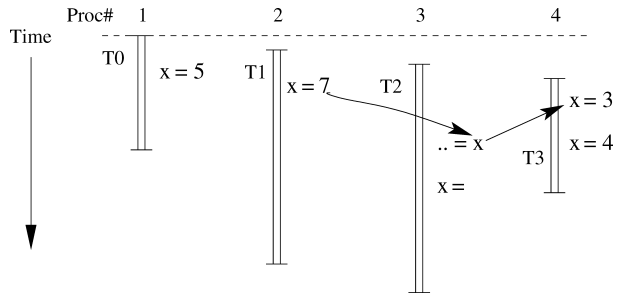


Fig. 2. Multiple versions of the same variable in the system. The figure shows examples of in-order RAW, out-of-order WAR, and WAW dependences.

Task–ID	Tag	Data
i	0x600	2
i	0x400	2
j	0x800	2

(a)

Task–ID	Tag	Data
i	0x400	2
j	0x400	4
k	0x400	6

(b)

Fig. 3. Example of a cache keeping state from several tasks (a) or state from several tasks and multiple versions of the same variable (b).

the other intermediate versions that the task produced. In Figure 2, task *T3* writes twice to the same variable. However, the processor only buffers the value 4 written by the last write.

2.2.3 *Multiple Speculative Tasks per Processor.* When a processor finishes executing a task, the task may still be speculative. If the buffering support is such that the processor can only hold state from a single speculative task, the processor stalls until the task commits. However, to better tolerate task load imbalance, the local buffer may have been designed to buffer state from several speculative tasks, enabling the processor to execute another task. In this case, the state of each task is tagged with the ID of the owner task (Figure 3a).

2.2.4 *Multiple Versions of the Same Variable in a Single Processor.* When a processor buffers state from multiple speculative tasks, it is possible that it may even have to hold multiple versions of the same variable (Figure 3b). This may occur in load-imbalanced applications with WAW dependences between tasks.

Note that when a task creates its own version, any subsequent load of the variable by that task will always read from its own version. However, if the task reads a variable that it did not first write, it needs to send an external request so that the correct version can be provided. Thus, on a cache read from the local processor, the data is provided only if both address and task ID match. On a external request, extra comparisons need to be done if the cache has two versions of the same variable. For example, in Figure 4, the cache of processor

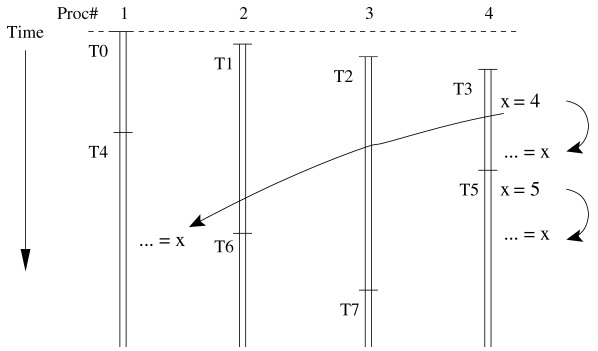


Fig. 4. External request to a cache that has two versions of the same variable x.

Table I. Application Characteristics that Illustrate the Challenges of Buffering

	# Spec Tasks (Average)		Written Footprint per Spec Task (Average)	
Application	In System	Per-Proc	Total (KB)	Mostly Priv (%)
P3m	800.0	50.0	1.7	87.9
Tree	24.0	1.5	0.9	99.5
Bdna	25.6	1.6	23.7	99.4
Apsi	28.8	1.8	20.0	60.0
Track	20.8	1.3	2.3	0.6
Dsmc3d	17.6	1.1	0.8	0.5
Euler	17.4	1.1	7.3	0.7

4 has two versions of variable x, one created by task *T3* and one by task *T5*. If task *T4* running on processor 1 reads x, it needs to get the version from task *T3*.

2.2.5 *Merging of Task State.* When a task commits, its state can be merged with the safe state in main memory. Since this merging is done frequently, it should be efficient. Furthermore, if the system (and the buffer) can have multiple versions of the same variable, they must be merged with memory following task order.

2.3 Application Behavior

To gain insight into these challenges, Table I shows some application characteristics. The applications (discussed in Section 4.2) execute speculatively parallelized loops on a simulated 16-processor machine (discussed in Section 4.1). Columns 2 and 3 show the average number of speculative tasks that coexist in the system and per-processor, respectively. In most applications, there are 17–29 speculative tasks in the system at a time, while each processor buffers about 1 to 2 speculative tasks at a time. Only *P3m*, which is very imbalanced, has many tasks.

Columns 4 and 5 show the size of the written footprint of a speculative task and the fraction of such footprint that exhibits mostly privatization access patterns, respectively. The written footprint is an indicator of the buffer size needed per task. Accesses to a variable follow a privatization pattern when one or more

```
speculative_parallel do i
  do j
    do k
      work(k) = work (f(i,j,k))
    end do
    call foo(work(j))
  end do
end do
```

(a) Apsi

```
speculative_parallel do i
  s1 = ihits(1,i)
  if (nused(s1) .le. 0) then
    nused(s1) = nused(s1) - 3
  endif
enddo
```

(b) Track

Fig. 5. Examples of nonanalyzable loops. The outermost loop is speculatively parallelized.

tasks access the variable, but they all write it before reading it. In some cases, accesses to a variable follow such a pattern, but either they do not follow it all the time, or the compiler cannot prove that they follow it all the time. In this case, we call the pattern mostly privatization. This pattern appears in tasks with many WAW dependences. Such tasks create many versions of the same variable.

As an example of code with mostly privatization patterns, Figure 5a shows a loop from *Apsi*. Each task generates its own *work(k)* elements before reading them. However, compiler analysis fails to prove *work* as privatizable, because it is unable to guarantee that there is no intersection between the *work(f(i,j,k))* elements that are read and the *work(k)* elements that are written. Table I shows that mostly privatization patterns play a major role in *P3m*, *Tree*, *Bdna*, and, to a lesser extent, in *Apsi*, increasing the complexity of buffering.

There are some applications without privatization patterns: *Track*, *Dsmc3d*, and *Euler*. In these applications, each task reads and writes relatively scattered elements in arrays. Thus, sometimes there is no intersection between the elements that each task accesses and, as a result, there are no data dependences among tasks. Other times, more than one task access the same data. However, if the colliding tasks are far apart, they will not overlap during execution and data dependences will execute in order. On the other hand, if the colliding tasks are close, they may overlap during the execution and data dependences may execute out of order. Figure 5b shows a loop from *Track*, where array *nused* is accessed with these patterns. Specifically, each iteration accesses one element of the array. The actual element accessed depends on the values of the array *ihits*.

3. TAXONOMY AND TRADEOFF ANALYSIS

To understand the tradeoffs in buffering speculative memory state under TLS, we present a novel taxonomy of possible approaches (Section 3.1), map existing schemes to it (Section 3.2), and perform a tradeoff analysis of benefits and complexity (Section 3.3).

3.1 Novel Taxonomy of Approaches

We propose two axes to classify the possible approaches to buffering: how the speculative task state in an individual processor is separated and how the task state is merged system-wide. The taxonomy is shown in Figure 6a.

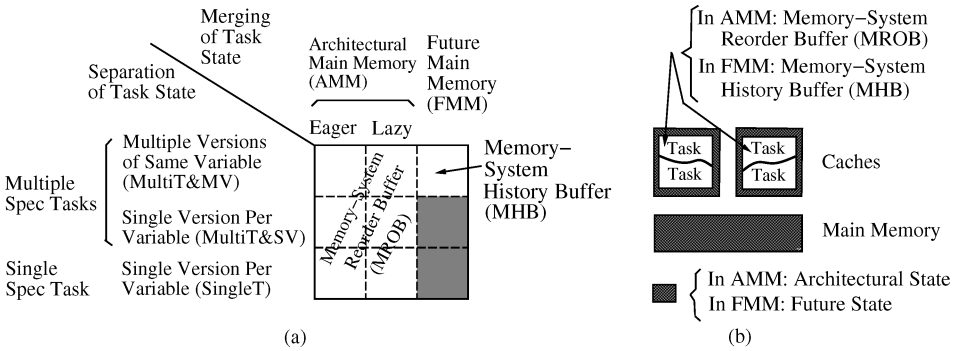


Fig. 6. Buffering and managing speculative memory state: taxonomy (a) and difference between architectural (AMM) and future main memory (FMM) schemes (b).

3.1.1 *Separation of Task State.* The vertical axis classifies the approaches based on how the speculative state in the buffer (e.g., cache) of an individual processor is separated: the buffer may be able to hold only the state of a single speculative task at a time (*SingleT*), multiple speculative tasks but only a single version of any given variable (*MultiT&SV*), or multiple speculative tasks and multiple versions of the same variable (*MultiT&MV*).

In SingleT systems, when a processor finishes a speculative task, it has to stall until the task commits. Only then can the processor start a new speculative task. In the other schemes, when a processor finishes a speculative task, it can immediately start a new one.[1] In MultiT&SV schemes, however, the processor stalls when a local speculative task is about to create its own version of a variable that already has a speculative version in the local buffer. The processor only resumes when the task that created the first local version becomes nonspeculative. In MultiT&MV schemes, each local speculative task can keep its own speculative version of the same variable.

3.1.2 *Merging of Task State.* The second (horizontal) axis classifies the approaches based on how the state produced by tasks is merged with main memory. This merging can be done strictly at task commit time (*Eager Architectural Main Memory*); at or after the task commit time (*Lazy Architectural Main Memory*); or at any time (*Future Main Memory*). We call these schemes Eager AMM, Lazy AMM, and FMM, respectively.

The largest difference between these approaches is on whether the main memory contains only safe data (Eager or Lazy AMM) or it can contain speculative data as well (FMM). To help understand this difference, we use an analogy with the concepts of architectural file, reorder buffer, future file, and history buffer proposed by Smith and Pleszkun for register file management Smith and Pleszkun [1988].

The architectural file in Smith and Pleszkun [1988] refers to the safe contents of the register file. The architectural file is updated with the result of an

[1]Intuitively, in *SingleT* schemes, the assignment of tasks to processors is "physical" or tied to a predetermined ordering of round-robin processors after the first round, while in *MultiT* schemes, it is "virtual" or flexible.

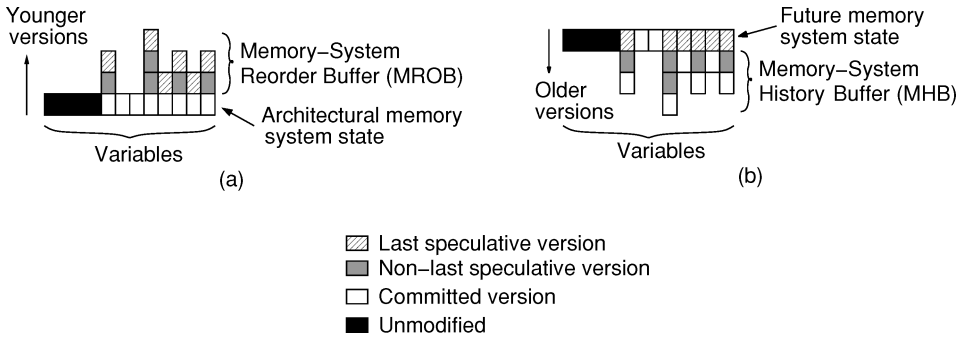


Fig. 7. Snapshot of the memory system state of a program under TLS using the concepts of Memory-System Reorder Buffer (a) and Memory-System History Buffer (b).

instruction only when the instruction has completed and all previous instructions have already updated the architectural file. The reorder buffer allows instructions to execute speculatively without modifying the architectural file. The reorder buffer keeps the register updates generated by instructions that have finished but are still speculative. When an instruction commits, its result is moved into the architectural file.

Analogously, in systems with *Architectural Main Memory* (AMM), all speculative versions remain in caches or buffers that are kept separate from the coherent main memory state. Only when a task becomes safe can its buffered state be merged with main memory. In this approach, caches or buffers become a distributed *Memory-System Reorder Buffer (MROB)*. Figure 7a shows a snapshot of the memory system state of a program using this idea. The architectural state is composed of unmodified variables (black region) and the committed versions of modified variables (white region). The remaining memory system state is comprised of speculative versions. These versions form the distributed MROB.

In Smith and Pleszkun [1988], the result of an instruction updates the architectural file at commit time. A straightforward approach in TLS closely follows this analogy. Specifically, when a task commits, its entire buffered state is eagerly merged with main memory. Such an approach we call *Eager AMM*. Merging may involve write back dirty lines to memory [Cintra et al. 2000] or requesting ownership for these lines to obtain coherence with main memory [Steffan et al. 2000].

Unfortunately, the state of a task can be large. Merging it all as the task commits delays the commit of future tasks. To solve this problem, we can allow the data versions produced by a committed task to remain in the cache where they are kept *incoherent* with other committed versions of the same variables in other caches or main memory. Committed versions are lazily merged with main memory later, usually as a result of line displacements from the cache or external requests. As a result, several different *committed* versions of the same variable may temporarily coexist in different caches and main memory. However, it is clear at any time which one is the latest one [Gopal et al. 1998; Prvulovic et al. 2001]. We call this approach *Lazy AMM*.

Consider now the future file in Smith and Pleszkun [1988]. It is the most recent contents of the register file. A future file entry is updated by the youngest instruction in program order updating that register. The future file is used as the working file by later instructions. When an instruction commits, no data movement or copying is needed because the future file has already been updated. However, the future file is unsafe: it is updated by uncommitted instructions. The history buffer allows the future file to be speculatively updated. The history buffer stores the previous contents of registers updated by speculative instructions. When an instruction commits, its history buffer entry is freed up. In an exception, the history buffer is used to revert the future file to the architectural file.

Analogously, in systems with *Future Main Memory* (FMM), versions from speculative tasks can be merged with the coherent main memory state. However, to enable recovery from task squashes, before a task generates a speculative version of a variable, the previous version of the variable is saved in a buffer or cache. This state is kept separate from the main memory state. Now, caches or buffers become a distributed *Memory-System History Buffer (MHB)*. Figure 7b shows a snapshot of the memory system state of a program using this idea. The future state is composed of unmodified variables, last speculative versions, and committed versions of modified variables that have no speculative version. The remaining speculative and committed versions form the MHB.

It is important to note that, in all cases, part of the coherent main memory state (architectural state in AMM systems and future state in FMM systems) can temporarily reside in caches (Figure 6b). This is because caches also function in their traditional role of extensions to main memory.

Finally, we shade SingleT FMM and MultiT&SV FMM schemes in Figure 6a to denote that they are relatively less interesting. We discuss why in Section 3.3.4.

3.2 Mapping Existing Schemes to the Taxonomy

Figure 8 maps existing schemes for TLS in multiprocessors onto our taxonomy.[2] Consider first SingleT Eager AMM schemes. They include Multiscalar with hierarchical ARB [Franklin and Sohi 1996], Superthreaded [Tsai et al. 1999], MDT [Krishnan and Torrellas 1999], and Marcuello99 [Marcuello and González 1999]. In these schemes, a per-processor buffer contains *speculative* state from, at most, a single task (SingleT) and this state is eagerly merged with main memory at task commit (Eager AMM). These schemes buffer the speculative state of a task in different parts of the cache hierarchy: one stage in the global ARB of a hierarchical ARB in Multiscalar, the Memory Buffer in Superthreaded, the L1 in MDT, and the register file (plus a shared Multi-Value cache) in Marcuello99.

Multiscalar with SVC [Gopal et al. 1998] is SingleT Lazy AMM because a processor cache contains *speculative* state from, at most, a single task (SingleT),

[2]Note that we are only concerned with the way in which the schemes buffer speculative memory state. Any other features, such as support for interprocessor register communication, are orthogonal to our taxonomy.

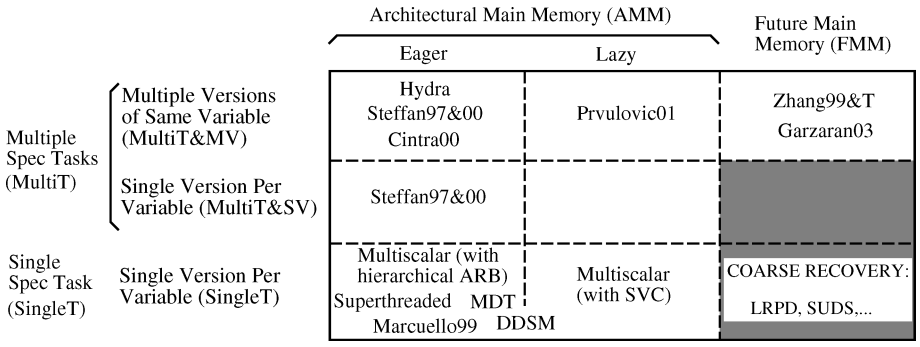


Fig. 8. Mapping schemes for TLS in multiprocessors onto our taxonomy.

while committed versions linger in the cache after the owner task commits (Lazy AMM). In DDSM [Figueiredo and Fortes 2001], speculative versions are also kept in caches. It is, therefore, AMM. However, work is partitioned so that each processor only executes a single task per speculative section. Since each processor commits only once, the distinction between Eager and Lazy does not apply.

MultiT&MV AMM schemes include Hydra [Hammond et al. 1998], Steffan97&00 [Steffan et al. 2000, 1997], Cintra00 [Cintra et al. 2000], and Prvulovic01 [Prvulovic et al. 2001]. Hydra stores speculative state in buffers between L1 and L2, while the other schemes store it in L1 and, in some cases, L2. A processor can start a new speculative task without waiting for the task that it has just run to become nonspeculative.[3] All schemes are MultiT&MV because the private cache hierarchy of a processor may contain state from multiple speculative tasks, including multiple speculative versions of the same variable. This requires appropriate cache design in Steffan97&00, Cintra00, and Prvulovic01. In Hydra, the implementation is easier because the state of each task goes to a different buffer. Two buffers filled by the same processor can contain different versions of the same variable.

Of these schemes, Hydra, Steffan97&00, and Cintra00 eagerly merge versions with main memory. Merging involves writing the versions to main memory in Hydra and Cintra00, or asking for the owner state in Steffan97&00. Prvulovic01 is Lazy: committed versions remain in caches and are merged when they are displaced or when caches receive external requests.

One of the designs in Steffan97&00 [Steffan et al. 2000, 1997] is MultiT&SV. The cache is not designed to hold multiple speculative versions of the same variable. When a task is about to create a second local speculative version of a variable, it stalls.

MultiT&MV FMM schemes include Zhang99&T [Zhang 1999; Zhang et al. 1999] and Garzaran03 [Garzarán et al. 2003]. In these schemes, task state is merged with main memory when lines are displaced from the cache or are requested externally, regardless of whether the task is speculative or not. The

[3]While this statement is true for Hydra in concept, the evaluation in Hammond et al. [1998] assumes only as many buffers as processors, making the system SingleT.

Table II. Different Supports Required

Support	Description
Cache Task ID (CTID)	Storage and checking logic for a task-ID field in each cache line
Cache Retrieval Logic (CRL)	Advanced logic in the cache to service external requests for versions
Memory Task ID (MTID)	Task ID for each speculative variable in memory and needed comparison logic
Version Combining Logic (VCL)	Logic for combining/invalidating committed versions
Undo Log (ULOG)	Logic and storage to support logging

Table III. Benefits Obtained and Support Required for Each of the Different Mechanisms

Upgrade	Performance Benefit	Additional Support Required
SingleT $\longrightarrow$ MultiT&SV	Tolerate load imbalance when there are no mostly privatization access patterns	CTID
MultiT&SV $\longrightarrow$ MultiT&MV	Tolerate load imbalance even when there are mostly privatization access patterns	CRL
Eager AMM $\longrightarrow$ Lazy AMM	Remove commit wavefront from critical path	CTID and (VCL or MTID)
Lazy AMM $\longrightarrow$ FMM	Faster version commit but slower version recovery	ULOG and (MTID if Lazy AMM had VCL)

MHB in Zhang99&T is kept in hardware structures called logs. In Garzaran03, the MHB is a set of software log structures, which can be in caches or displaced to memory.

Finally, there is a class of schemes labeled *Coarse Recovery* in Figure 8 that is different from those discussed so far. These schemes only support *coarse-grain* recovery. The MHB can only contain the state that existed before the speculative section. In these schemes, if a violation occurs, the state reverts to the beginning of the entire speculative section. These schemes typically use no hardware support for buffering beyond plain caches. In particular, they rely on software copying to create versions. The coarse recovery makes them *effectively* SingleT. Examples of such schemes are LRPD [Rauchwerger and Padua 1995], SUDS [Frank et al. 2001], and other proposals [Gupta and Nim 1998; Rundberg and Stenström 2000].

3.3 Tradeoff Analysis of Benefits and Complexity

To explore the design space of Figure 6a, we start with the simplest scheme (SingleT Eager AMM) and progressively complicate it. For each step, we consider performance benefits and support required. Tables II and III summarize the analysis.

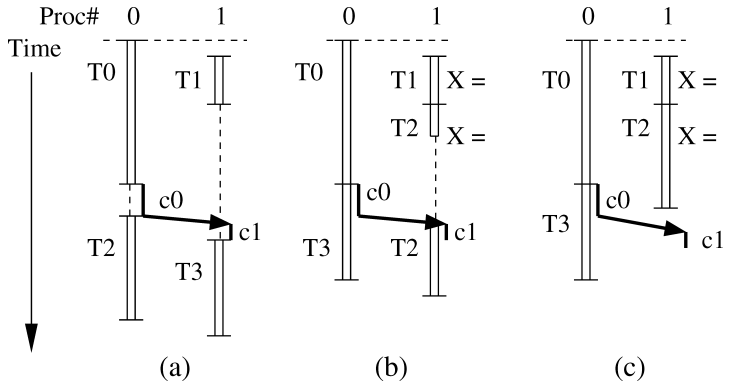


Fig. 9. Four tasks executing under SingleT (a), MultiT&SV (b), and MultiT&MV (c).

3.3.1 *Implementing a SingleT Eager AMM Scheme.* In SingleT Eager AMM schemes, each task stores its state in the local MROB (e.g., the processor's cache). When the task commits, its state is merged with main memory. If a task finishes while speculative, the processor stalls until it can commit the task. If the state of the task does not fit in the cache, the processor stalls until the task becomes nonspeculative to avoid polluting memory. Finally, recovery involves invalidating at least all dirty lines in the cache that belong to the squashed task.

3.3.2 *Multiple Speculative Tasks and Versions per-Processor.* The benefit of these schemes is that they tolerate load imbalance and mostly-privatization patterns; the supports required are Cache Task ID and Cache Retrieval Logic. We consider each issue in turn.

Benefits. SingleT schemes may perform poorly if tasks have a load imbalance: a processor that has completed a short speculative task has to wait for the completion of all (long) predecessor tasks running elsewhere. Only when the short task finally commits can the processor start a new task. For example, consider Figure 9, where T_i and c_i mean execution and commit, respectively, of task i. Figure 9a shows a SingleT scheme: processor *1* completes task *T1* and waits; when it receives the commit token, it commits *T1* and starts *T3*.

MultiT schemes do not need to slow down under load imbalance because processors that complete a speculative task can immediately start a new one. However, MultiT&SV schemes can still run slowly if tasks have *both* load imbalance and create multiple versions per variable. The latter occurs, for example, under mostly privatization patterns (Section 2.3). In this case, a processor stalls when a task is about to create a second local speculative version of a variable. When the task that created the first version becomes nonspeculative and, as a result, the first version can merge with memory, the processor resumes.

As an example, Figure 9b shows that processor *1* generates a version of *X* in *T1*. Then, it executes *T2*, but stalls when it is about to generate a second version of *X* in *T2*. When processor *1* receives the commit token for *T1*, the first version of *X* is merged with memory and *T2* restarts.

Under MultiT&MV, load imbalanced tasks do not cause stalls, even if they have mostly privatization patterns. An example is shown in Figure 9c. The result is faster execution.

Supports. In MultiT schemes, the cache hierarchy of a processor holds *speculative* versions from the multiple tasks that the processor has been executing. As a result, each line (or variable in some protocols) must be tagged with the owner task ID. Furthermore, when the cache is accessed, the address tag and task ID of the chosen entry are compared to the requested address and the ID of the requester task, respectively. This support we call Cache Task ID (CTID) in Table II.

Access from the local processor hits only if both tag and task ID match. In an external access, the action is different under MultiT&SV and MultiT&MV. Under MultiT&SV, the cache hierarchy can only keep a single version of a given variable. Therefore, an external access can trigger, at most, one address match. In this case, the relative value of the IDs indicates if the external access is out of order. If it is, a squash may be required. Otherwise, the data may be safely returned.

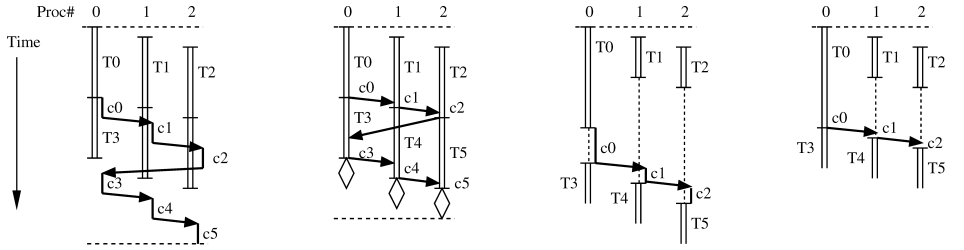
Under MultiT&MV, a cache hierarchy can hold multiple entries with the same address tag and different task ID. Such entries can go to different lines of the same cache set [Cintra et al. 2000; Steffan et al. 1997]. In this case, access to the cache may hit in several lines. Consider the case of an external read request. The cache controller has to identify which of the selected entries has the highest task ID that is still lower than the requester's ID. That one is the correct version to return. This operation requires some comparisons that may increase cache occupancy or more hardware for parallel comparisons. Furthermore, responses may require combining different words from the multiple cached versions of the requested line, depending on the speculative cache coherence protocol. This support we call Cache Retrieval Logic (CRL) in Table II.

3.3.3 *Lazy Merging with Architectural Main Memory (AMM).* The benefit of these schemes is that they remove the commit wavefront from the critical path; the supports required are Cache Task ID and Version Combining Logic (or Memory Task ID). We consider each issue in turn.

Benefits. Program execution under TLS involves the concurrent advance of two wavefronts: the *Execution Wavefront* advances as processors execute tasks in parallel, while the *Commit Wavefront* advances as tasks commit in strict sequence by passing the commit token. Figure 10a shows the wavefronts for a MultiT&MV Eager AMM scheme.

Under Eager AMM schemes, before a task passes the commit token to its successor, the task needs to write the data it wrote back to memory [Cintra et al. 2000] or get ownership for it [Steffan et al. 2000]. These operations may cause the commit wavefront to appear in the critical path of program execution. Specifically, they do it in two cases.

In one case, the commit wavefront appears at the end of the speculative section (Figure 10a). To understand this case, we call *Commit/Execution Ratio* the ratio between the average duration of a task commit and a task execution.



(a) MultiT&MV Eager AMM (b) MultiT&MV Lazy AMM (c) SingleT Eager AMM (d) SingleT Lazy AMM

Fig. 10. Progress of the execution and commit wavefronts under different schemes.

For a given machine, this ratio is an application characteristic that roughly measures how much state the application generates per unit of execution. If the Commit/Execution Ratio of the application, multiplied by the number of processors, is higher than 1, the commit wavefront can significantly delay the end of the speculative section (Figure 10a).

The second case occurs when the commit wavefront delays the restart of processors stalled because of the load-balancing limitations of MultiT&SV or SingleT (Figure 10c). In these schemes, a processor may have to stall until it receives the commit token and, therefore, commits are in the critical path.

Under Lazy AMM, committed versions generated by a task are lazily merged with main memory, on demand. Since commit now only involves passing the commit token, the commit wavefront advances fast and can hardly affect the critical path. As an example, Figures 10b and d correspond to Figures 10a and c, respectively, under Lazy AMM. In Figure 10b, instead of a long commit wavefront at the end of the speculative section, we have a final merge of the versions still remaining in caches [Prvulovic et al. 2001]. This is shown in the figure using diamonds. In Figure 10d, the commit wavefront affects the critical path minimally. In both cases, the program runs faster.

Supports. Lazy schemes present two challenges. The first one is to ensure that different versions of the same variable are merged into main memory in version order. Such in order merging must be explicitly enforced, given that committed versions are lazily written back to memory on displacement or external request. The second challenge is to find the latest committed version of a variable in the machine; the difficulty is that several different committed versions of the same variable can coexist in the machine.

These challenges are addressed with two supports: logic to combine versions (Version Combining Logic or VCL in Table II) and logic for ordering the versions of a variable. Different implementations of these two supports are proposed by Prvulovic01 [Prvulovic et al. 2001] and Multiscalar with SVC [Gopal et al. 1998].

When a committed version is displaced from a cache, the VCL identifies the latest committed version of the same variable still in the caches, writes it back to memory, and invalidates the other versions [Gopal et al. 1998; Prvulovic et al. 2001]. This prevents the earlier committed versions from overwriting memory later. A similar operation occurs when a committed version in a cache is requested by a processor. Note that if the machine uses multiword cache lines on

displacements and requests, the VCL has to collect the latest committed versions for *all* the words in the line from the caches and combine them [Prvulovic et al. 2001].

For the VCL to work, it needs the second support indicated above: support to order the different committed versions of the variable. This can be accomplished by tagging all the versions in the caches with their task IDs. This support is used by Prvulovic01 [Prvulovic et al. 2001] and was called CTID in Table II. An alternative approach used by Multiscalar with SVC [Gopal et al. 1998] is to link all the cached versions of a variable in an ordered linked list called VOL. The relative version order is determined by the location of the version in the list. This support is harder to maintain than CTID, especially when a cache can hold multiple versions of the same variable. As a result, we only list CTID in Table III.

We note that the version-combining support provided by VCL can instead be provided by a scheme proposed in Zhang99&T [Zhang 1999]. The idea is for main memory to selectively reject write-backs of versions. Specifically, for each variable under speculation, main memory keeps a task-ID tag that indicates what version the memory currently has. Moreover, when a dirty line is displaced from a cache and written back to memory, the message includes the producer task's ID (from CTID). Main memory compares the task ID of the incoming version with the one already in memory. The write-back is discarded if it includes an earlier version than the one already in memory. This support we call Memory Task ID (MTID) in Table II.

3.3.4 *Future Main Memory (FMM).* The benefit of these schemes is that they commit version faster, although they recover more slowly; the supports required are Cache Task ID, Memory Task ID, and Undo Log. In the following, we consider each issue in turn.

Benefits. In AMM schemes, when a new speculative version of a variable is created, it is simply written to the same address as the architectural version of the variable. However, it is kept in a cache or buffer until it can commit, to prevent overwriting the architectural version. Unfortunately, the processor may have to stall to prevent the displacement of such a version from the buffer. The problem gets worse when the buffer has to hold state from multiple speculative tasks. A partial solution is to provide a special memory area where speculative versions can safely overflow into [Prvulovic et al. 2001]. Unfortunately, such an overflow area is slow when asked to return versions, which especially hurts when committing a task. Overall, the process of going from a speculative to a committed version in AMM schemes carries the potential performance cost of stall to avoid overflows or of slow accesses to an overflow area.

In FMM schemes, the process of going from speculative to committed version is simpler and avoids penalizing performance. Specifically, when a task generates a new speculative version, the older version is *copied to another address* and the new version takes its place. The new version can be freely displaced from the cache at any time and written back to main memory. When the task commits, the version automatically becomes committed and nothing needs to be

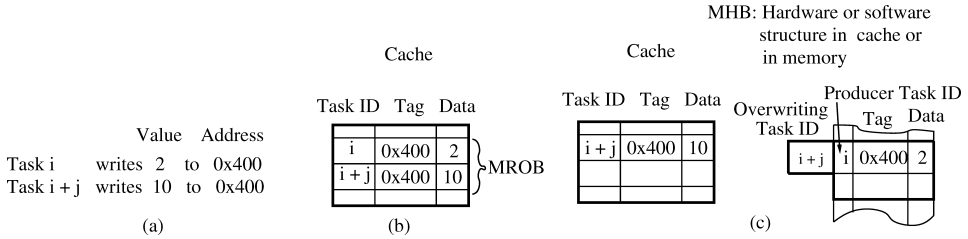


Fig. 11. Implementing the MROB and the MHB.

done. The older version can *also be safely displaced* from the cache and written back to memory at any time, since it lives in a different address. Hopefully, it is never reaccessed.

FMM, however, loses out in version recovery. AMM recovery simply involves discarding from the MROB (e.g., cache) the speculative versions generated by the offending task and successors. In contrast, FMM recovery involves restoring from the MHB to main memory all the versions overwritten by the offending task and successors in strict reverse task order.

Supports. Consider an example of a program where each task generates its own private version of variable *X*. Figure 11a shows the code for two tasks that run on the same processor. If we use an AMM scheme, Figure 11b shows the processor's cache and local MROB, assuming MultiT&MV support.

If we use an FMM scheme, Figure 11c shows the processor's cache and local MHB. The MHB is a hardware or software structure in the cache or in memory. When a task is about to generate its own version of a variable, the MHB saves the most recent local version of the variable (the one belonging to an earlier local task).

Note that we need to know what versions we have in the MHB. Such information is needed after a violation when, to recover the system, we need to reconstruct the total order of the versions of a variable *across* the distributed MHB. It is also needed in the rare case when a logged version is needed during normal execution. Such case, called retrieval, occurs at an in order RAW dependence if the requested version has been pushed into the log of the producer processor by a newer task that both runs on the producer processor and overwrote the variable.

Consequently, each MHB entry is tagged with the ID of the task that generated that version (*Producer Task ID i* in the MHB of Figure 11c). This ID cannot be deduced from the task that overwrites the version. Consequently, all versions in the cache must be tagged with their task IDs, so that the latter can be saved in the MHB when the version is overwritten. Finally, groups of MHB entries are also tagged with the *Overwriting Task ID* (*i+j* in the MHB of Figure 11c).

Overall, FMM schemes need three supports. One is a per-processor, sequentially accessed undo log that implements the MHB. When a task updates a variable for the task's first time, a log entry is created. Logs are accessed on recovery and on the rare retrieval operations. Both hardware [Zhang 1999;

Zhang et al. 1999] and software [Garzarán et al. 2003] logs have been proposed. This support is called Undo Log (ULOG) in Table II.

The second support (discussed above) is to tag all the versions in the caches with their task IDs. This is the CTID support in Table II. Unfortunately, such tags are needed even in SingleT schemes. This is unlike in the MROB, where SingleT schemes do not need task-ID tags. Therefore, SingleT FMM needs nearly as much hardware as MultiT&SV FMM, without the latter's potential benefits. The same can be shown for MultiT&SV FMM relative to MultiT&MV FMM. For this reason, we claim that the shaded area in Figure 6a is uninteresting (except for coarse recovery).

A third support is needed to ensure that main memory is updated with versions in increasing task-ID order for any given variable. Committed and uncommitted versions can be displaced from caches to main memory and the main memory has to always keep the latest future state possible. To avoid updating main memory out of task-ID order, FMM schemes [Garzarán et al. 2003; Zhang 1999; Zhang et al. 1999] use the Memory Task ID (MTID) support of Section 3.3.3 (see Table II). Note that the Version Combining Logic (VCL) of Section 3.3.3 is not an acceptable alternative to MTID in FMM schemes. The reason is that, under FMM, even an uncommitted version can be written to memory. In this case, earlier versions may be unavailable for invalidating/combining because they may not have been created yet. Consequently, VCL would not work.

3.3.5 *Discussion.* Table III can be used to qualitatively compare the implementation complexity of the different supports. We start with SingleT Eager AMM and progressively add features.

We argue that full support for multiple tasks&versions (MultiT&MV Eager AMM) is less complex than support for laziness (SingleT Lazy AMM): the former needs CTID and CRL, while the latter needs CTID and either VCL or MTID. CRL only requires *local* modification to the tag checking logic in caches, while VCL requires version combining logic in main memory, as well as global changes to the coherence protocol. The alternative to VCL is MTID, which is arguably more complex than VCL (Section 3.3.3). This is because MTID requires maintaining tags for regions of memory and comparison logic in the main memory. Such tags have to be maintained in the presence of page remapping, multiple speculative sections, etc.

Supporting multiple tasks&versions and laziness under AMM (MultiT&MV Lazy AMM) is less complex than supporting the FMM scheme. The latter needs all the support of the former (with MTID instead of VCL), plus ULOG.

3.4 Effect of Application Characteristics

Complexity considerations should be assessed against expected performance gains. In this section, we examine the main application characteristics that limit the performance of our schemes. A summary of our discussion is shown in Figure 12.

If we compare AMM to FMM systems, the main application characteristics that determine the performance are the frequency of true data dependences

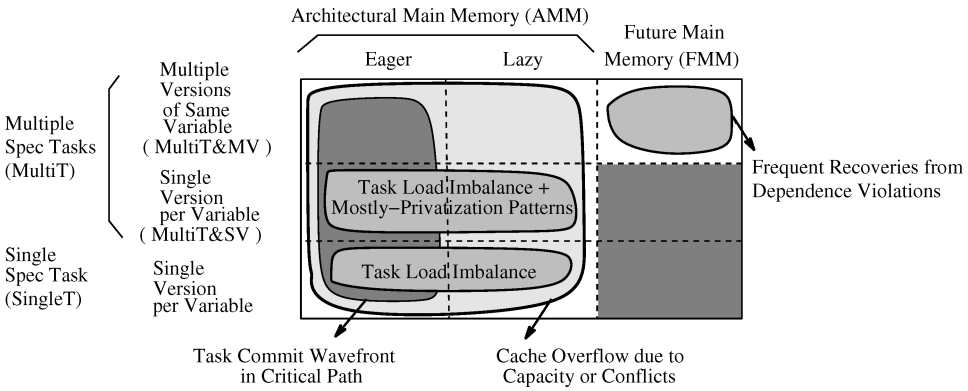


Fig. 12. Application characteristics that limit the performance of our schemes.

across neighboring tasks and the amount of speculative state generated by the tasks. If the frequency of dependences across neighboring tasks is high, some of them are likely to cause violations at run time, as tasks may execute out of order. If so, they will cause task squashes and recoveries, which are more expensive under FMM systems. On the other hand, if the amount of speculative state generated by the tasks is large, it may overflow the caches and the tasks will have to stall in AMM systems. Adding an overflow area in the local memory to hold an overflowing speculative state is only a partial solution, because versions still need to be accessed at commit time in AMM systems and such accesses are slow. The situation can get worse in MultiT schemes, where a cache can hold the speculative state of multiple tasks.

If we compare Eager to Lazy AMM schemes, the performance of Eager schemes will suffer if the Task Commit Wavefront appears in the critical path. This can occur under SingleT or MultiT&SV schemes for load-imbalanced applications. It also occurs under MultiT&MV schemes for applications with a high Commit/Execution Ratio if they execute with many processors.

Finally, we compare SingleT, MultiT&SV, and MultiT&MV AMM schemes. The performance of SingleT schemes is low for applications with task load imbalance. This shortcoming can be removed with MultiT schemes. However, if the support only includes MultiT&SV, and the application has mostly privatization access patterns, the performance will be similar to SingleT schemes. This problem is completely eliminated with MultiT&MV support.

All of these tradeoffs are evaluated in Section 5.

4. EVALUATION METHODOLOGY

4.1 Simulation Environment

We use execution-driven simulations to model two architectures: a multichip shared-memory machine with one processor per chip (SMP) and a chip multiprocessor (CMP). Both systems use four-issue dynamic superscalars with a 64-entry instruction window, 4 Int, 2 FP, and 2 Ld/St functional units, up to 8 pending loads and 16 stores, and a 2K-entry BTB with two-bit counters.

The SMP has 16 single-processor chips. Each chip has a two-way 32-Kbyte D-L1 and a four-way 512-Kbyte L2, both write-back with 64-byte lines. We use a small L2 because the applications's working sets are relatively small. The minimum round-trip latencies from a processor to the L1 and L2 are 2 and 12 cycles, respectively. The chips are connected with a two-dimensional (2D) mesh.

The CMP has eight cores. Each one has a two-way 32-Kbyte D-L1 and a four-way 256-Kbyte L2. As indicated above, L2s are small to account for the small problem size simulated. All caches are write-back with 64-byte lines. The L2s connect through a crossbar to eight on-chip banks of both directory and L3 tags. Each bank has its own control logic and private interface to connect with its corresponding data array bank of a shared off-chip L3. The L3 has four ways and holds 16 Mbytes. The minimum round-trip latencies from a processor to the L1, L2, another processor's L2, and L3 are 2, 8, 18, and 38 cycles, respectively.

Both systems have a high-performance main memory subsystem, with DDR2 DRAM. The front side bus has a width of 128 bits, while the DRAM has a bandwidth of 8.52 GB/sμ. The minimum round trip latency to the main memory for the CMP is 102 ns. For the SMP, it ranges from 75 ns for the closest memory module to 208 ns for the farthest one.

We model all the nonshaded buffering approaches in our taxonomy of Figure 6a. To model the approaches, we use a TLS protocol similar to Prvulovic et al. [2001], but without its support for high-level access patterns. The protocol is appropriately modified to adapt to each box in Figure 6a. Using the same base protocol for all cases is needed to evaluate the true differences between them. This protocol supports multiple concurrent versions of the same variable in the system, and triggers squashes only on out-of-order RAWs to the same word [Prvulovic et al. 2001]. It needs a single task-ID tag per cache line. We do not use any data dependence predictor. We avoid processor stalls in AMM due to L2 conflict or capacity limitations by using a per-processor overflow memory area similar to Prvulovic et al. [2001]. For FMM systems, the per-processor MHB is allocated in main memory.

In Eager AMM systems, each processor uses a hardware table to record the lines that a speculative task modifies in the L2 and overflow area. When the task commits, these lines are written back to main memory, either explicitly by the processor (SingleT schemes) or in the background by special hardware (MultiT schemes). If we changed our baseline speculative protocol, we could instead use the ORB table proposed by Steffan et al. [2000]. The ORB only contains modified lines that are not owned and triggers line ownership requests rather than write-backs. Using an ORB and a compatible speculation protocol may change the overheads of eager data merging relative to those measured in this paper. Quantifying these changes is beyond the scope of this paper.[4]

In all AMM and FMM systems, there is also a similar table for the L1. The table is traversed when a task finishes to write back modified lines to L2.

[4]We note that, for numerical codes like the ones considered in this paper (Section 4.2), the ORB has to hold many more lines that the number reported by Steffan et al. [2000], who used nonnumerical, fine-grained applications. Indeed, for numerical applications, Prvulovic et al. [2001] shows that the number of nonowned modified lines per speculative task is about 200, on average.

Table IV. Application Characteristics[a]

Appl	% of Tseq	# Invoc; # Tasks per Invoc	# Instr per Task (Thous.)	Commit/Exec Ratio (%) SMP	Commit/Exec Ratio (%) CMP	Appl Characteristics Load Imbal	Appl Characteristics Most Priv Pattern	Appl Characteristics Comm/ Exec Ratio
P3m	56.5	1;97336	69.1	0.3	0.1	High	Med	Low
Tree	92.2	41;4096	28.7	1.4	0.4	Med	High	Low
Bdna	44.2	1;1499	103.3	6.0	3.9	Low	High	Med
Apsi	29.3	900;63	102.6	11.4	6.1	Low	High	High-Med
Track	58.1	56;126	22.3	8.4	2.0	Med	Low	High-Med
Dsmc3d	41.2	80;46777	5.4	6.2	4.4	Low	Low	Med
Euler	89.8	120;1871	3.9	12.6	14.5	Low	Low	High
Average	58.8	171;21681	47.9	6.6	4.5			

[a]Each task is one iteration, except in *Track*, *Dsmc3d*, and *Euler*, where it is 4, 16, and 32 consecutive iterations, respectively. In *Apsi*, we use an input grid of $512 \times 1 \times 64$. In *P3m*, while the loop has 97,336 iterations, we only use the first 9000 iterations in the evaluation. In *Euler*, since all six loops have the same patterns, we only simulate dflux_do100. All the numbers except *Tseq* correspond to this loop. In the table, Med stands for Medium.

This table traversal takes largely negligible time, given the size of the tasks (Section 4.2).

Finally, our simulations model all overheads, including dynamic scheduling of tasks, task commit, and recovery from dependence violations. In FMM systems, recovery is performed using software handlers, whose execution is fully simulated. However, in all of our simulations, we assume that the TLS circuitry added to the caches does not increase their access time.

4.2 Applications

For the evaluation, we use a set of numerical applications. In each application, we use the Polaris parallelizing compiler [Blume et al. 1996] to identify the sections that are not fully analyzable by a compiler. Typically, these are sections where the dependence structure is either too complicated or unknown, for example, because it depends on input data or control flow. These code sections often include arrays with subscripted subscripts and conditionals that depend on array values.

The applications used are: *Apsi* from SPECfp2000, *Track* and *Bdna* from Perfect Club, *Dsmc3d* and *Euler* from HPF-2, *P3m* from NCSA, and *Tree* from Barnes [1994]. We use these applications because they spend a large fraction of their time executing code that is not fully analyzable by a parallelizing compiler. The only exception is *Bdna*, which has been shown parallelizable by research compiler techniques [Eigenmann et al. 1998], although no commercial compiler can parallelize it. Our application suite contains only numerical applications because our compiler infrastructure only allows us to analyze Fortran codes. While the results of our evaluation are necessarily a function of the architectures simulated and application domain used, we will see that our applications cover a very wide range of buffering behaviors.

Table IV characterizes the non analyzable sections of the applications. These sections are the following loops: pp_do100 (*P3m*), accel_do10 (*Tree*), actfor_do240 (*Bdna*), run_do [20,30,40,50,60,100] (*Apsi*), nlfilt_do300 (*Track*), move3_goto100

(*Dsmc3d*) and dflux_do[100,200], psmoo_do20 and eflux_do[100,200,300] (*Euler*). The speculative tasks are one or several consecutive iterations of these loops. The tasks are dynamically scheduled. Column 2 of Table IV lists the combined weight of these loops relative to *Tseq*, the total *sequential* execution time of the application with I/O excluded. This value, which is obtained on a workstation, is, on average, 58.8%. The table also shows the number of invocations of these loops during execution, the number of tasks per invocation, the number of instructions per task, and the ratio between the time taken by a task to commit and to execute (Commit/Execution Ratio). This ratio was computed under MultiT&MV Eager, where tasks do not stall. It is shown for both the SMP and CMP architectures. Finally, the last three columns give a qualitative measure of the load imbalance between nearby tasks, the weight of mostly privatization patterns, and the value of the Commit/Execution Ratio.

The applications exhibit a range of squashing behaviors. Specifically, *Euler's* execution is substantially affected by squashes, as it suffers 0.02 squashes per committed task. On the other hand, while the selected sections of *P3m*, *Tree*, *Bdna*, and *Apsi* are not fully analyzable, the interleaving of tasks is such that no squashes occur in the execution that we measure. Finally, *Track* and *Dsmc3d* have an intermediate behavior, as they suffer squashes but fewer than *Euler*.

All the data presented in Section 5, including speedups, refer only to the code sections in the table. Given that barriers separate analyzable from nonanalyzable code sections, the overall application speedup can be estimated by weighting the speedups that we show in Section 5 by the percentage of *Tseq* from the table.

5. EVALUATION

We first focus on the SMP system, and then evaluate the CMP in Section 5.3.

5.1 Separation of Task State under Eager AMM

Figure 13 compares the execution time of the nonanalyzable sections of the applications under schemes where individual processors support: a single speculative task (SingleT), multiple speculative tasks, but only single versions (MultiT&SV), and multiple speculative tasks and multiple versions (MultiT&MV). Both Eager and Lazy AMM schemes are shown for each case. The bars are normalized to SingleT Eager and broken down into: (a) instruction execution plus nonmemory pipeline hazards (Busy), (b) and stalls due to memory access, not enough task/version support, and end-of-loop stall due to commit wavefront or load imbalance (Stall). It is not possible for our simulator to accurately separate the components of the Stall category. The numbers on top of the bars are the speedups over sequential execution of the code where all data is in the memory module closest to the processor chip. In this section, we examine the Eager schemes, which are the bars in odd positions; we leave the Lazy ones for Section 5.2.

5.1.1 *Comparing MultiT&MV to SingleT.* MultiT&MV should perform better than SingleT in two cases. One is in highly load-imbalanced applications

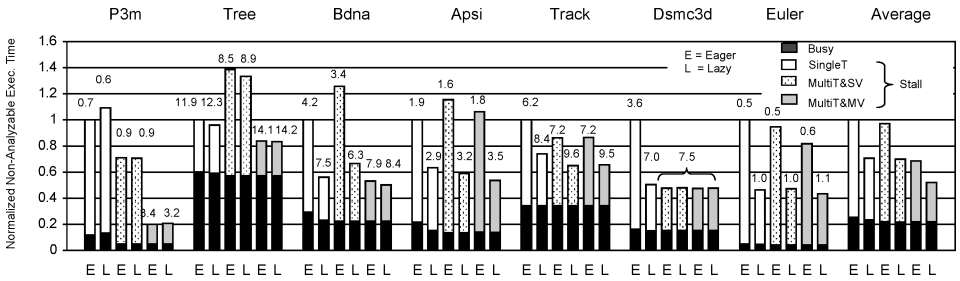


Fig. 13. Supporting single or multiple speculative tasks or versions per processor, for eager or lazy architectural main memory (AMM) schemes. In the figure, *E* and *L* stand for Eager and Lazy, respectively.

(Figure 9c vs 9a). According to Table IV, only *P3m* has high imbalance. As shown in Figure 13, MultiT&MV is faster than SingleT in *P3m*.

The other case is under modest load imbalance, but medium-sized Commit/Execution Ratio. The latter affects performance because commits in SingleT are in the critical path of restarting stalled processors (Figure 10c). MultiT&MV removes the commits from the critical path. Note, however, that the Commit/Execution Ratio should not be high. If it is, the end-of-loop commit wavefront eliminates any gains of MultiT&MV (Figure 10a). According to Table IV, *Bdna* and *Dsmc3d* have a medium Commit/Execution Ratio in SMP. As shown in Figure 13, MultiT&MV is faster than SingleT in *Bdna* and *Dsmc3d*. In the other applications, the Commit/Execution Ratio is either too high or too low and MultiT&MV tends to be only moderately faster than SingleT.

Overall, however, MultiT&MV is a good scheme: applications run on average 32% faster than in SingleT.

5.1.2 *Comparing MultiT&SV to the Other Schemes.* MultiT&SV should match MultiT&MV when mostly privatization patterns are rare. According to Table IV, *Track*, *Dsmc3d*; and *Euler* do not have such patterns. As shown in Figure 13, MultiT&SV largely matches MultiT&MV in these applications. This observation indirectly agrees with Steffan et al. [2000], who found no need to support multiple writers in their applications.

However, MultiT&SV should resemble SingleT when mostly privatization patterns dominate. This is because, as shown in Figure 9b, as tasks write to mostly privatized variables early in their execution, processors stall immediately as in SingleT. According to Table IV, such patterns are common in *P3m* and dominant in *Tree*, *Bdna*, and *Apsi*. As shown in Figure 13, MultiT&SV performs in between SingleT and MultiT&MV in *P3m*. For *Tree*, *Bdna*, and *Apsi*, however, MultiT&SV is even slower than SingleT.

The reason for this poor MultiT&SV performance is that our simple greedy assignment of tasks to processors at run time causes unfavorable task interactions in MultiT&SV. The result is additional stalls over SingleT.

To see why this happens, consider the following case that occurs in *Tree*. In this example, we consider six tasks running on three processors. The tasks are load imbalanced as shown in Figure 14a and have mostly privatization

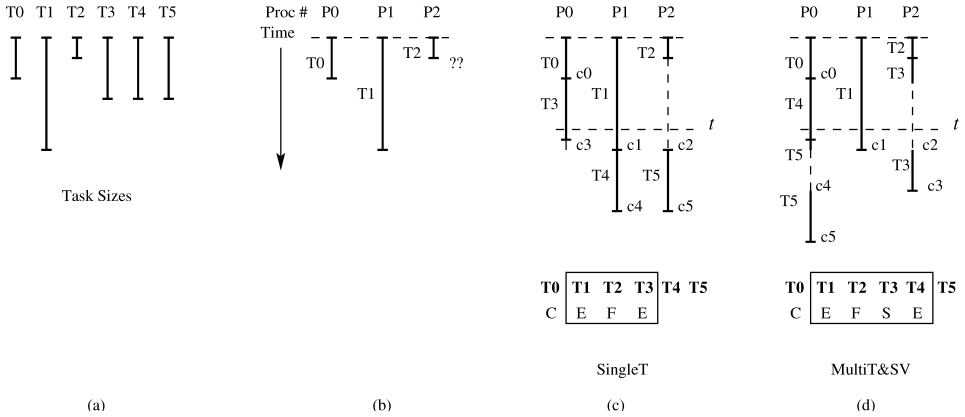


Fig. 14. Example in *Tree* where MultiT&SV is slower than SingleT. Charts (c) and (d) also show the status of the tasks at time t, where C, E, F, and S stand for committed, executing, finished, and stalled.

patterns. Suppose that tasks *T0*, *T1*, and *T2* have been assigned to processors *P0*, *P1*, and *P2*, respectively. Figure 14b shows the time when *P2* finishes *T2*. Under SingleT (Figure 14c), *P2* will wait until *T2* commits (point *c2*) and then grab the next available task (in our case, *T5*). Under MultiT&SV (Figure 14d), *P2* will greedily start executing *T3* and then stall as soon as *T3* attempts to generate a new version of a variable that has been accessed by *T2*. Recall that in MultiT&SV, a cache can only contain a single speculative version of any given variable. Unfortunately, with this scheduling, *P0* will later grab a task (*T4*) that is far from the nonspeculative task at that time (namely, *T1*). After *P0* finishes *T4*, it grabs *T5*. During the execution of *T5*, *P0* remains stalled for a long time, again because *T5* attempts to generate new versions of variables that have been accessed by *T4* (and are still speculative in the cache). The overall result is that MultiT&SV is slower than SingleT: the commit time of the last task (*c5*) in Figure 14d is later than in Figure 14c.

The reason why SingleT's round-robin assignment often suffers fewer stalls in this environment is that the tasks that are making progress are often those that are the closest to the task that is currently nonspeculative. For example, consider Figure 14c at time t, where *T1* is the nonspeculative task. We can see that *T1* is executing (*E*), *T2* is finished (*F*), and *T3* is executing in *P0*.

However, under MultiT&SV's greedy assignment, tasks far from the nonspeculative one may be making progress, while other tasks closer to the nonspeculative one are stalled. For example, consider Figure 14d at time t. We can see that the nonspeculative task *T1* is executing (*E*), *T2* is finished (*F*), *T3* is stalled (*S*), and *T4* is executing in *P0*. Stalling tasks closer to the nonspeculative one delays the commit wavefront, which is counterproductive for MultiT&SV schemes running applications with mostly privatization patterns: the commit wavefront is in the critical path because a stalled task cannot resume execution until the previous task that ran on the same processor commits.

Overall, we conclude that MultiT&SV is not an attractive scheme.

5.2 Merging of Task State with Main Memory

5.2.1 *Comparing Eager to Lazy AMM Schemes.* Laziness can speed up execution in the two cases where the commit wavefront appears in the critical path of an Eager scheme. These cases are shown in Figures 10c and 10a.

The first case (Figure 10c) occurs when processors stall during task execution, typically under SingleT and, if mostly privatization patterns dominate, under MultiT&SV. It can be shown that this situation occurs frequently in our applications: in all applications under SingleT, and in the privatization applications (*P3m*, *Tree*, *Bdna*, and *Apsi*) under MultiT&SV. In this case, the impact of laziness (Figure 10d) is roughly proportional to the application's Commit/Execution Ratio. From Table IV, we see that the ratio is significant for all applications except *P3m* and *Tree*. Consequently, in this first case, laziness should speed up SingleT for *Bdna*, *Apsi*, *Track*, *Dsmc3d*, and *Euler*, and MultiT&SV for *Bdna* and *Apsi*. Figure 13 confirms these expectations.

The second case where the wavefront is in the critical path (Figure 10a) occurs when processors do not stall during task execution, but the Commit/Execution Ratio times the number of processors is higher than 1. In this case, the wavefront appears at the end of the loop. This case could occur in all applications under MultiT&MV and in the nonprivatization ones (*Track*, *Dsmc3d*, and *Euler*) under MultiT&SV. However, according to Table IV, only *Apsi*, *Track*, and *Euler* have a Commit/Execution Ratio sufficiently high in SMP such that, when multiplied by 16, the result is over 1. Consequently, laziness (Figure 10b) should speed up MultiT&MV for *Apsi*, *Track*, and *Euler*, and MultiT&SV for *Track* and *Euler*. Figure 13 again confirms these expectations.[5]

Overall, Lazy AMM is effective. For the simpler schemes (SingleT and MultiT&SV), it reduces the average execution time by about 30%, while for MultiT&MV the reduction is 24%.

We have also evaluated a MultiT&MV system where the number of supported versions is limited by the size and associativity of the L2 cache: 512-Kbyte and 4, respectively, in our experiments. Figure 15 shows results for an Eager and a Lazy scheme. For each scheme there are two bars. The first one has an overflow area with support for as many versions as necessary (*Ovf*). This bar repeats the one in Figure 13. In the second one, there is no overflow area and, as a result, a task stalls in case of trying to displace a cache line with uncommitted data (*NoOvf*). The stalled task can only proceed when the task that produced the data that needs to be displaced becomes nonspeculative.

Figure 15 shows differences only for *P3m* and *Bdna*. In *P3m*, which is an imbalanced application, limiting the amount of speculative versions hurts performance in both *Eager* and *Lazy* schemes. In *Bdna*, *NoOvf* runs 16% slower

[5]Our conclusions on laziness agree with [Prvulovic et al. 2001] for 16 processors for the applications common to both papers: *Tree*, *Bdna*, and *Euler* (*Apsi* and *Track* cannot be compared because the problem sizes or the number of iterations per task are different). Our MultiT&MV Eager and Lazy schemes roughly correspond to their OptNoCT and Opt, respectively, without the support for high-level access patterns [Prvulovic et al. 2001].

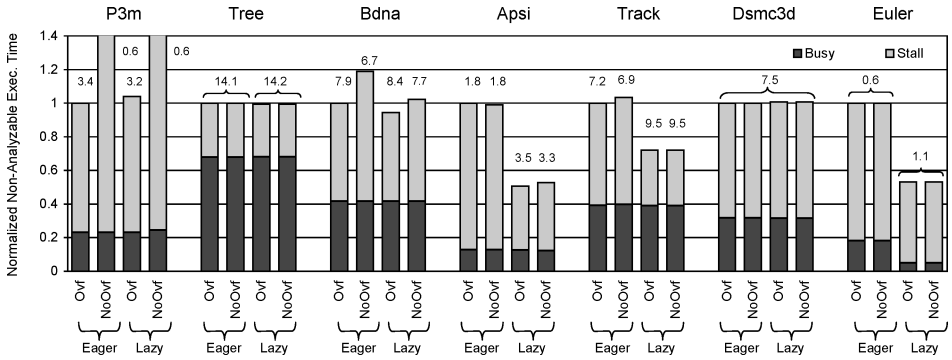


Fig. 15. Effect of limited support for Eager or Lazy merging with main memory state.

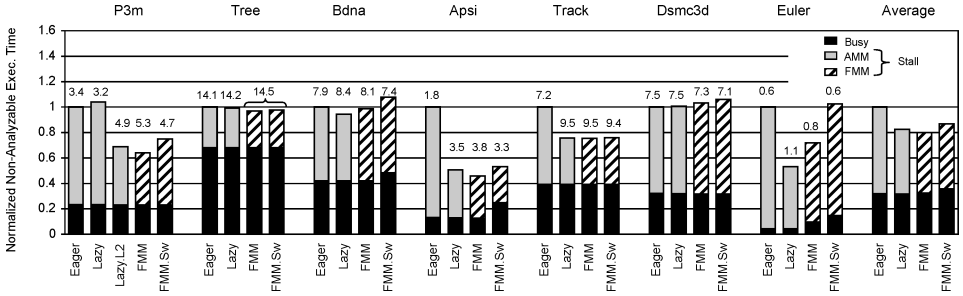


Fig. 16. Supporting an architectural main memory (AMM) or a future (FMM) one.

than *Ovf* with an Eager system. However, the difference between *NoOvf* and *Ovf* with a Lazy system is only 7%. The reason is that with a Lazy scheme, commit is faster. Laziness speeds-up the transfer of the commit token, reducing the probability of task stall. For the rest of our applications, since they are only slightly imbalanced and their working sets tend to largely fit in the cache, there are almost no differences between *Ovf* and *NoOvf*.

Thus, the conclusion of this experiment is that when the speculative state moderately overflows, laziness helps reduce the stall time. However, in case of high imbalance, where speculative versions would overflow, laziness has no effect, as the execution time is dominated by the task stall time.

5.2.2 *Comparing AMM to FMM Schemes.* Figure 16 compares the execution time of AMM schemes (Eager and Lazy) to the FMM scheme. All schemes are MultiT&MV. We also show the same FMM scheme except that the implementation and management of the MHB is done in software, with plain instructions added to the application [Garzarán et al. 2003] (FMM.Sw). This scheme eliminates the need for hardware support for the undo log. The appendix describes how the MHB is implemented and managed in software.

Section 3.3.4 argued that FMM schemes are better suited to version commit, while AMM schemes are better at version recovery. Figure 16 shows that there are few differences between Lazy AMM and FMM. The only significant ones occur in *P3m* and *Euler*. *P3m* has high load imbalance and mostly privatization

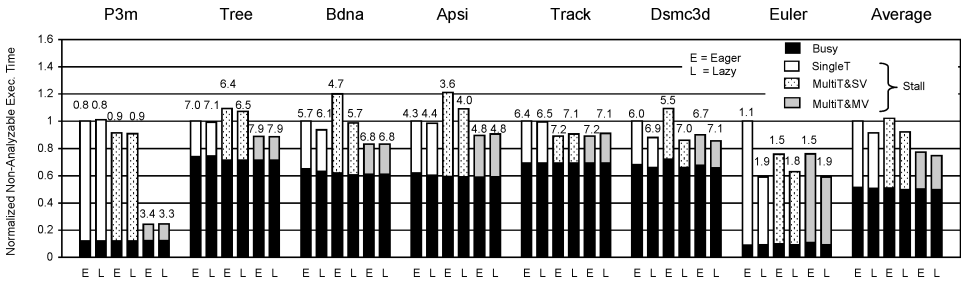


Fig. 17. Supporting architectural main memory (AMM) schemes in a CMP. In the figure, *E* and *L* stand for Eager and Lazy, respectively.

patterns. As a result, in Lazy (and Eager) AMM, the MROB in a processor may need to keep the state of numerous speculative tasks, with multiple versions of the same variable competing for the same cache set. Overflowing read-only, nonspeculative data is silently discarded, while overflowing speculative data is sent to the overflow area. Note that, unlike the versions in the MHB, the versions in the overflow area have to be accessed eventually. Overall, the resulting long-latency accesses to the overflow area or to memory to refetch data slow down *P3m*. To eliminate this problem, we have increased L2's size and associativity to 4 Mbytes and 16 ways, respectively (Lazy.L2 bar in *P3m*). In this case, AMM performs just as well as FMM.

In *Euler*, the Lazy AMM scheme performs better than the FMM scheme. The reason is that *Euler* has frequent squashes due to violations. Recall that AMM schemes recover faster than FMM schemes (Section 3.3.4).

We conclude that, in general, Lazy AMM and FMM schemes deliver a similar performance. However, Lazy AMM has an advantage in the presence of frequent squashes, while FMM has an advantage when task execution puts pressure on the size or associativity of the caches.

Finally, Figure 16 shows that the slowdown caused by updating the MHB in software (FMM.Sw) is modest. This agrees with Garzarán et al. [2003]. In Garzarán et al. [2003], we analyze in detail the effects of updating the MHB in software. On average, FMM.Sw takes 6% longer to run than FMM. FMM.Sw eliminates the need for the ULOG hardware in Table III, although it still needs the other FMM hardware in the table.

5.3 Evaluation of the Chip Multiprocessor (CMP)

Figure 17 repeats Figure 13 for the CMP architecture. Overall, we see that the trends are the same as in the SMP architecture. The most obvious change is that the relative differences between the different buffering schemes are smaller in the CMP than in the SMP architecture. This is not surprising, since buffering mainly affects memory system behavior. The CMP is less affected by the choice of buffering because its lower memory latencies, on average, result in less memory stall time. This observation is clear from the relatively higher Busy time in the CMP bars.

One observation in the CMP is that the improvement of Lazy over Eager schemes is smaller than before. There are two reasons for this. First, since the

number of processors is smaller, the commit serialization is less of a bottleneck. Second, the Commit/Execution Ratios are smaller (Table IV) because of the lower memory latencies. Overall, laziness reduces the average execution time by 9% in the simpler schemes (SingleT and MultiT&SV) and by only 3% in MultiT&MV. We also note that adding support for multiple task&versions still significantly improves speedups: when applied to SingleT Eager, it reduces the execution time by 23%, on average (compared to 32%, in SMP).

Finally, a comparison between Lazy AMM and FMM schemes for CMP is not shown because it is very similar to Figure 16. The Lazy AMM and FMM schemes perform similarly to each other.

5.4 Summary

Starting from the simplest scheme (SingleT Eager AMM), we have the choice of adding support for multiple tasks&versions (MultiT&MV) or for laziness. Our main conclusion is that supporting multiple tasks&versions is more complexity-effective than supporting laziness: the reduction in execution time is higher (32 versus 30% in our SMP; 23 versus 9% in our CMP), and Section 3.3.5 showed that the implementation complexity is lower for adding multiple tasks&versions. We also note that laziness is only modestly effective in tightly coupled architectures like our CMP.

A second conclusion is that the improvements due to multiple tasks&versions and due to laziness are fairly orthogonal in a large machine like our SMP. Indeed, adding laziness to the MultiT&MV Eager AMM scheme reduces the execution time by an additional 24% (Figure 13). In our CMP, however, the gains are only 3% (Figure 17).

A third conclusion is that the resulting system (MultiT&MV Lazy AMM) is competitive against what Table III indicated was the most complex system: MultiT&MV FMM. The Lazy AMM scheme is generally as fast as the FMM scheme (Figure 16). While Lazy AMM is not as tolerant of high capacity and conflict pressure on the buffers (*P3m* in Figure 16), it behaves better when squashes, because of dependence violations are frequent (*Euler* in Figure 16).

Finally, we show that MultiT&SV is not very attractive for applications like the ones we use, which often have mostly privatization patterns: it is as fast as SingleT (Figures 13 and 17), while it requires support beyond SingleT (Table III).

6. CONCLUSION

The contribution of this paper is threefold. First, it introduces a novel taxonomy of approaches to buffer multiversion memory state for TLS in multiprocessors. Second, it presents a detailed complexity-benefit tradeoff analysis of the approaches. Finally, it uses numerical applications to evaluate their performance under a single architectural framework.

Our analysis provides an upgrade path of features with decreasing complexity-effectiveness. Specifically, starting from the simplest scheme (SingleT Eager AMM), the most complexity-effective improvement is to add support for multiple tasks and versions per processor (MultiT&MV Eager AMM).

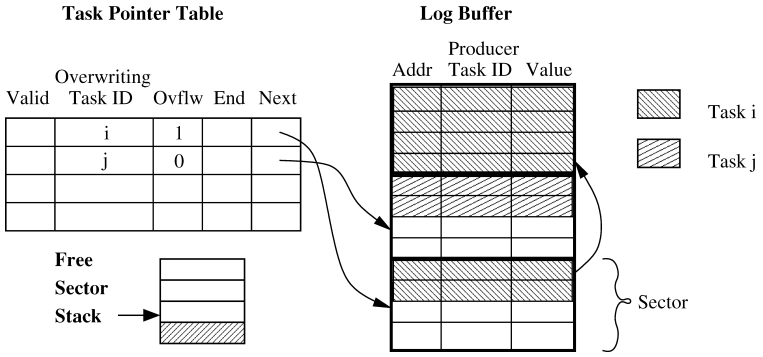


Fig. 18. Per-processor software structures that we use to implement the software MHB in FMM.Sw.

Performance can then be additionally improved in large machines by adding support for lazy merging of task state (MultiT&MV Lazy AMM). Finally, if the applications do not suffer frequent squashes, additional performance can be obtained by supporting future main memory (MultiT&MV FMM). This change adds complexity and only modest average performance benefits. If, instead, applications suffer frequent squashes, MultiT&MV Lazy AMM is faster. Overall, with our mix of applications, we find that MultiT&MV Lazy AMM and FMM have a similar performance.

APPENDIX: IMPLEMENTING AND MANAGING THE MHB IN SOFTWARE

The MHB is a logging system that must support four operations, namely, saving a new record in the log (*Insertion*), finding a record in the log (*Retrieval*), unwinding the log to undo tasks (*Recovery*), and freeing up log records after their information is not needed for retrieval or recovery (*Recycle*).

Figure 18 shows the per-processor software structures that we use to implement the software MHB in FMM.Sw. A Log Buffer is broken down into fixed-sized sectors that are used to log individual tasks. The compiler sets the size of the sectors and Log Buffer based on the estimated number of writes per task and the estimated number of uncommitted tasks per processor, respectively.

When a task starts running, it is dynamically assigned an entry in the Task Pointer Table and one sector in the Log Buffer. Free sectors are obtained from the Free Sector Stack. Two pointers in the Task Pointer Table point to the Next entry to fill and the End entry to check for overflow. If the task needs more entries than a sector, we dynamically assign another sector and link it to the previous one, while we set the Overflow bit and update the End pointer (Figure 18). If the Free Sector Stack runs out of entries, we resize the Log Buffer and Stack accordingly.

With these software structures, the following four operations are supported:

- **Insertion.** At compile time, the compiler instruments stores in the code with instructions to save a log record. As shown in Figure 18, a record includes

the following information about the variable that is about to be updated: its virtual address (the only one the software knows), the ID of its producer task, and the value before the update. After the record is inserted at run time, the Next pointer is incremented. At the end of a task, all the records that the task generated are in contiguous locations in one or more sectors—easily retrievable through the Task Pointer Table with the ID of that task.

- **Recycle.** When a task commits, its entry in the Task Pointer Table becomes useless: its updates will never have to be undone. Consequently, a processor regularly runs the log-recycle algorithm. It involves identifying the entries in the Task Pointer Table that correspond to committed tasks. These entries are invalidated, and their sectors in the Log Buffer are recycled by returning them to the Free Sector Stack.
- **Recovery and Retrieval.** Recovery occurs when we need to repair the state after the detection of a data dependence violation due to an out-of-order RAW across tasks. Retrieval occurs in an in order RAW dependence across tasks that requires log access. The access is required when a new task running on the producer processor has overwritten a variable that is requested by the consumer processor, pushing the desired version of the variable into the log. These two cases happen infrequently for our applications and, therefore, are not performance critical. We solve them with software exception handlers that access the logs.

The compiler should instrument the application so that, at run time, the log is managed fully in software. Of all the operations described, insertion is the only one that is truly overhead-sensitive. This is because it is performed very frequently. The other operations occur much less frequently and can be handled by less efficient routines. Inserting a record in the local log involves collecting the items to save, saving them in sequence using the *Next* pointer, and advancing the pointer (Figure 18). Notice that a record should be inserted for both speculative and nonspeculative stores. A store is considered speculative if it may access data whose access pattern cannot be fully analyzed by the compiler; speculative stores should trigger the TLS protocol.

To reduce code instrumentation, the log only saves the value overwritten by the *first store* to the variable in the task. Identifying first stores for variables accessed with nonspeculative accesses should be easy, since their dependence structure is analyzable. To identify first stores for speculative accesses, the ID of the writing task is compared with the producer task ID of the variable. If they are the same, this is not a first store and logging is skipped. Otherwise, a new record is inserted in the log. It can be shown [Garzarán et al. 2003] that, for speculative stores, inserting an entry in the log involves executing a total of 10 additional instructions if the store is a first store, while only 2 instructions if the store is not a first store.

ACKNOWLEDGMENTS

We thank the members of the I-ACOMA group for their valuable feedback. We also thank the referees for their comments.

REFERENCES

AKKARY, H. AND DRISCOLL, M. A. 1998. A dynamic multithreading processor. In *International Symposium on Microarchitecture*. 226–236.

BARNES, J. E. 1994. ftp://hubble.ifa.hawaii.edu/pub/barnes/treecode/. *University of Hawaii*.

BLUME, W., DOALLO, R., EIGENMANN, R., GROUT, J., HOEFLINGER, J., LAWRENCE, T., LEE, J., PADUA, D., PAEK, Y., POTTENGER, B., RAUCHWERGER, L., AND TU, P. 1996. Advanced program restructuring for high-performance computers with polaris. *IEEE Computer 29*, 12 (December), 78–82.

CINTRA, M., MARTÍNEZ, J. F., AND TORRELLAS, J. 2000. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. 13–24.

EIGENMANN, R., HOEFLINGER, J., AND PADUA, D. 1998. On the automatic parallelization of the perfect benchmarks. In *IEEE Transactions on Parallel and Distributed Systems 9*, 5–23.

FIGUEIREDO, R. AND FORTES, J. 2001. Hardware support for extracting coarse-grain speculative parallelism in distributed shared-memory multiprocesors. In *Proceedings of the International Conference on Parallel Processing*.

FRANK, M., LEE, W., AND AMARASINGHE, S. 2001. A Software Framework for Supporting General Purpose Applications on Raw Computation Fabrics. Tech. rep., MIT/LCS Technical Memo MIT-LCS-TM-619. July.

FRANKLIN, M. AND SOHI, G. S. 1996. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers 45*, 5 (May), 552–571.

GARZARÁN, M. J., PRVULOVIC, M., LLABERÍA, J. M., VIÑALS, V., RAUCHWERGER, L., AND TORRELLAS, J. 2003. Using software logging to support multi-version buffering in thread-level speculation. In *Proceeding of the International Conference on Parallel Architectures and Compilation Techniques*. 170–181.

GOPAL, S., VIJAYKUMAR, T. N., SMITH, J. E., AND SOHI, G. S. 1998. Speculative versioning cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*. 195–205.

GUPTA, M. AND NIM, R. 1998. Techniques for speculative run-time parallelization of loops. In *Proceedings of Supercomputing 1998*.

HAMMOND, L., WILLEY, M., AND OLUKOTUN, K. 1998. Data speculation support for a chip multiprocessor. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*. 58–69.

KNIGHT, T. 1986. An architecture for mostly functional languages. In *ACM Lisp and Functional Programming Conference*. 500–519.

KRISHNAN, V. AND TORRELLAS, J. 1999. A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. on Computers*, 866–880.

MARCUELLO, P. AND GONZÁLEZ, A. 1999. Clustered speculative multithreaded processors. In *Proceedings of the 1999 International Conference on Supercomputing*. 365–372.

PRVULOVIC, M., GARZARÁN, M. J., RAUCHWERGER, L., AND TORRELLAS, J. 2001. Removing architectural bottlenecks to the scalability of speculative parallelization. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*. 204–215.

RAUCHWERGER, L. AND PADUA, D. 1995. The LRPD Test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the SIGPLAN 1995 Conference on Programming Language Design and Implementation*. 218–232.

RUNDBERG, P. AND STENSTRÖM, P. 2000. Low-cost thread-level data dependence speculation on multiprocessors. In *Fourth Workshop on Multithreaded Execution, Architecture and Compilation*.

SMITH, J. E. AND PLESZKUN, A. R. 1988. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers C-37*, 5 (May), 562–573.

SOHI, G. S., BREACH, S., AND VIJAYKUMAR, S. 1995. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. 414–425.

STEFFAN, J., COLOHAN, C. B., AND MOWRY, T. C. 1997. Architectural Support for Thread-Level Data Speculation. Tech. rep., CMU-CS-97-188, Carnegie Mellon University. November.

STEFFAN, J., COLOHAN, C., ZHAI, A., AND MOWRY, T. C. 2000. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. 1–12.

TREMBLAY, M. 1999. MAJC: Microprocessor Architecture for Java Computing. Hot Chips.

TSAI, J. Y., HUANG, J., AMLO, C., LILJA, D., AND YEW, P. C. 1999. The superthreaded processor architecture. *IEEE Trans. on Computers 48,* 9 (Sept.), 881–902.

ZHANG, Y. 1999. Hardware for Speculative Run-Time Parallelization in DSM Multiprocessors. Ph.D. Thesis, Dept. of Elec. and Comp. Engin., Univ. of Illinois at Urbana-Champaign.

ZHANG, Y., RAUCHWERGER, L., AND TORRELLAS, J. 1999. Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*. 135–139.

Received February 2005; revised July 2005; accepted July 2005