Appears in the Proceedings of the
Ninth International Symposium on High Performance Computer Architecture (HPCA-9),
February 2003.

Tradeoffs in Buffering Memory State for Thread-Level Speculation in Multiprocessors [*]

María Jesús Garzarán, Milos Prvulovic[†], José María Llabería[‡], Víctor Viñals, Lawrence Rauchwerger[§], and Josep Torrellas[†]

Universidad de Zaragoza, Spain
{garzaran,victor}@posta.unizar.es

[†]University of Illinois at Urbana-Champaign
{prvulovi, torrellas}@cs.uiuc.edu

[‡]Universitat Politècnica de Catalunya, Spain
llaberia@ac.upc.es

[§]Texas A&M University
rwerger@cs.tamu.edu

Abstract

Thread-level speculation 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 state may contain multiple versions of the same variable.

In this paper, we introduce a novel taxonomy of approaches to buffer and manage multi-version 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 for merging the state of tasks with main memory lazily. Moreover, both supports can be gainfully combined and, in large machines, their effect is nearly fully additive. Finally, the more complex support for future state in main memory can boost performance when buffers are under pressure, but hurts performance when squashes are frequent.

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. 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. [1, 4, 6, 8, 10, 12, 13, 14, 15, 16, 20, 21, 23, 24, 26]) to software-based (e.g. [7, 11, 17, 18]), and targeting small machines (e.g. [1, 8, 10, 12, 14, 15, 20, 23, 24]) or large ones (e.g. [4, 6, 11, 16, 17, 18, 21, 26]).

Each scheme for thread-level speculation has to solve two major problems: detection of violations and, 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 order 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, and is discarded when a violation is detected. In some programs, different speculative tasks running concurrently may buffer different versions of the same variable. Moreover, a processor executing multiple speculative tasks in sequence may end up buffering speculative state from multiple tasks, and maybe even multiple versions of the same variable. In all cases, the speculative state must be organized such that reader tasks receive the correct versions, and versions eventually merge into the safe state in order. 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 [4, 6, 10, 14, 21], write buffers [12, 24] or special buffers [8, 16] 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 [7, 9, 25, 26]. 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.

One contribution of this paper is to provide such a systematic study. We introduce a novel taxonomy of approaches to buffer and manage multi-version *speculative memory state* in multiprocessors. In addition, we 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

[*]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 TIC 2001-0995-C02-02; and by gifts from IBM and Intel.

single architectural framework. In the evaluation, we examine both chip and scalable multiprocessors.

Our results show that buffering the state of multiple speculative tasks and versions per processor is more complexity-effective than merging the state of tasks with main memory lazily. Moreover, both supports can be gainfully combined and, in large machines, their effect is nearly fully additive. Finally, the more complex support for 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 tradeoff analysis; Section 4 describes our evaluation methodology; Section 5 evaluates the different buffering approaches; and Section 6 concludes.

2 Buffering Memory State

As tasks execute under thread-level speculation, they have a certain relative order given by sequential execution semantics. In the simplest case, if we give increasing IDs to successor tasks, the lowest-ID running task is *non-speculative*, while its successors are *speculative*, and its predecessors are *committed*. In general, the state generated by a task must be managed and buffered differently depending on whether the task is speculative, non-speculative or committed. In this section, we list the major challenges in memory state buffering in this environment and present supporting data.

2.1 Challenges in Buffering State

Separation of Task State. Since a speculative task may be squashed, its state is unsafe. Therefore, its state is typically kept separate from that of other tasks and main memory by buffering it in caches [4, 6, 10, 14, 21] or special buffers [8, 12, 16, 24]. Alternatively, the task state is merged with memory, but the memory overwritten in the process is saved in an undo log [7, 9, 25, 26].

Multiple Versions of the Same Variable in the System. A task has at most a single version of any given variable. However, different speculative tasks that run concurrently may produce different versions of the same variable. These versions must be buffered separately and provided to their consumers.

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 a processor can only hold state for a single speculative task, the processor stalls until the task commits. In more advanced designs, the local buffer can buffer several speculative tasks, enabling the processor to execute another task.

Multiple Versions of the Same Variable in a Single Processor. If multiple speculative tasks can be buffered for each processor, a local buffer may need to hold multiple versions of the same variable. On external request, the buffer must provide the correct version.

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

2.2 Application Behavior

To gain insight into these challenges, Figure 1-(a) shows some application characteristics. The applications (discussed in Section 4.2) execute speculatively parallelized loops on a simulated 16-processor scalable machine (discussed in Section 4.1). Columns 2 and 3 show the average number of speculative tasks that co-exist 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 two speculative tasks at a time.

Appl.	Average # Spec Tasks		Average Written Footprint per Spec Task	
	In Sytem	Per Proc	Total (KB)	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

(a)

```
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
```

(b)

Figure 1. Application characteristics that illustrate the challenges of buffering.

Columns 4 and 5 show the size of the written footprint of a speculative task and the percent of it that results from mostly-privatization access patterns, respectively. The written footprint is an indicator of the buffer size needed per task. Mostly-privatization patterns are those that end up being private most (but not all) of the time, but that the compiler cannot prove as private. These patterns result in many tasks creating a new version of the same variable. As an example of code with such patterns, Figure 1-(b) 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. Figure 1-(a) shows that this pattern is present in some applications, increasing the complexity of buffering.

3 Taxonomy and Tradeoff Analysis

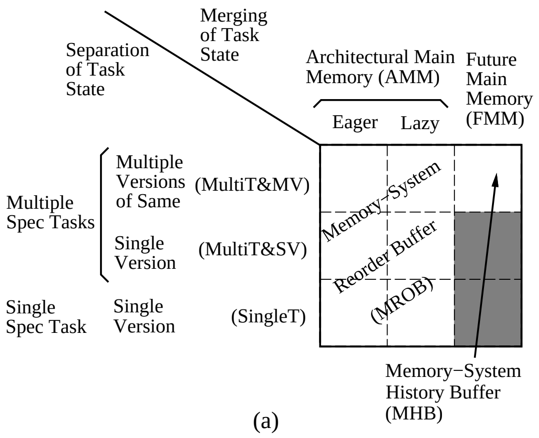
To understand the tradeoffs in buffering speculative memory state under thread-level speculation, we present a novel taxonomy of possible approaches (Section 3.1), map existing schemes to it (Section 3.2), and perform a tradeoff analysis of benefits and complexity (Section 3.3).

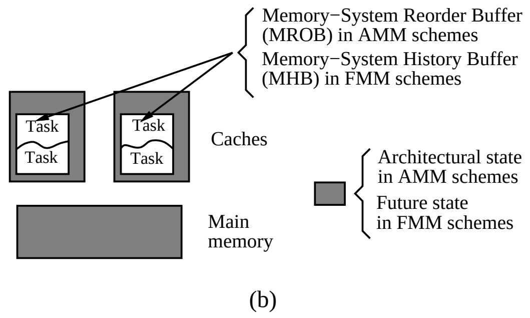
3.1 Novel Taxonomy of Approaches

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 2-(a).

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





Memory−System Reorder Buffer (MROB) in AMM schemes
Memory−System History Buffer (MHB) in FMM schemes
Task
Task
Task
Task
Caches
Architectural state in AMM schemes
Future state in FMM schemes
Main memory
(b)

Figure 2. Buffering and managing speculative memory state: taxonomy (a) and difference between architectural (AMM) and future main memory (FMM) schemes (b).

a single speculative task at a time (*SingleT*); multiple speculative tasks but only a single version of given any 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 non-speculative. In MultiT&MV schemes, each local speculative task can keep its own speculative version of the same variable.

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.

[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 fi rst round, while in *MultiT* schemes, it is "virtual" or flexible.

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 [19].

The architectural file in [19] refers to the safe contents of the register file. The architectural file is updated with the result of an 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 3-(a) 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 [19], the result of an instruction updates the architectural file at commit time. A straightforward approach in thread-level speculation 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-backs of dirty lines to memory [4] or ownership requests for these lines to obtain coherence with main memory [21].

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 co-exist in different caches and main memory. However, it is clear at any time which one is the latest one [10, 16]. This approach we call *Lazy AMM*.

Consider now the future file in [19]. 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 to the architectural file.

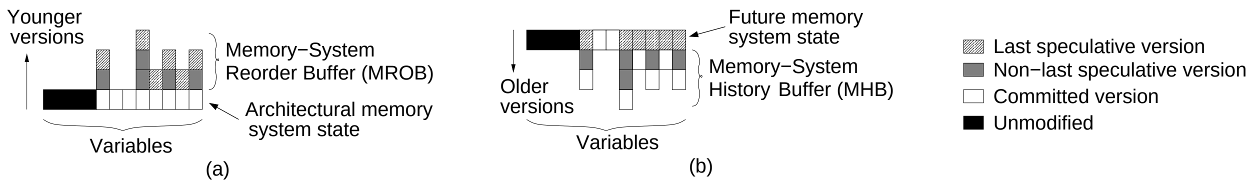

Younger versions
Memory−System Reorder Buffer (MROB)
Architectural memory system state
Variables
(a)
Future memory system state
Memory−System History Buffer (MHB)
Older versions
Variables
(b)
▨ Last speculative version
■ Non−last speculative version
□ Committed version
■ Unmodified

Figure 3. Snapshot of the memory system state of a program under thread-level speculation using the concepts of Memory-System Reorder Buffer (a) and Memory-System History Buffer (b).

		Architectural Main Memory (AMM)		Future Main Memory (FMM)
		Eager	Lazy	
Multiple Spec Tasks (MultiT)	Multiple Versions of Same (MultiT&MV)	Hydra Steffan97&00 Cintra00	Prvulovic01	Zhang99&T Garzaran01
	Single Version (MultiT&SV)	Steffan97&00		■
Single Spec Task (SingleT)	Single Version	Multiscalar (with hierarchical ARB) Superthreaded MDT Marcuello99 DDSM	Multiscalar (with SVC)	COARSE RECOVERY: LRPD, SUDS,...

Figure 4. Mapping schemes for thread-level speculation in multiprocessors onto our taxonomy.

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 3-(b) 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 2-(b)). 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 2-(a) 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 4 maps existing schemes for thread-level speculation in multiprocessors onto our taxonomy[2]. Consider first SingleT Eager AMM schemes. They include Multiscalar with hierarchical ARB [8], Superthreaded [24], MDT [14], and Marcuello99 [15]. 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 [10] is SingleT Lazy AMM because a processor cache contains *speculative* state from at most a single task (SingleT), while committed versions linger in the cache after the owner task commits (Lazy AMM). In DDSM [6], 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. For this reason, the distinction between Eager and Lazy does not apply.

MultiT&MV AMM schemes include Hydra [12], Steffan97&00 [21, 22], Cintra00 [4], and Prvulovic01 [16]. 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 non-speculative[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 [21, 22] 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.

[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.

[3]While this statement is true for Hydra in concept, the evaluation in [12] assumes only as many buffers as processors, making the system SingleT.

MultiT&MV FMM schemes include Zhang99&T [25, 26] and Garzaran01 [9]. 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 MHB in Zhang99&T is kept in hardware structures called logs. In Garzaran01, 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 4 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 [17], SUDS [7], and other proposals [11, 18].

3.3 Tradeoff Analysis of Benefits and Complexity

To explore the design space of Figure 2-(a), we start with the simplest scheme (SingleT Eager AMM) and progressively complicate it. For each step, we consider performance benefits and support required. Tables 1 and 2 summarize the analysis.

Support	Description
Cache Task ID (CTID)	Storage and checking logic for a task-ID fi eld 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 1. Different supports required.

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 non-speculative 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 & Versions per Processor

Benefits: Tolerate Load Imbalance and Mostly-Privatization Patterns

SingleT schemes may perform poorly if tasks have 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 5, where T_i and c_i mean execution and commit, respectively, of task i. Figure 5-(a) shows a SingleT scheme: processor *1* completes task *T1* and waits; when it receives the commit token, it commits *T1* and starts *T3*.

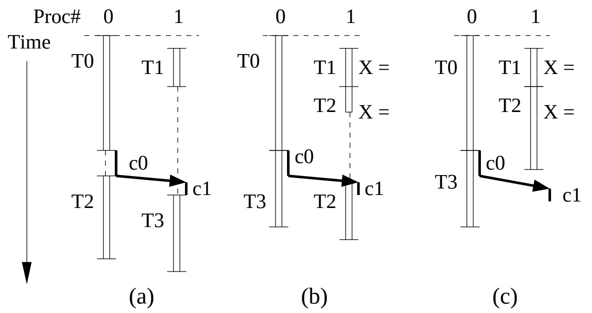


Figure 5. Example of four tasks executing under SingleT (a), MultiT&SV (b), and MultiT&MV (c).

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 run slowly if tasks have *both* load imbalance and create multiple versions per variable. The latter occurs, for example, under mostly-privatization patterns (Section 2.2). 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 non-speculative and, as a result, the first version can merge with memory, the processor resumes.

As an example, Figure 5-(b) shows that processor *1* generates a version of *X* in *T1* and stalls when it is about to generate a second one 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 5-(c). The result is faster execution.

Supports: Cache Task ID and Cache Retrieval Logic

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) 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 1.

An 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 tells 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 [4, 22]. In this case, an 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 serial comparisons that may increase cache occupancy, or more hardware for parallel comparisons. Furthermore, responses may require combining different words from

Upgrade	Performance Benefi t	Additional Support Required
SingleT ⟶ MultiT&SV	Tolerate load imbalance without mostly-privatization access patterns	CTID
MultiT&SV ⟶ MultiT&MV	Tolerate load imbalance even with mostly-privatization access patterns	CRL
Eager AMM ⟶ Lazy AMM	Remove commit wavefront from critical path	CTID and (VCL or MTID)
Lazy AMM ⟶ FMM	Faster version commit but slower version recovery	ULOG and (MTID if Lazy AMM had VCL)

Table 2. Benefits obtained and support required for each of the different mechanisms.

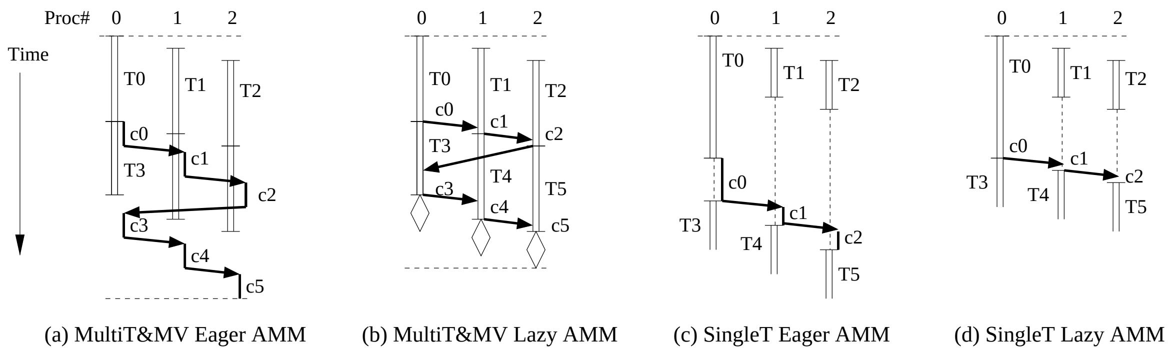


Figure 6. Progress of the execution and commit wavefronts under different schemes.

the multiple cached versions of the requested line. This support we call Cache Retrieval Logic (CRL) in Table 1.

3.3.3 Lazy Merging with Architectural Main Memory (AMM)

Benefits: Remove Commit Wavefront from Critical Path

Program execution under thread-level speculation 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 6-(a) 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 back to memory the data it wrote [4] or get ownership for it [21]. 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 6-(a)). To understand this case, we call *Commit/Execution Ratio* the ratio between the average duration of a task commit and a task execution. 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 6-(a)).

The second case occurs when the commit wavefront delays the restart of processors stalled due to the load-balancing limitations of MultiT&SV or SingleT (Figure 6-(c)). 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 merged with main memory lazily, 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 6-(b) and (d) correspond to Figures 6-(a) and (c), respectively, under Lazy AMM. In Figure 6-(b), 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 [16]. This is shown with diamonds in the figure. In Figure 6-(d), the commit wavefront affects the critical path minimally. In both cases, the program runs faster.

Supports: Cache Task ID and Version Combining Logic (or Memory Task ID)

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 co-exist in the machine.

These challenges are addressed with two supports: logic to combine versions (Version Combining Logic or VCL in Table 1) and logic for ordering the versions of a variable. Different implementations of these two supports are proposed by Prvulovic01 [16] and Multiscalar with SVC [10].

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 [10, 16]. 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 multi-word cache lines, on displacements and requests, the VCL has to collect committed versions for *all* the words in the line from the caches and combine them [16].

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

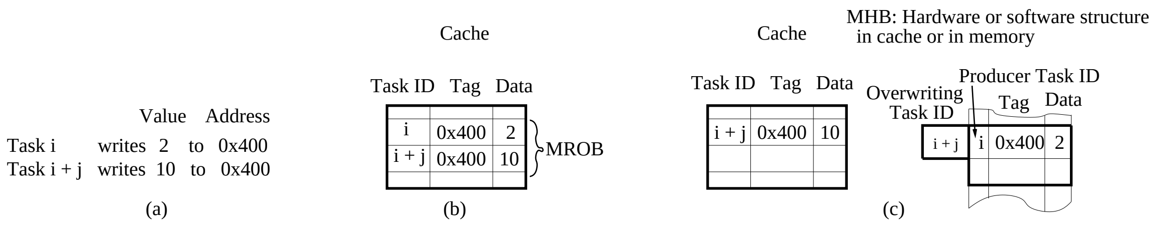


Figure 7. Implementing the MROB and the MHB.

Prvulovic01 [16] and was called CTID in Table 1. An alternative approach used by Multiscalar with SVC [10] 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 2.

We note that the version-combining support provided by VCL can instead be provided by a scheme proposed in Zhang99&T [25]. 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 1.

3.3.4 Future Main Memory (FMM)

Benefits: Faster Version Commit but Slower Recovery

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 [16]. 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 one 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 simply commits. The older version can *also be safely displaced* from the cache and written back to memory at any time. Hopefully, it is never accessed again.

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 copying all the versions overwritten by the offending task and successors from the MHB to main memory, in strict reverse task order.

Supports: Cache Task ID, Memory Task ID, and Undo Log

Consider an example of a program where each task generates its own private version of variable *X*. Figure 7-(a) shows the code for two tasks that run on the same processor. If we use an AMM scheme, Figure 7-(b) shows the processor's cache and local MROB, assuming MultiT&MV support.

If we use an FMM scheme, Figure 7-(c) 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 (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. 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 7-(c)). 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 7-(c)).

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 rare retrieval operations [9]. Both hardware [25, 26] and software [9] logs have been proposed. This support is called Undo Log (ULOG) in Table 1.

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 1. 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 2-(a) 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 main memory has to always keep the latest future state possible. To avoid updating main memory out of

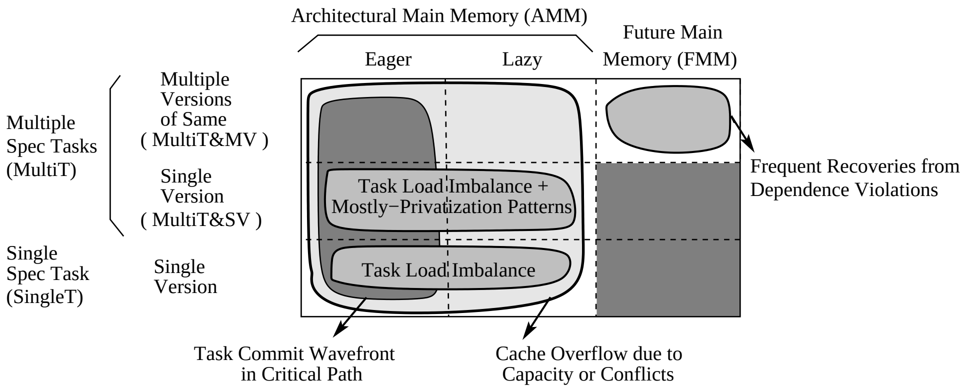

Architectural Main Memory (AMM)
Eager
Lazy
Future Main Memory (FMM)
Multiple Spec Tasks (MultiT)
Multiple Versions of Same (MultiT&MV)
Single Version (MultiT&SV)
Single Spec Task (SingleT)
Single Version
Task Load Imbalance + Mostly−Privatization Patterns
Task Load Imbalance
Frequent Recoveries from Dependence Violations
Task Commit Wavefront in Critical Path
Cache Overflow due to Capacity or Conflicts

Figure 8. Application characteristics that limit performance in each scheme.

task-ID order, FMM schemes [9, 25, 26] use the Memory Task ID (MTID) support of Section 3.3.3. 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 2 can be used to qualitatively compare the implementation complexity of the different supports. We start with SingleT Eager AMM and progressively add features.

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). Indeed, 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.

Complexity considerations should be assessed against performance gains. From our discussion, Figure 8 lists application characteristics that we expect to limit performance in each scheme.

4 Evaluation Methodology

4.1 Simulation Environment

We use execution-driven simulations to model two architectures: a scalable CC-NUMA and a chip multiprocessor (CMP). Both systems use 4-issue dynamic superscalars with a 64-entry instruction window, 4 Int, 2 FP, and 2 Ld/St units, up to 8 pending loads and 16 stores, a 2K-entry BTB with 2-bit counters, and an 8-cycle branch penalty.

The CC-NUMA has 16 nodes of 1 processor each. Each node has a 2-way 32-Kbyte D-L1 and a 4-way 512-Kbyte L2, both write-back with 64-byte lines. The nodes are connected with a 2D mesh. The minimum round-trip latencies from a processor to the L1, L2, memory in the local node, and memory in a remote node that is 2 or 3 protocol hops away are 2, 12, 75, 208 and 291 cycles, respectively.

The CMP models a more technologically advanced system. It has 8 processors. Each one has a 2-way 32-Kbyte D-L1 and a 4-way 256-Kbyte L2. All caches are write-back with 64-byte lines. The L2s connect through a crossbar to 8 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 4 ways and holds 16 Mbytes. The minimum round-trip latencies from a processor to the L1, L2, another processor's L2, L3, and main memory are 2, 8, 18, 38, and 102 cycles, respectively. In both machines, contention is accurately modeled in the whole system.

We model all the non-shaded buffering approaches in our taxonomy of Figure 2-(a). To model the approaches, we use a speculative parallelization protocol similar to [16], but without its support for High-Level Access Patterns. The protocol is appropriately modified to adapt to each box in Figure 2-(a). 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 [16]. It needs a single task-ID tag per cache line. We avoid processor stalls in AMM due to L2 conflict or capacity limitations by using a per-processor overflow memory area similar to [16]. 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.* [21]. 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].

[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.* [21], who used non-numerical, fine-grained ap-

Appl	Non-Analyzable Sections (Loops)	% of Tseq	# Invoc; # Tasks per Invoc	# Instr per Task (Thousand)	Commit/Exec Ratio (%)		Appl Characteristics		
					NUMA	CMP	Load Imbal	Priv Pattern	Comm/Exec Ratio
P3m	pp_do100	56.5	1;97336	69.1	0.3	0.1	High	Med	Low
Tree	accel_do10	92.2	41;4096	28.7	1.4	0.4	Med	High	Low
Bdna	actfor_do240	44.2	1;1499	103.3	6.0	3.9	Low	High	Med
Apsi	run_do[20,30, 40,50,60,100]	29.3	900;63	102.6	11.4	6.1	Low	High	High-Med
Track	nlfilt do300	58.1	56;126	22.3	8.4	2.0	Med	Low	High-Med
Dsmc3d	move3_goto100	41.2	80;46777	5.4	6.2	4.4	Low	Low	Med
Euler	dflux_do[100,200], psmoo_do20, eflux_do[100,200,300]	89.8	120;1871	3.9	12.6	14.5	Low	Low	High
Average		58.8	171;21681	47.9	6.6	4.5			

Table 3. Application characteristics. 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 512x1x64. In P3m, while the loop has 97,336 iterations, we only use the first 9,000 iterations in the evaluation. In Euler, since all 6 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.

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. 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.

4.2 Applications

For the evaluation, we use a set of numerical applications. In each application, we use the Polaris parallelizing compiler [3] 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 [2]. 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 [5], 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 application domain used, we will see that our applications cover a very wide range of buffering behaviors.

Table 3 shows the non-analyzable sections in each application. These sections are loops, and the speculative tasks are chunks of consecutive iterations. The chunks are dynamically scheduled. The table lists the 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 Sun Ultra 5 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 CC-NUMA and CMP architectures.

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, while Euler's execution is substantially affected by squashes (0.02 squashes per committed task), the opposite is true for P3m, Tree, Bdna, and Apsi. Track and Dsmc3d are in between.

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

5 Evaluation

We first focus on the CC-NUMA system, and then evaluate the CMP in Section 5.3.

5.1 Separation of Task State under Eager AMM

Figure 9 compares the execution time of the non-analyzable 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 instruction execution plus non-memory pipeline hazards (Busy), and stalls due to memory access, not enough task/version support, and end-of-loop stall due to commit wavefront or load imbalance (Stall). The numbers on top of the bars are the speedups over sequential execution of the code where all data is in the local memory module. In this section, we examine the Eager schemes, which are the bars in odd positions; we leave the Lazy ones for Section 5.2.

Consider MultiT&MV first. It should perform better than SingleT in two cases. One is in highly load-imbalanced applications (Figure 5-(c) vs 5-(a)). According to Table 3, only P3m has high imbalance. As shown in Figure 9, MultiT&MV is faster than SingleT in P3m.

plications. Indeed, for numerical applications, [16] shows that the number of non-owned modified lines per speculative task is about 200 on average.

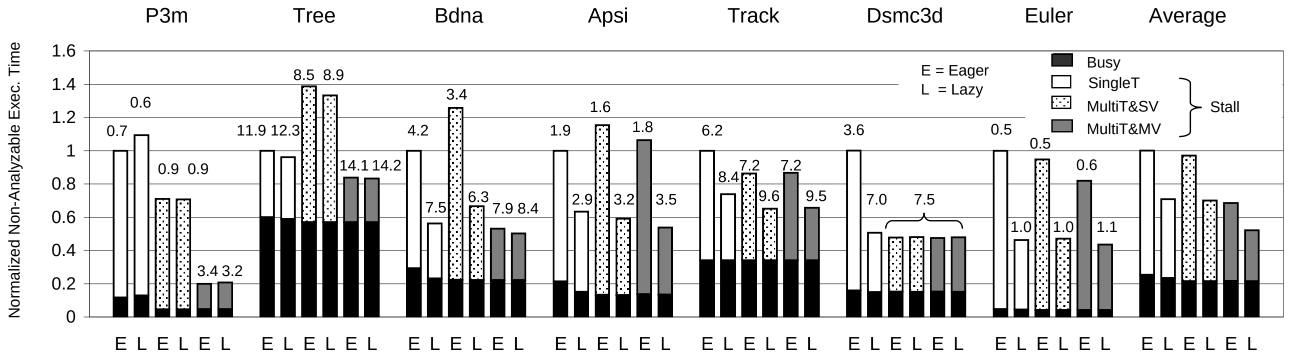


Figure 9. Supporting single or multiple speculative tasks or versions per processor, for eager or lazy architectural main memory (AMM) schemes. In the figure, *E* and *L* stand for Eager and Lazy, respectively.

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 6-(c)). 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 6-(a)). According to Table 3, Bdna and Dsmc3d have a medium Commit/Execution Ratio in NUMA. As shown in Figure 9, 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 for MultiT&MV to be much faster than SingleT.

Consider now MultiT&SV. It should match MultiT&MV when mostly-privatization patterns are rare. According to Table 3, Track, Dsmc3d and Euler do not have such patterns. As shown in Figure 9, MultiT&SV largely matches MultiT&MV in these applications. This observation indirectly agrees with Steffan *et al.* [21], who found no need to support multiple writers in their applications.

However, MultiT&SV should resemble SingleT when mostly-privatization patterns dominate. According to Table 3, such patterns are common in P3m and dominant in Tree, Bdna, and Apsi. As shown in Figure 9, MultiT&SV takes in between SingleT and MultiT&MV in P3m. For Tree, Bdna, and Apsi, tasks write to mostly-privatized variables early in their execution. As a result, a processor stalls immediately (Figure 5-(b)) as in SingleT. In practice, it can be shown that our greedy dynamic assignment of tasks to processors causes unfavorable task interactions in MultiT&SV that result in additional stalls. This is why MultiT&SV is even slower than SingleT in Tree, Bdna, and Apsi.

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

5.2 Merging of Task State with Main Memory

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 6-(c) and 6-(a).

The first case (Figure 6-(c)) 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 6-(d)) is roughly proportional to the application's Commit/Execution Ratio. From Table 3, 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 9 confirms these expectations.

The second case where the wavefront is in the critical path (Figure 6-(a)) 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 non-privatization ones (Track, Dsmc3d, and Euler) under MultiT&SV. However, according to Table 3, only Apsi, Track, and Euler have a Commit/Execution Ratio sufficiently high in NUMA such that, when multiplied by 16, the result is over 1. Consequently, laziness (Figure 6-(b)) should speed up MultiT&MV for Apsi, Track, and Euler, and MultiT&SV for Track and Euler. Figure 9 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%.

Comparing AMM to FMM Schemes

Figure 10 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 copying of overwritten versions to the MHB is done in software, with plain instructions added to the application [9] (FMM.Sw). This scheme eliminates the need for hardware support for the undo log.

Section 3.3.4 argued that FMM schemes are better suited to version commit, while AMM schemes are better at version recovery. Figure 10 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 patterns. As a result, in Lazy (and Eager) AMM, the MROB in a pro-

[5]Our conclusions on laziness agree with [16] 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 [16].

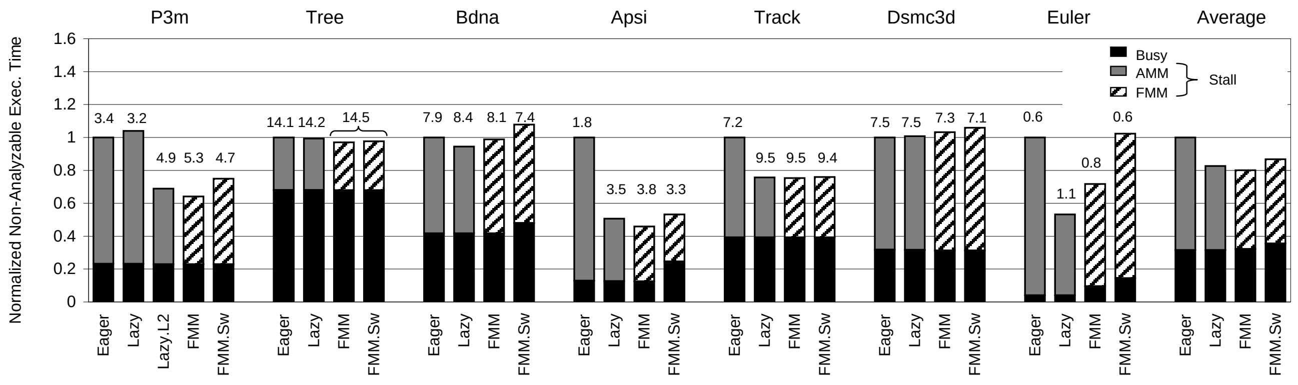


Figure 10. Supporting an architectural main memory (AMM) or a future (FMM) one.

cessor may keep the state of numerous speculative tasks, with multiple versions of the same variable competing for the same cache set. Overflowing read-only, non-speculative 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 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 10 shows that the slowdown caused by updating the MHB in software (FMM.Sw) is modest. This agrees with [9]. On average, FMM.Sw takes 6% longer to run than FMM. FMM.Sw eliminates the need for the ULOG hardware in Table 2, although it still needs the other FMM hardware in the table.

5.3 Evaluation of the Chip Multiprocessor (CMP)

Figure 11 repeats Figure 9 for the CMP architecture. Overall, we see that the trends are the same as in the NUMA architecture. The most obvious change is that the relative differences between the different buffering schemes are smaller in the CMP than in the NUMA 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 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 much 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 3) 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 is still good: when applied to SingleT Eager, it reduces the execution time by 23% on average (compared to 32% in NUMA).

Finally, a comparison between Lazy AMM and FMM schemes for CMP is not shown because it is very similar to Figure 10. 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 NUMA; 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 NUMA. Indeed, adding laziness to the MultiT&MV Eager AMM scheme reduces the execution time by an additional 24% (Figure 9). In our CMP, however, the gains are only 3% (Figure 11).

A third conclusion is that the resulting system (MultiT&MV Lazy AMM) is competitive against what Table 2 billed as the most complex system: MultiT&MV FMM. The Lazy AMM scheme is generally as fast as the FMM scheme (Figure 10). While Lazy AMM is not as tolerant of high capacity and conflict pressure on the buffers (P3m in Figure 10), it behaves better when squashes due to dependence violations are frequent (Euler in Figure 10).

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 9 and 11), while it requires support beyond SingleT (Table 2).

6 Conclusion

The contribution of this paper is threefold. First, it introduces a novel taxonomy of approaches to buffer multi-version memory state for thread-level speculation 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.

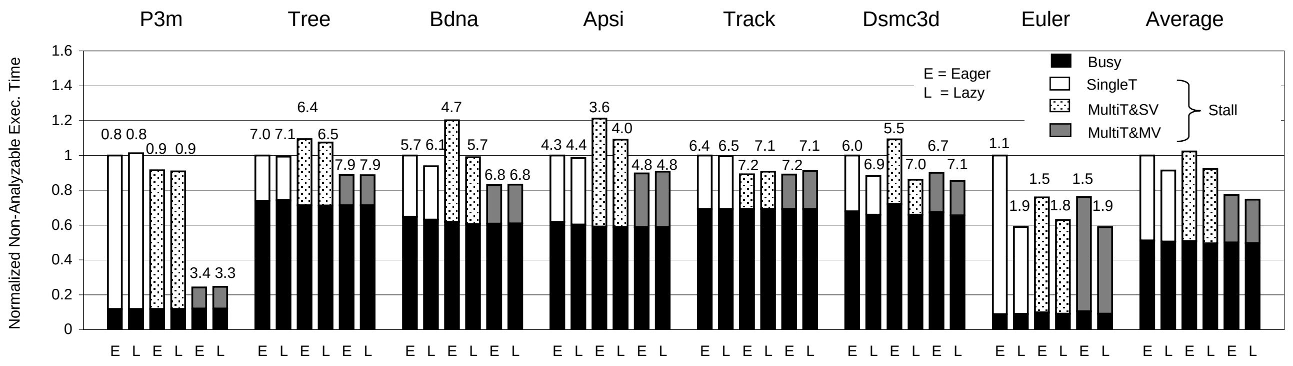


Figure 11. Supporting architectural main memory (AMM) schemes in a CMP. In the figure, *E* and *L* stand for Eager and Lazy, respectively.

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&versions per processor (MultiT&MV Eager AMM). Then, performance can 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 similar performance.

References

[1] H. Akkary and M. A. Driscoll. A Dynamic Multithreading Processor. In *Intl. Symp. on Microarchitecture*, pages 226–236, Dec. 1998.

[2] J. E. Barnes. ftp://hubble.ifa.hawaii.edu/pub/barnes/treecode/. *University of Hawaii*, 1994.

[3] W. Blume et al. Advanced Program Restructuring for High-Performance Computers with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[4] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *Proc. 27th Annual Intl. Symp. on Computer Architecture*, pages 13–24, June 2000.

[5] R. Eigenmann, J. Hoeflinger, and D. Padua. On the Automatic Parallelization of the Perfect Benchmarks. In *IEEE Trans. Parallel and Distributed Systems*, volume 9, pages 5–23, January 1998.

[6] R. Figueiredo and J. Fortes. Hardware Support for Extracting Coarse-grain Speculative Parallelism in Distributed Shared-memory Multiprocesors. In *Proc. Intl. Conf. on Parallel Processing*, September 2001.

[7] M. Frank, W. Lee, and S. Amarasinghe. A Software Framework for Supporting General Purpose Applications on Raw Computation Fabrics. Tech. Rep., MIT/LCS Technical Memo MIT-LCS-TM-619, July 2001.

[8] M. Franklin and G. S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Trans. Computers*, 45(5):552–571, May 1996.

[9] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. Software Logging under Speculative Parallelization. In *Workshop on Memory Performance Issues, in conjunction with ISCA-28*, July 2001.

[10] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. In *Proc. 4th Intl. Symp. on High-Performance Computer Architecture*, pages 195–205, February 1998.

[11] M. Gupta and R. Nim. Techniques for Speculative Run-Time Parallelization of Loops. In *Proc. Supercomputing 1998*, November 1998.

[12] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *8th Intl. Conf. on Arch. Support for Prog. Lang. and Oper. Systems*, pages 58–69, October 1998.

[13] T. Knight. An Architecture for Mostly Functional Languages. In *ACM Lisp and Functional Programming Conf.*, pages 500–519, August 1986.

[14] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Trans. on Computers*, pages 866–880, September 1999.

[15] P. Marcuello and A. González. Clustered Speculative Multithreaded Processors. In *Proc. 1999 Intl. Conf. on Supercomputing*, pages 365–372, June 1999.

[16] M. Prvulovic, , M. J. Garzarán, L. Rauchwerger, and J. Torrellas. Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization. In *Proc. 28th Annual Intl. Symp. on Computer Architecture*, pages 204–215, July 2001.

[17] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In *Proc. SIGPLAN 1995 Conf. on Prog. Lang. Design and Implementation*, pages 218–232, June 1995.

[18] P. Rundberg and P. Stenström. Low-Cost Thread-Level Data Dependence Speculation on Multiprocessors. In *4th Workshop on Multithreaded Execution, Architecture and Compilation*, December 2000.

[19] J. E. Smith and A. R. Pleszkun. Implementing Precise Interrupts in Pipelined Processors. *IEEE Trans. Computers*, C-37(5):562–573, May 1988.

[20] G. S. Sohi, S. Breach, and S. Vijaykumar. Multiscalar Processors. In *Proc. 22nd Annual Intl. Symp. on Computer Architecture*, pages 414–425, June 1995.

[21] J. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proc. 27th Annual Intl. Symp. on Computer Architecture*, pages 1–12, June 2000.

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

[23] M. Tremblay. MAJC: Microprocessor Architecture for Java Computing. Hot Chips, August 1999.

[24] J. Y. Tsai, J. Huang, C. Amlo, D. Lilja, and P. C. Yew. The Superthreaded Processor Architecture. *IEEE Trans. on Computers*, 48(9):881–902, September 1999.

[25] Y. Zhang. Hardware for Speculative Run-Time Parallelization in DSM Multiprocessors. Ph.D. Thesis, Dept. of Elec. and Comp. Engineering, Univ. of Illinois at Urbana-Champaign, May 1999.

[26] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Parallelization of Partially-Parallel Loops in DSM Multiprocessors. In *Proc. 5th Intl. Symp. on High-Performance Computer Architecture*, pages 135–139, January 1999.