Using Software Logging to Support Multi-Version Buffering in Thread-Level Speculation *

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

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

†Universidad de Zaragoza, Spain
victor@posta.unizar.es

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

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

Abstract

In Thread-Level Speculation (TLS), speculative tasks generate memory state that cannot simply be combined with the rest of the system because it is unsafe. One way to deal with this difficulty is to allow speculative state to merge with memory but back up in an undo log the data that will be overwritten. Such undo log can be used to roll back to a safe state if a violation occurs. This approach is said to use Future Main Memory (FMM), as memory keeps the most speculative state.

While the aggressive approach of FMM systems often delivers better performance than more conservative approaches, it also requires additional hardware support. To simplify the design of FMM systems, this paper proposes a *software-only* design for the undo log system. We show that an FMM system with software logging is a good design point: the design has less implementation complexity than an FMM system with hardware logs, and it only reduces performance moderately. In particular, in a simulated 16-processor machine, applications take only 10% longer to execute than if the system had the logging system fully implemented in hardware.

1 Introduction

Thread-Level Speculation (TLS) attempts to extract parallelism from hard-to-analyze codes, like those with pointers, indirectly-indexed structures, interprocedural dependences, or input-dependent patterns. With TLS, a program is sliced into a sequence of tasks that are speculatively executed in parallel. The sequential semantics of the program creates dependences between tasks. If any such dependence is violated at run time by the parallel execution, TLS rolls back the incorrectly-executed tasks and re-executes them. To be able to do this, TLS schemes keep the memory state of speculative tasks separated in some way, often buffered away in caches. With this support, a task is rolled back (squashed) by discarding its buffered memory state and restoring the register state saved at task initiation.

In TLS systems, a task that can still be squashed is *speculative*. Once a task is known to have executed correctly, it becomes non-speculative. Non-speculative tasks *commit* in order by freeing their saved register state and allowing their memory state to be merged with the safe state of the program.

Many different schemes for TLS have been proposed. They range from software-only [10, 18, 19] to hardware-based systems [3, 9, 12, 13, 14, 15, 21, 22, 23, 24, 25]. They target small machines [9, 12, 14, 15, 21, 23] or large ones [3, 22, 24, 25].

Each scheme for TLS has to solve two major problems: detection of dependence violations and, if a violation occurs, state repair. To detect data dependence violations, TLS schemes mark in some way the data that a speculative task accesses, typically with a task-ID tag. Later, if a conflicting access (e.g. a write from another task) occurs that should have preceded the first one in sequential order, a data dependence violation is detected. As for state repair, the key support is to keep the unsafe memory state that a speculative task generates as it executes, somehow separated from the rest of the system. This state will be discarded if the task is squashed. This second problem is the subject of this paper.

1.1 Multi-Version Speculative State Buffering

Different approaches have been proposed to manage the unsafe memory state of a speculative task. In some cases, this state is kept buffered in caches [3, 5, 9, 14, 22] or special buffers [7, 12, 17, 23]. In other cases, the speculative state is merged with memory, but the values to be overwritten in the process are saved in a hardware undo log [24, 25]. Finally, some other schemes use software-only support to do a limited form of buffering [6, 10, 18]. These schemes simply back up the state that existed at the very begin-

*This work was supported in part by the National Science Foundation under grants EIA-0081307, EIA-0072102, and EIA-0103741; DARPA under grant F30602-01-C-0078; DOE under grant B347886; Ministry of Education of Spain under grant TIC 2001-0995-C02-02; and gifts from IBM, Intel and HP.

ning of the speculative section of the code. If a violation occurs while executing, the program rolls back to the beginning of the speculative section.

Another important issue is whether the buffer for speculative state is designed to hold the state of a single or multiple speculative tasks. If the former, a new task cannot typically start to execute on a processor until the task that previously ran there becomes non-speculative [7, 9, 14, 15, 23]. This limitation can hurt performance.

On the other hand, if the buffer is designed to hold the state of multiple speculative tasks, it may have to buffer multiple versions of the same variable, one for each local task. This occurs, for example, with mostly-privatization patterns, where task accesses end up being private most (but not all) of the time, but the compiler cannot prove as private. As a result, many tasks generate their own versions of the same variable. To support this case, some designs use the different ways of an associative cache to store the potentially multiple versions of the same variable [3, 17, 22].

To organize the design space, a taxonomy of buffering approaches [8] classified buffering into two main approaches, depending on when the state that tasks produce can be merged with the main memory system. In systems with Architectural Main Memory (AMM), task state can only be merged with main memory when it is safe, namely at task commit time or later. In systems with Future Main Memory (FMM), task state can be merged at any time, therefore potentially dumping speculative state into main memory. However, before a task generates a speculative version of a variable, the previous version of the variable is saved in a buffer or log. This log is kept in case recovery is needed.

It was shown that, in applications with few dependence violations, the aggressive approach of FMM schemes often delivers faster application execution. However, FMM schemes also need more hardware support. In particular, one major support is the per-processor undo log machinery.

1.2 Goal of the Paper

To address this problem, in this paper, we design an undo log system that implements buffering for FMM systems *in software*. With this support, we simplify the implementation complexity of FMM systems while only reducing performance moderately. We present a design that makes FMM buffering using software undo logging cost-effective. For a simulated 16-processor machine, our scheme only introduces an average execution overhead of 10% when compared to a hardware-only FMM buffering scheme, while significantly reducing the hardware complexity. For reference, we also compare our FMM design to an advanced AMM scheme.

This paper is organized as follows: Section 2 reviews the buffering taxonomy; Section 3 describes the hardware support used; Section 4 presents the software support; Section 5 discusses our evaluation environment; and Section 6 evaluates our design.

2 A Taxonomy of Buffering

The taxonomy of buffering approaches presented in [8] classifies buffering into two main approaches, depending on when the state that tasks produce can be merged with the main memory system. These two approaches are systems with *Architectural Main Memory* (AMM) and systems with *Future Main Memory* (FMM).

In AMM systems, task state can only be merged with main memory when it is safe, namely at task commit time or later. Note that state merging may involve write-backs of dirty lines to memory [3] or ownership requests for these lines to obtain coherence with main memory [22]. Also, state merging can occur eagerly at commit time or lazily after that, typically in a line displacement.

To gain insight, we can use an analogy with the concepts used in the management of the register file in processors [20]. As instructions complete out of order, the reorder buffer keeps the register updates. Only when instructions finally commit (in order) are the updates moved to the architectural register file. In AMM systems, buffers or caches become analogous to a reorder buffer for memory state. Consequently, we call them a distributed Memory-System Reorder Buffer (MROB).

In FMM systems, task state can be merged at any time, therefore potentially storing speculative data in main memory. However, to enable recovery from task squashes, before a task generates a speculative version of a variable, the previous version of the variable is copied into a buffer or cache. This state is kept separated from the main memory state.

Using a register file analogy, consider a different way of managing the registers. As instructions complete out of order, they update a ”future” register file, which always keeps the very latest values of the registers. Since these values may be unsafe, before one of the registers is updated, a copy of the old value is saved in the history buffer. The history buffer is used in an exception to revert the registers to a safe value. In FMM systems, buffers or caches with older copies become analogous to a history buffer for memory state. Consequently, we call them a distributed Memory-System History Buffer (MHB).

To understand the differences between AMM and FMM schemes, consider an example of a program where each task generates its own private version of variable *X*. Figure 1-(a) shows the code for two tasks that run on the same processor. Assume that both tasks are speculative.

If we use an AMM scheme, when the second speculative version of *X* is created, it is tagged with its address and Task ID, and kept in the same buffer as the previous speculative version (Figure 1-(b)). Versions remain in the buffer (e.g. cache) at least while speculative. If we use an FMM scheme (Figure 1-(c)), when the second version is created, the first one is saved in the MHB. Since we need to know what versions we have in the MHB, we also need to save the ID of the task that generated that version (*Producer Task ID i* in the MHB of Figure 1-(c)). Finally, groups of MHB

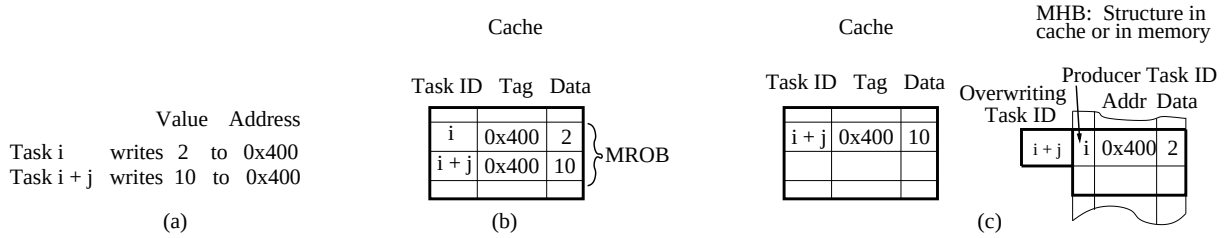

Figure 1. Implementing the MROB and the MHB.

entries are also tagged with the *Overwriting Task ID* (*i+j* in the MHB of Figure 1-(c)). Note that the MHB can be a structure in the cache or in main memory.

2.1 Tradeoffs AMM/FMM

It was shown that, in applications with few violations, FMM schemes tend to be faster because they streamline version commit [8]. To see why, consider AMM systems. In AMM systems, speculative versions of the same variable are all kept in the cache until they can commit, to prevent overwriting the architectural version in main memory. Unfortunately, the cache may fill up and the processor may have to stall to prevent the displacement of uncommitted versions. A partial solution is to provide a special memory area where speculative versions can safely overflow into [17]. Unfortunately, such an overflow area is slow when asked to return versions, which especially hurts when committing a task.

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* in the MHB, 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, that new version simply commits by default. The older version in the MHB can *also be safely displaced* from the cache and written back to memory at any time. Since it has a different address, it does not overwrite good data in memory. Overall, in FMM systems, cache overflows do not hurt performance.

Unfortunately, FMM schemes need more hardware support. In particular, one major support is the MHB, which has been implemented as a per-processor, sequentially-accessed hardware undo log [24, 25]. When a task updates a variable for the task's first time, a log entry is created. Logs are later accessed in a violation to recover a safe state.

FMM systems also need additional supports, such as a hardware module for data tagging in memory. However, the difference in complexity between such supports and the corresponding AMM structures is less clear and more implementation-dependent than the more obvious case of the MHB support [8]. Consequently, we focus this paper on reducing the cost of the MHB in FMM systems by implementing it in software.

Since the MHB is an undo log, in the rest of the paper, we will refer to it using the term *log*.

3 Hardware Support

Even a software implementation of the log needs some hardware support. In this section we start by describing some relevant aspects of the TLS protocol used (Section 3.1). Then, we describe the hardware hooks needed to support our software implementation of the log (Section 3.2). Finally, we outline a hardware-only implementation of the log that has been proposed in the past and that will be compared to our proposal (Section 3.3).

3.1 Hardware TLS Protocol Used

The design of the software log is affected by some aspects of the TLS protocol used in the machine. Consequently, we outline some relevant aspects of the TLS protocol assumed. The complete protocol is described in [24].

As shown in Figure 1-(c), a log has to store the IDs of both the task that produced the overwritten version of the data and the task that overwrites that version. Consequently, there are two main aspects of the TLS protocol that are relevant to logging: the grain size of the data associated with a task ID, and how/where the task IDs are stored.

The protocol that we use logically associates task-ID information to words, rather than cache lines. Consequently, on a write, we will generate a log entry with information corresponding to a single word. Moreover we only update the task ID of the written word.

To reduce cache real state, the protocol does not store the task IDs in the cache. Instead, they are in (local) memory. To reduce messages to these task IDs, each cached word stores two bits that summarize the current task's pattern of access to the word. With this support, task IDs in local memory are usually accessed only in the first read and write to the work by the task.

The actual task ID and cache bit information recorded is similar to what some other TLS protocols use. It involves recording the identity of the tasks that write, and those that read before writing. The latter operation is also called exposed read. Such information

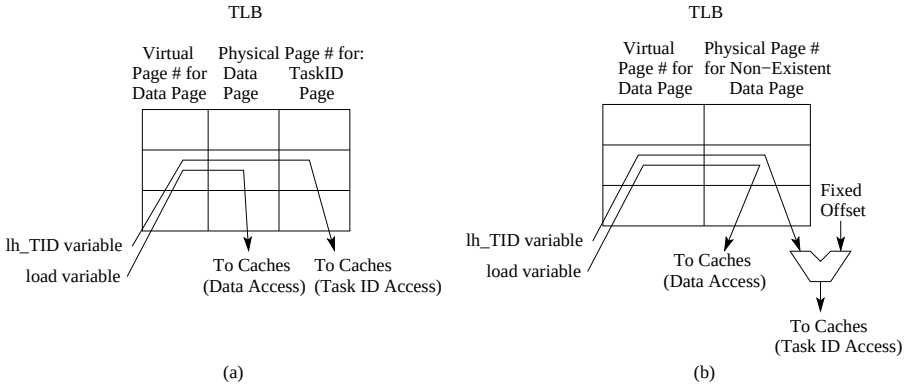

Figure 2. Address translation for a plain load and a *lh_TID* load using a modified TLB (a) and an unmodified TLB (b).

is used for dependence checking. In our protocol, each cached word has two task IDs in memory: the ID of the latest local task that wrote it (*MaxW*) and the latest local task that expose-read it (*MaxR1st*). Then, the cache tags keep one *Write* and one *Read1st* bit per word. They indicate whether the currently-running task has issued a write or an exposed read to the word, respectively. These bits are cleared when a new task starts.

3.2 Hooks to Support Software Logging

When building a log in software, we must be able to read the *MaxW* task ID of a variable in software. Such *MaxW* will be saved as the Producer Task ID, as shown in the log of Figure 1-(c). Unfortunately, in the baseline TLS protocol, task IDs in memory are managed in hardware. Furthermore, since there is little provision for software access, they are not even mapped in the virtual address space of the tasks. Consequently, to implement software logging, we must find a way to make task IDs visible to the software (Section 3.2.1) and even bring them into the caches (Section 3.2.2).

3.2.1 Making Task IDs Visible to Software

Conceptually, one solution is to add an extra field to each TLB entry. This field stores the number of the physical page that contains the corresponding task IDs in local memory. A TLB access can now generate the physical addresses of both the data and their corresponding task IDs. With this setup, we can read the task ID of a variable with a new instruction called *load half-word task ID (lh_TID)*. This instruction takes the virtual address of a variable and a destination register. When this instruction executes, the TLB does not deliver the physical page of the variable; instead, it delivers the physical page of the variable's task ID (Figure 2-(a)). The physical address issued to memory is that of the two task IDs associated with the variable. That location contains the 16-bit *MaxR1st* and the 16-bit *MaxW*. A load half-word allows us to get the least significant 16 bits, which is where *MaxW* is. The *MaxW* is loaded into the cache and into the destination register. Overall, with this support, the software can read *MaxW* as data, while needing only the virtual address of the variable.

We implement this idea without modifying the TLB. In our baseline TLS protocol [24], when a processor touches a page of speculation data for the processor's first time, the OS allocates the page of task IDs for the processor in local memory. At that time, the TLB mapping of the virtual page for the data can be changed to point to a non-existent physical page, located at a fixed address offset from the physical page that contains the task IDs. Subsequent CPU accesses to the data obtain from the TLB such (non-existent) physical addresses, which are used in the tags of the caches. In case of a cache miss, the true physical address of the data is obtained from a translation module similar to that in Prism [4], WildFire [11], or S3.MP [16]. On the other hand, the *lh_TID* instruction to the virtual address of the data obtains the non-existent physical address from the TLB and, in hardware, subtracts the offset (Figure 2-(b)). The result is an access to the address of the task ID of the variable, as intended.

3.2.2 Caching Task IDs

We want *lh_TID* to bring a copy of *MaxW* into the cache and reuse it from there in subsequent accesses. However, the *MaxW* master copy in memory may be modified in hardware by the baseline TLS protocol as the program executes. Specifically, every first-write by a local task to the variable will trigger the hardware to set *MaxW* to the ID of the writing task. Consequently, to keep the cached copy up to date, we must update it in software to the same value that the hardware updates the master copy to.

We accomplish this with a *store half-word task ID (sh_TID)* instruction. After a task creates a log entry in software, it writes its own task ID to the cached copy of the *MaxW* of the overwritten variable. For this write, it uses *sh_TID*. This operation updates the cached copy to the correct value. *sh_TID* takes a source register and, as in *lh_TID*, the virtual address of the variable. It updates in the cache the half word where the *MaxW* copy is. Note that the master task ID in memory is unaffected. The reason is that

the pages for the master task IDs in memory are *read-only* for the software and for cache displacements.

With this support, we are storing software-updated task IDs in the cache while keeping the hardware-managed master copy in memory. This is a trade-off decision that maximizes the performance of software logging while minimizing the additional support required. The coherence of a cached task ID is ensured, even in the presence of cache displacements, if we execute *sh_TID* every time that the protocol modifies the master copy of the task ID.

3.3 Alternative: Hardware-Only Logging

In the evaluation section, we compare our software logging system to a high-end hardware-only implementation of logging described in [25]. The latter uses a hardware logging module included in the directory controller of each node. The two systems have three main differences. First, in the hardware implementation, entries are inserted into the log in the background, with no overhead visible to the program. Similarly, log recycling has practically no overhead. In the software implementation, instead, the processor is directly involved in inserting entries into the log. The resulting overhead plus the overhead of log recycling are visible to the program.

The second difference is that the hardware implementation keeps the log in memory, therefore avoiding cache pollution. The opposite is the case for the software implementation.

Finally, the hardware implementation can only record in the log the *physical* addresses of the logged variables. As a result, a recovery, which involves reading the logged data and writing some of them back to their original positions, must be performed by exception handlers with privilege to write directly to physical addresses. In addition, the mapping of the pages of data cannot change at run time. If it did, the recovery exception handler writing directly to physical addresses could update the wrong locations. Since the software implementation stores *virtual* addresses in the log, these problems are not present.

4 Software Support

To understand the software requirements of efficient software logging in a TLS protocol, we first analyze the log operations that need to be supported (Section 4.1). Then, we present our algorithm for automatic code instrumentation (Section 4.2).

4.1 Required Log Operations

A logging system 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 3 shows the per-processor software structures that we use to implement the undo log. 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.

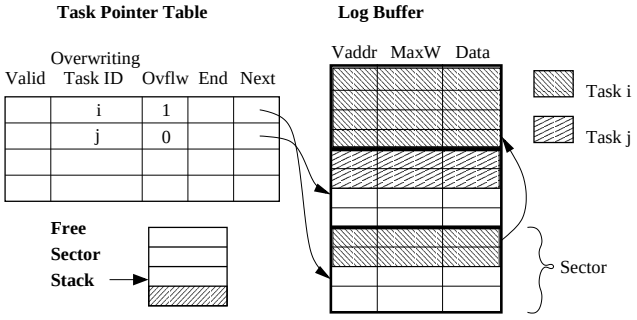

Figure 3. Per-processor software structures that we use to implement the undo log.

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 3). If the Free Sector Stack runs out of entries, we resize the Log Buffer and Stack accordingly.

With these software structures, the four operations are supported as follows:

Insertion. Insertion is the only truly overhead-sensitive operation because it occurs frequently. At compile time, the compiler instruments stores in the code with instructions to save a log record. As shown in Figure 3, 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 (*MaxW*), and the value before the update. Section 3.2.1 discussed how *MaxW* is obtained in software. 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 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. While recycling in this manner can produce some fragmentation in the Log Buffer, we did not take any special action to reduce it in our experiments.

```
        ; r1 = upper limit of the sector. r2 = address in memory to insert the log record
        ; offset(r3) = address of the variable to update. r5 = ID of the executing task
        bgt       r1, r2, insertion          ; check for sector overflow
        ... allocate another sector
insertion:
        addu      r4, r3, offset             ; compute address of variable
        sw        r4, 0(r2)                  ; store in log
        lh_TID    r4, offset(r3)             ; load the 16-bit MaxW task ID
        sw        r4, 4(r2)                  ; store as a full word in log
        lw        r4, offset(r3)             ; load value of variable
        sw        r4, 8(r2)                  ; store in log
        addu      r2, r2, log_record_size    ; increment pointer
        sh_TID    r5, offset(r3)             ; update cached MaxW
```

Figure 4. Instructions that we add before every speculative store in the program.

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 algorithms used are discussed in Appendix A.

4.2 Algorithm for Code Instrumentation

We would like the compiler to 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. Consequently, in this paper, we focus on the algorithm that instruments the code to perform log insertions. We start with a naive algorithm and then present an optimized one.

4.2.1 Naive Algorithm

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 3). Figure 4 shows the MIPS assembly instructions that our automatic pass adds before every speculative store in the program. We call a store or a load speculative if it may access data whose access patterns cannot be fully analyzed by the compiler; speculative stores and loads trigger our TLS protocol transactions, as described in [24].

From Figure 4, we see that we need 9 instructions: 1 to check for sector overflow, 6 to collect and insert the information, 1 to increment the pointer, and 1 to update the cached task ID. Note that we use the new *lh_TID* and *sh_TID* instructions from Section 3.2. In the code, the load that reads the value of the variable to be logged is treated as non-speculative.

Occasionally, fewer instructions are needed. For example, if the compiler knows the dynamic number of speculative stores in a region, it may only need to check for sector overflow once. Furthermore, if the compiler knows the exact number and order of the stores to log, it can hard-code the offsets in the writes to the log and avoid updating the pointer.

Finally, to be able to undo tasks, we must also log variables written with non-speculative stores. However, in this case, we do not save task IDs in the log because non-speculative variables do not have them.

4.2.2 Reducing Overhead by Filtering the First Store

When we backtrack execution to recover from a violation, we undo whole tasks only; it would be too complicated to support undoing parts of a task. In addition, when we need to retrieve a logged value, we only want the value left at the end of a task; we never use a value from the middle of another task. We can use these facts to reduce instrumentation. Specifically, the log only needs to save the value overwritten by the "*first store*" to the variable in the task. Consequently, the compiler should identify first stores in a task and instrument only those.

Identifying first stores is easy for variables accessed with non-speculative accesses, since their dependence structure is analyzable. However, it is hard for variables accessed with speculative accesses. For these, we modify the instrumentation in Figure 4 to dynamically test whether or not a store is a first store, and only log if it is. To identify first stores, we propose two possible approaches.

Filtering Using Task IDs

To identify first stores, we compare the ID of the writing task to the *MaxW* of the variable. The latter is the ID of the task that produced the version. If they are the same, this is not a first store and logging is skipped. This approach is implemented in Figure 5-(a), which repeats Figure 4 with the new extension.

```
                ; r2 = address in memory to insert the log record
                ; offset(r3) = address of the variable to write
                ; r5 = ID of the executing task
    lh_TID      r6, offset(r3)          ; load MaxW
    beq         r6, r5, no_insert       ; first store?
    addu        r4, r3, offset          ; insert as usual
    sw          r4, 0(r2)
    sw          r6, 4(r2)
    lw          r4, offset(r3)
    sw          r4, 8(r2)
    addu        r2, r2,log_record_size
    sh_TID      r5, offset(r3)
no_insert:
```

(a)

```
    xlw         r6, r1, offset(r3)  ;Write bit goes to r6
    bgtz        r6, no_insert       ; first store?
    addu        r4, r3, offset      ; insert as usual
    sw          r4, 0(r2)
    lh_TID      r4, offset(r3)
    sw          r4, 4(r2)
    sw          r1, 8(r2)
    addu        r2, r2,log_record_size
    sh_TID      r5, offset(r3)
no_insert:
```

(b)

Figure 5. Filtering "*first stores*" using task IDs (a) and extended loads (b). For simplicity, we do not include the instruction that checks for sector overflow.

Filtering Using Extended Loads

The second way to identify first stores is to use our *Write* bit present in the cache tags. As indicated in Section 3.1, this bit is set for a word when the current task updates it for the first time. Consequently, we load the *Write* bit on a register and test it. If it is already set, this is not a first store and logging is skipped.

This approach avoids cache pollution because *MaxW* is only loaded into the cache if logging is necessary. In the previous approaches, *MaxW* is loaded into the cache every time. The disadvantage of this approach, however, is that it needs special support to load the *Write* bit. For example, we could define an *Extended Load* that loads a variable into a register and its *Write* bit into another one.

This approach is used in Figure 5-(b), where we use *xlw Ri Rj Addr* for an extended load that loads the data into *Rj* and the *Write* bit into *Ri*.

5 Evaluation Methodology

5.1 Simulation Environment

We use an execution-driven simulation system to model in detail a 16-node CC-NUMA multiprocessor. Each node contains one 4-issue dynamic superscalar processor 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 saturating counters, and an 8-cycle branch penalty.

Each processor has a 2-way 32-Kbyte L1 data cache and a 4-way 512-Kbyte L2, both with 64-byte lines and a write-back policy. The nodes are connected with a 2D mesh. The average no-contention round-trip latencies from the processor to the on-chip L1 cache, L2 cache, memory in the local node, and memory in a remote node 2 and 3 protocol hops away are 1, 12, 75, 208 and 291 cycles, respectively.

We use release consistency and a cache coherence protocol like that of DASH. Pages of shared data are allocated round-robin across the nodes. We choose this allocation because our applications have irregular access patterns and the tasks are dynamically scheduled; it is not possible to optimize shared data allocation at compile time. Private data are allocated locally.

We use the TLS protocol outlined in Section 3.1. In the evaluation, we simulate all software overheads, including allocation and recycling of log sectors, and the dynamic scheduling and committing of tasks. We wrote software handlers for parallel recovery after a dependence violation and to retrieve data from logs (Section 4.1). In addition, a processor that allocates a page of task IDs is penalized with 4,000 cycles.

5.2 Workloads

We use a set of numerical applications where a large fraction of the code is not analyzable by a parallelizing compiler. These applications are: *Apsi* from SPECfp2000, *Track* from Perfect, *Dsmc3d* and *Euler* from HPF-2, *P3m* from NCSA, and *Tree* from [1]. We use the Polaris parallelizing compiler [2] to identify the non-analyzable sections and prepare them for speculative parallelization. The source of non-analyzability is that the dependence structure is either too complicated or unknown because it depends on input data. For example, the code often has arrays with subscripted subscripts, and sections with complex control flow. In these cases, Polaris marks the speculative references, which will trigger TLS protocol actions. Polaris also identifies store instructions that may need to be logged and instruments them according to Section 4.2.

Table 1 shows the non-analyzable sections in each application. These sections are loops. 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 61.2%. The table also shows the number of invocations of these loops during program execution, the average number of tasks per invocation, and the average number of instructions per task. Finally, the last three columns show the weight of several types of references: speculative reads and speculative writes as a percentage of the *total references* in

Appl	Non-Analyzable Sections (Loops)	% of Tseq	# of Invoc	Tasks per Invoc	Instruc per Task (Thousand)	Spec Refs / Total Refs (%)		Instrum Wr / Total Instructs (%)
						Rd	Wr	
P3m	*pp_do100*	56.5	1	97336	69.1	11.1	2.1	0.8
Tree	*accel_do10*	92.2	41	4096	28.7	2.9	2.9	0.8
Apsi	*run_do[20,30, 40,50,60,100]*	29.3	900	63	102.6	49.4	33.4	11.6
Track	*nlfi lt_do300*	58.1	56	126	22.3	0.3	0.2	0.4
Dsmc3d	*move3_goto100*	41.2	80	46777	5.4	0.0	0.0	1.2
Euler	*dflux_do[100,200] psmoo_do20 eflux_do[100,200,300]*	89.8	120	1871	3.9	30.3	30.3	11.9
Average		61.2	200	25045	38.7	15.7	11.5	4.5

Table 1. 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.

the section, and instrumented writes (including both speculative and non-speculative) as a percentage of the *total instructions* in the section. All counts are dynamic. Note that all the data presented in the evaluation, including speedups, refer only to the code sections in the table.

The applications exhibit a range of dependence violation behaviors. For example, *Euler*'s execution is substantially affected by squashes (0.02 squashes per committed task). However, dependence violations affect the performance of the other applications only modestly or not at all. When a RAW dependence violation occurs, the recovery exception handler is invoked (Section 4.1). The retrieval exception handler (Section 4.1) is not executed in any of the applications: although in-order RAW dependences appear in the applications, the requested version is found in a cache instead of in a log.

6 Evaluation

To evaluate software logging, we examine its overall impact on execution time, quantify its overheads, and finally look at alternative designs.

6.1 Impact of Software Logging on Execution Time

To assess the impact of software logging on execution time, we compare two FMM systems that are identical except that one uses software logging (*FMM.Sw*) and the other hardware logging (*FMM.Hw*). *FMM.Sw* uses our advanced scheme of Figure 5-(a), which filters first stores using task IDs. *FMM.Hw* uses the hardware logging support of Section 3.3.

For reference, we also compare these two systems to a very advanced AMM system (*AMM.Lazy*). Using the terminology in [8], the AMM system chosen is multi-version and lazy. In this system, individual buffers can support the state of multiple speculative tasks and multiple speculative versions of the same variable; moreover, data from committed tasks merge with main memory *lazily*, when lines are displaced from the cache or when they are requested by another processor. Comparing *AMM.Lazy* to *FMM.Hw*, we argue that *FMM.Hw* has a higher implementation complexity, mostly because of the logging support. However, if we compare *AMM.Lazy* to *FMM.Sw*, since logging is now supported in software, we feel that both schemes have a roughly comparable degree of implementation complexity.

In our design of *FMM.Sw*, we minimize L1 pollution due to software logging by making logs L1-uncacheable. Moreover, log records created by committed tasks are regularly recycled in software. For comparison, in *FMM.Hw*, all log operations except recovery are performed with neither instruction overhead nor cache pollution. Recovery in *FMM.Hw* is performed with a software handler as in *FMM.Sw*.

Figure 6 compares the execution time of the applications running on 16 processors for these three systems. For each application, the bars are normalized to *FMM.Hw* and broken down into execution of instructions (*Busy*), waiting on data, control, and structural pipeline hazards (*Hazard*), synchronization (*Sync*), and waiting on data from the memory system (*Memory*). Note that *Busy* includes useful application instructions as well as squashed instructions and software-logging instructions. Finally, the numbers on top of each bar show the speedup relative to the sequential execution of the code, with all the application data placed in the local memory of the single active processor[1].

Comparing *FMM.Hw* to *FMM.Sw*, we see the overheads introduced by our software implementation of logging. This overhead is only noticeable in *P3m*, *Apsi*, and *Euler*. The overhead comes from the more instructions executed (*Busy* time), and the higher memory stall due to accesses to logs and task IDs (*Memory* time). As a result, *Hazard* and *Sync* may also increase. On average, the execution time of *FMM.Sw* is only 10% higher than *FMM.Hw*. This is a modest and very tolerable overhead, given that *FMM.Sw* removes the complexity of implementing the log in hardware.

[1] A few of the bars in Figure 6 are slightly different from [8]. The reason is that the TLS protocol used in the two papers has a few differences. Note also that one of the applications in [8] (Bdna) has been removed because it has been parallelized by a research compiler.

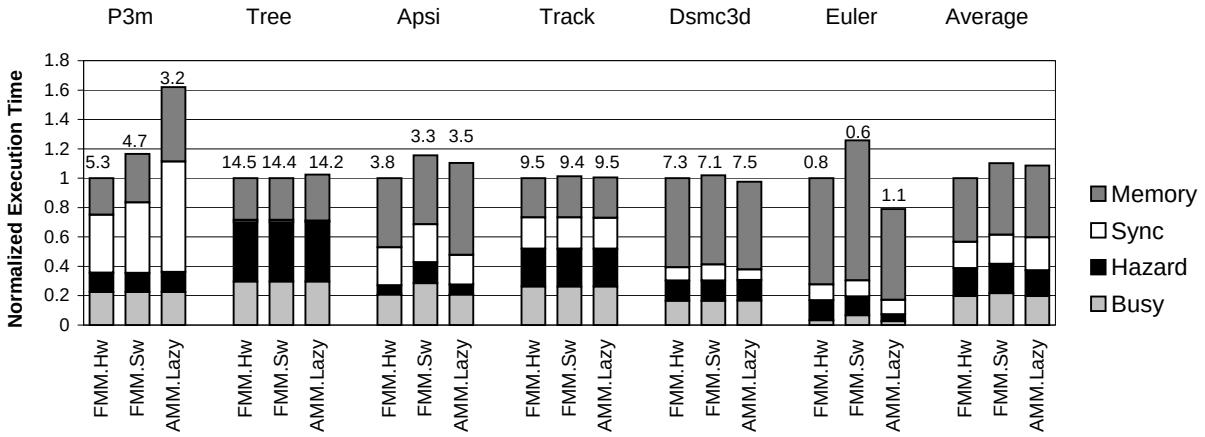

Figure 6. Execution time of the applications on a 16-node multiprocessor with FMM and either hardware (*FMM.Hw*) or software (*FMM.Sw*) logging support. For comparison, the figure includes a system with advanced AMM (*AMM.Lazy*).

If we compare *FMM.Sw* to *AMM.Lazy*, we see that the two systems have similar performance on average. Each system, however, has a different weakness. *AMM.Lazy* is slower in applications such as *P3m*, where caches have to hold the state of numerous speculative tasks and where multiple speculative versions of the same variable compete for the same cache set. However, *FMM.Sw* is slower in applications with frequent violations such as *Euler*. In *Euler*, about 15% of the total execution time of *FMM.Sw* is taken up by software handlers for data recovery.

Overall, we argue that *FMM.Sw* is a good design point. It simplifies the hardware of *FMM.Hw* at the cost of a 10% slowdown. The result is a system that is comparable in performance and implementation complexity to an advanced AMM scheme such as *AMM.Lazy*. The choice between *FMM.Sw* and *AMM.Lazy* depends on the mix of applications: *FMM.Sw* has an advantage for applications that put pressure on the caches, while *AMM.Lazy* wins for applications with frequent violations.

6.2 Overheads of Software Logging

We now examine the sources of the 10% average overhead observed for *FMM.Sw*. The bulk of the overhead is induced by the record-insertion operation performed at instrumented stores. This overhead includes the contribution of additional instruction execution and additional memory accesses (Figure 5-(a)). In the following, we examine each of these two effects separately.

Instruction Execution Overhead

The last column of Table 1 shows the number of instrumented stores as a fraction of all instructions executed in the program before instrumentation. Most applications have a small fraction. The exceptions are *Apsi* and *Euler*, where the fractions are 11.6% and 11.9%, respectively. Store instrumentation contributes to the increase in *Useful* time for these two applications in Figure 6.

However, instrumented stores have different costs. At run time, two different cases are possible for speculative stores. If the store turns out not to be a first store, only 3 instructions are executed (including the check for sector overflow); if the store is a first store, 10 instructions are executed. For the less-frequent non-speculative stores, the instrumentation overhead is 6 instructions.

Figure 7 classifies all the dynamic stores in the code into three classes: instrumented and first (*Inst.First*), instrumented and not first (*Inst.Non_First*), and not instrumented (*Non_Inst*). The *Inst.First* category includes the instrumented non-speculative stores.

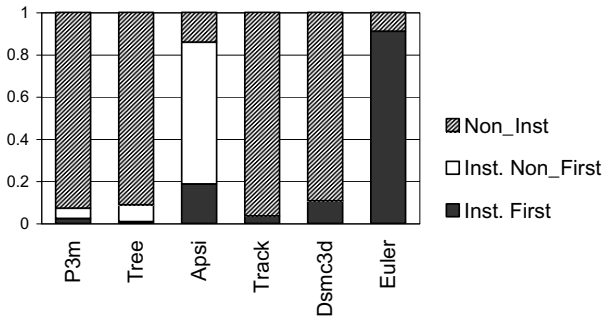

Figure 7. Breakdown of dynamic stores.

Figure 7 shows that attempting to identify first stores at run time paid off: *Apsi* and, to a much lesser extent *Tree* and *P3m*, have many instrumented stores that are proved not to be first stores at run time. In such cases, a significant amount of overhead is eliminated.

Memory Access Overhead

This overhead is due to accesses to two new objects, namely logs and task IDs. Focusing first on logs, Table 3 shows the average size of the log created by a task (Column 2) and the number of tasks that contribute to the log of a processor (Column 4). The other columns in the table will be discussed later.

The column with the log size is labeled *Filter* to indicate that our algorithm filters the first store (Figure 5-(a)). We can see that, for some applications, the average task creates a 7-40 Kbyte log. The column with the number of tasks per log is labeled *Log Recycle* because log records are dynamically recycled. In our algo-

Appl	Log L1 Bypass				No Log L1 Bypass			
	Total L1 Rd Miss Rate (%)		L1 Rd Misses Coming From Task ID Rd (%)		Total L1 Rd Miss Rate (%)		L1 Rd Misses Coming From Task ID Rd (%)	
	Task ID Filtering	Xload Filtering	Task ID Filtering	Xload Filtering	Task ID Filtering	Xload Filtering	Task ID Filtering	Xload Filtering
P3m	8.86	6.20	8.4	0.6	8.86	6.24	7.7	0.6
Tree	21.02	21.11	0.0	0.1	21.11	21.11	0.0	0.0
Apsi	1.01	0.60	40.4	32.6	2.01	0.80	22.2	29.3
Track	1.29	1.29	9.3	9.3	1.37	1.37	9.8	9.8
Dsmc3d	1.90	1.90	0.0	0.0	1.94	1.94	0.1	0.1
Euler	61.03	61.03	22.7	22.7	61.58	61.58	18.1	18.1
Average	15.85	15.35	13.4	10.8	16.14	15.5	9.6	9.6

Table 2. Effect of task ID loads on the L1 miss rate. The experiments correspond to 16-processor runs. *Xload* stands for extended load.

Appl	Log Size per Task (Kbytes)		# Tasks in the Log of a Processor	
			Log Recycle (Measured at End of Task)	No Log Recycle (Measured at End of Code)
	Filter	All	$Max;Avg$	$Max;Avg$
P3m	18.2	56.2	100;50.0	132;80.3
Tree	0.4	3.5	8;2.0	70;64.0
Apsi	40.0	184.0	4;1.8	5;3.9
Track	1.2	1.2	4;1.3	11;6.4
Dsmc3d	1.0	1.0	3;1.1	1136;489.3
Euler	7.0	7.0	2;1.1	40;36.0
Average	11.3	42.1	20;9.5	232;113.3

Table 3. Log statistics. The experiments are for 16-processor runs. Log records are 16 bytes.

rithm, a processor attempts to recycle its log records every time that it is about to start a new task. The table shows the tasks per log in two cases: the average of all processors and the maximum value. These values are obtained by taking a snapshot every time that a processor finishes a task. Note that, for some applications, if we multiply either of these values by the log size per task, we get large memory footprints.

In reality, although logs can grow large, they are primarily accessed in record-inserting operations through stores to consecutive locations. Since write latency is often relatively easy to hide, we expect logs to impact *Memory* stall only in extreme cases such as in *P3m*.

Consider now the task IDs. Task IDs are loaded at every instrumented speculative store. Table 2 shows the contribution of these task ID loads to the total L1 read miss rate. In the table, we focus on the columns under *Log L1 Bypass* and, among those, the first and the third one, which are labeled *Task ID Filtering*. These two columns correspond to our default algorithm that filters with task IDs (Figure 5-(a)). They show the total L1 read miss rate of the application and the contribution of the task ID loads to the read misses, respectively. The other columns in the table will be discussed later.

The data show that task ID loads usually contribute with only 10% or less of the read misses. The exceptions are *Apsi* and *Euler*. In *Apsi*, task ID loads contribute with 40.4% of the misses, but the total miss rate is only 1.01%. In *Euler*, task ID misses contribute significantly to a large L1 miss rate. Consequently, *Euler* is an application where task ID accesses can noticeably affect *Memory* stall.

All these observations are confirmed by Figure 6. The figure shows that, in practice, *Memory* time increases only in *P3m* (where logging contributes) and *Euler* (where task ID accesses contribute).

6.3 Alternative Designs for Software Logging

No First-Store Filtering

This design uses the naive insertion algorithm of Figure 4, where we create a log record at every instrumented store. Figure 8 shows the execution time with this new algorithm, which we label *All*. The system is compared to our *FMM.Sw* system of Figure 6, which we have normalized and re-labeled *Filter*. From the figure, we see a significant increase in the execution time of *Apsi*. The main reason is a large increase in the number of memory accesses, as shown by the larger log created by a task in the *All* column of Table 3. As a result, we recommend to filter first stores.

No Log Recycling or Bypassing the Cache

This design eliminates two log optimizations that minimized L1 pollution: recycling the log space at run time and forcing log accesses to bypass L1. Figure 9 shows the effect of eliminating these optimizations on the execution time. In the figure, *NoRec* means no recycling, while *NoBy* means no bypass. All four possible combinations are shown. For each application, the bars are normalized to bypass and recycle (*By.Rec*), which is our *FMM.Sw* system from Figure 6. While we use the default filtering algorithm in all applications (*Filter*), we also show bars with the *All* algorithm of Figure 8 for *Apsi*.

Figure 9 shows that removing these optimizations does not affect the performance of *Filter* significantly. The only exception is *Apsi*, where L1 bypassing is effective. The small effect of no bypassing can be deduced from Table 2, where the values under *No Log L1 Bypass* are similar to those under *Log L1 Bypass*. The small effect of no recycling is intuitive, given that old log records are rarely accessed again. We note, however, that non-recycled logs can grow very large, as seen in the *No Log Recycle* column of

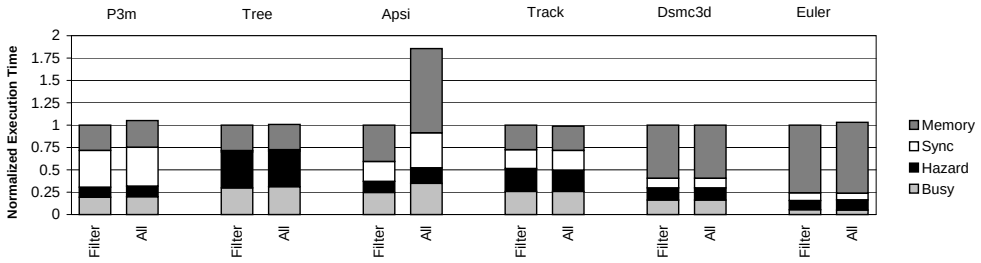

Figure 8. Effect of eliminating first-store filtering.

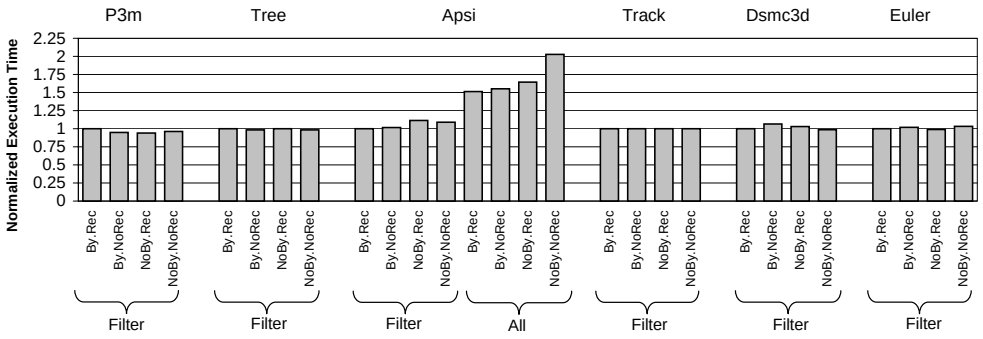

Figure 9. Effect of eliminating log recycling and L1 cache bypassing for logs.

Table 3. Overall, we still recommend log recycling and L1 cache bypassing for logs.

Extended Loads

To reduce the memory overhead induced by task ID loading, we can use extended loads (Figure 5-(b)). With this support, when a store does not need to create a log record (i.e. the store is not a first store), no task ID is loaded into the cache, therefore reducing cache pollution.

Figure 10 shows the resulting impact on the execution time. The figure compares our default system, which uses task IDs for first-store filtering (Task ID) to a system that uses extended loads (Xload). For each case, we show an environment with L1 bypass and recycle for logs (*By.Rec*) and one with no L1 bypass and no recycle (*NoBy.NoRec*). From the figure we can see that only *P3m* and *Apsi* benefit slightly from using extended loads.

To understand the results, consider Table 2, which compares the L1 miss rate using task IDs (*Task ID* columns) and extended loads (*Xload* columns). Using extended loads can only impact applications where a large fraction of the misses come from task ID loading (Columns 4 and 8) and where many instrumented stores turn out not to be first stores (Figure 7). The only applications for which this is true are *P3m* and *Apsi*. Overall, however, since using extended loads does not reduce execution time noticeably, we do not recommend it.

7 Conclusions

In TLS systems with FMM, one of the sources of implementation complexity is the undo log that supports speculative buffering. Such log has traditionally been implemented in hardware. To simplify the design of FMM systems, this paper has proposed a software-only design for the undo log.

The proposed design involves accessing TLS task IDs in software to bring them into the cache and reusing them from there. In addition, application codes are automatically instrumented so that undo log software structures are managed at run time with low overhead.

Overall, an FMM system with software undo logging is a good design point. It is simpler than an FMM system with hardware logs and only introduces modest slowdowns. In particular, in a simulated 16-processor FMM system with software-only logging support, applications take only 10% longer to execute than if the system had hardware logs. We have also analyzed several design variations of software logging. One of the main conclusions is that it is very desirable to filter first stores.

References

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

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

[3] 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.

[4] K. Ekanadham, B.-H. Lim, P. Pattnaik, and M. Snir. PRISM: An Integrated Architecture for Scalable Shared Memory. In *Proc. 4th Intl. Symp. on High-Performance Computer Architecture*, February 1998.

[5] 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.

[6] 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.

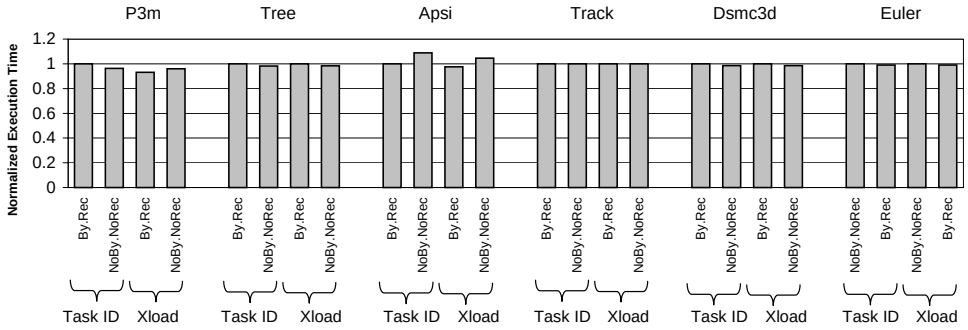

Figure 10. Comparing filtering with task IDs to filtering with extended loads.

[7] 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.

[8] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. Tradeoffs in Buffering Memory State for Thread-level Speculation in Multiprocessors. In *Proc. Intl. Symp. on High-Performance Computer Architecture*, pages 191–202, February 2003.

[9] 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.

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

[11] E. Hagersten and M. Koster. WildFire – A Scalable Path for SMPs. In *Proc. 5th Intl. Symp. on High-Performance Computer Architecture*, January 1999.

[12] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *8th Intl. Conf. on Architectural Support for Prog. Lang. and OS*, 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, Special Issue on Multithreaded Architectures*, 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] A. Nowatzyk et al. The S3.mp Scalable Shared Memory Multiprocessor. In *Proc. 1995 Intl. Conf. on Parallel Processing*, pages I1–I10, August 1995.

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

[23] J. Tsai et al. The Superthreaded Processor Architecture. *IEEE Trans. on Computers, Special Issue on Multithreaded Architectures*, 48(9):881–902, September 1999.

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

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

Appendix A: Recovery and Retrieval

Recovery: Undo and Re-Run Tasks

When an out-of-order RAW occurs, we recover by undoing the memory system changes performed by the task that read prematurely and all its successors. This is done by a software exception handler running on each of the processors that have executed one of such tasks.

Undoing tasks involves two steps. First, the processor writes back the dirty data in its caches to memory. Second, the processor traverses its local log sequentially and in order from younger to older Task ID, reading the data in the log records created by the tasks to be undone. For each record, the processor reads the data, task ID, and virtual address. Then, it writes the data back to memory, to its virtual address, while including the task ID in the message. The process described proceeds in parallel across several processors.

The home memory may receive several versions of the same variable, which it distinguishes using the task ID in the message. However, the memory only accepts versions whose task ID is younger than or equal to the version that it currently has, and older than or equal to the commit point. For this, the memory uses the same hardware support that it ordinarily uses in non-recovery mode to pick up the youngest displaced version [8]. Some important details can be found in [24]. At the end of this process, the memory is in the correct state and the tasks can be re-run.

Retrieval: Dig Out the Correct Version

Sometimes a load by a consumer task cannot be satisfied because, on the processor where the producer task ran, a third task has overwritten the variable and pushed the desired version into the log. In this case, we run a software exception handler. The handler knows which processor logs may have the version it wants. It also knows that the task ID of the version is both younger than a certain lower bound and older than the consumer task's own ID. Consequently, the handler sequentially reads each of these logs. Each log is traversed sequentially and in order from younger to older Task ID until that lower-bound ID. The youngest version found in this range is selected.