# Software Logging under Speculative Parallelization

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

Universidad de Zaragoza, Spain
{garzaran,victor}@posta.unizar.es
http://www.cps.unizar.es/gaz

[†]University of Illinois
{prvulovi, torrellas}@cs.uiuc.edu
http://iacoma.cs.uiuc.edu

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

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

**Summary.** Speculative parallelization aggressively runs hard-to-analyze codes in parallel. Speculative tasks generate unsafe state, which is typically buffered in caches. Often, a cache may have to buffer the state of several tasks and, as a result, it may have to hold multiple versions of the same variable. Modifying the cache to hold such multiple versions adds complexity and may increase the hit time. It is better to use logging, where the cache only stores the last versions of variables while the log keeps the older ones. Logging also helps to reduce the size of the speculative state to be retained in caches.

This paper explores efficient software-only logging for speculative parallelization. We show that such an approach is very attractive for programs with tasks that create multiple versions of the same variable. Using simulations of a 16-processor CC-NUMA, we show that the execution time of such programs on a system with software logging is on average 36% shorter than on a system where caches can only hold a single version of any given variable. Furthermore, execution takes only 10% longer than in a system with hardware support for logging.

## 1.1 Introduction

Speculative thread-level parallelization attempts to extract parallelism from hard-to-analyze codes like those with pointers, indirectly-indexed structures, interprocedural dependences, or input-dependent patterns. The idea is to break the code into tasks and speculatively run some of them in parallel. A combination of software and hardware support tracks memory accesses at run time and, if a dependence violation occurs, the state is repaired and parallel execution resumes. Many different schemes have been proposed, ranging from software-only [9, 16, 17] to hardware-based [4, 7, 8, 10, 12, 13, 14, 15, 18, 20, 21, 23, 25], and targeting small systems [7, 8, 10, 13, 14, 18, 21] or scalable ones [4, 9, 15, 16, 17, 20, 23, 25].

As a speculative task runs, it generates state that cannot be merged with main memory because it is unsafe. Different schemes handle the buffering of speculative state differently. In some schemes, each task uses its own private range of storage addresses [9, 16, 17, 24]. In most schemes, however, tasks buffer the speculative state dynamically in caches [4, 8, 13, 20], write buffers [10, 21], or special buffers [7, 15]. If the cache or buffer overflows due to conflicts or insufficient capacity, the processor has to stall or squash the task.

The size of the speculative state to be buffered by a processor depends on the working set size of individual tasks and on the load imbalance between tasks. Indeed, task load imbalance may force a processor to buffer the state of several speculative tasks at a time, which increases the overall speculative state size. Many of these tasks may be finished, but are still speculative because a predecessor task is still running.

Buffering the speculative state of several tasks at a time may be challenging. Specifically, such state may contain individual variables that have been written by several tasks. Such variables are common in applications that have quasi-privatization access patterns. In this case, the buffer must be organized to hold several speculative versions of the same variable. Furthermore, for these variables, it is preferable to keep the last version more handy, since it is more likely to be needed next.

Past work on speculative parallelization takes different approaches to handle this multi-version problem. Many schemes do not address this issue. Thus, it must be assumed that the processor stalls or squashes the task before creating a second local version of a speculative variable. Other schemes propose redesigning the cache to hold multiple speculative versions of the variable, e.g. in different ways of a set-associative cache [4, 19]. This approach complicates cache operation and may increase cache hit time. Finally, other schemes propose to store only last versions in the cache and automatically displace older speculative versions to a hardware-managed undo log [23, 25]. With logging, before a task overwrites a speculative version generated by a previous task, the hardware saves the version in the log. Logging is attractive because it reduces the size of the speculative state to be retained in caches and keeps last versions more handy. However, this solution requires non-trivial hardware support.

Since logging has advantages but also has a noticeable hardware cost, this paper explores buffering multiple versions through efficient software-only logging. Logs are declared as plain user data structures and are managed in software. We present one efficient implementation. Simulations of a 16-processor CC-NUMA show that software logging is very attractive for programs with tasks that create multiple versions of the same variable. The execution time of such programs on a system with software logging is on average 36% shorter than on a system where caches can only hold a single version of any given variable. Furthermore, execution takes only 10% longer than in a system with hardware support for logging.

This paper is organized as follows: Section 1.2 introduces speculative parallelization and versioning; Section 1.3 introduces the speculation protocol used; Section 1.4 presents efficient software logging; Section 1.5 discusses our evaluation environment; and Section 1.6 presents the evaluation.

## 1.2 Speculative Parallelization and Versioning

### 1.2.1 Basics of Speculative Parallelization

When several tasks run under speculative parallelization, they have a relative order imposed by the sequential code they come from. Consequently, we use the terms predecessor and successor tasks. If we give increasing IDs to successor tasks, the lowest-ID task still running is called non-speculative, while its successors are called speculative.

The set of variables that a speculative task writes is typically kept buffered away in the cache [4, 8, 13, 20], write buffer [10, 21], or special buffers [7, 15]. These variables cannot be merged with main memory because they are unsafe. They are called the speculative state. Only when the task becomes non-speculative can its speculative state be merged with main memory.

When the non-speculative task finishes, it commits. Any state that it kept buffered can be merged with memory and the non-speculative status is passed to a successor task. When a speculative task finishes, it cannot commit. The processor on which it ran can start to execute another speculative task, but the cache has to be able to hold speculative state from the two (or more) uncommitted tasks. Thus, in order to distinguish which of these tasks produced a particular cached variable, we associate a task-ID field with each variable (or line) in the cache.

### 1.2.2 Multiple Local Speculative Versions

In some cases, the speculative tasks that share a given cache as a reservoir for their speculative state may try to generate multiple versions of the same variable. This occurs, for example, in codes with quasi-privatization access patterns. In this case, if we have a simple cache, we may decide to support only a single version of each variable and, when a second local version is about to be created, stall the processor or squash the task.

One alternative approach that has been proposed is to redesign the cache to hold multiple versions of the same variable at a time [4, 19]. The cache must be able to buffer several lines with the same address tag but different task-IDs. For example, Figure 1.1-(a) shows a cache with three versions of line

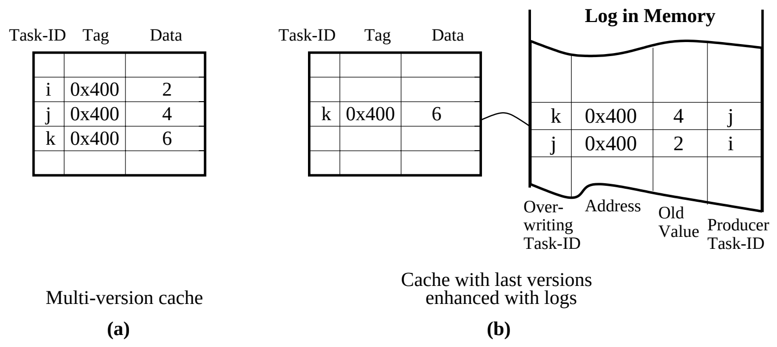

**Fig. 1.1.** Two ways of keeping multiple local speculative versions.

0x400 generated by task-IDs i, j, and k. These lines can go into different ways of the same set in a set-associative cache [4, 19].

Unfortunately, this approach adds complexity to the cache. In addition, the extra comparisons needed affect a sensitive timing path and may increase the cache hit latency. Moreover, since all the versions share the cache, the chance of line displacements due to capacity or conflict increases. The result may be lower performance, since existing schemes typically prevent the displacement of speculative versions to memory by either squashing the task or stalling the processor.

A final shortcoming of this approach is that it makes it equally hard to access any of the versions of a given variable. Instead, we would like to be able to access the *last* version of the variable faster. Such a version is the one generated by the youngest task that ran on the processor and wrote the variable. It is the version that will be needed to satisfy any subsequent load by this task or younger ones.

The older versions are much less likely to be accessed. For example, they may be accessed by a read request from an old task running on a second processor. Since we expect these events to be relatively infrequent, we could afford to make accesses to non-last versions a bit slower.

Our proposed approach is to keep *last* versions in the cache, and copy *non-last* versions to a software *log structure*, mapped to the virtual space of the application. A natural implementation of the log is a list of records that are placed in memory contiguously in real time. In general, a log record includes the previous version of the variable (before it is overwritten), its address, the producer task-ID, and the overwriting task-ID. Log records can be displaced from the cache and possibly even bypass it to minimize space contention with last versions. Figure 1.1-(b) shows a cache with its associated log organization in memory. Version k is the last one.

```
specul_par_do i
  do j = 1, i-1
    xdt(j) = ...
  enddo
  Compute L
  do j = 1, L
    ... = xdt(ind(j))
  enddo
enddo
```

**(a)** Bdna

```
specul_par_do i
  do j
    do k
      work(k) = work (f(i,j,k))
    enddo
    call foo(work(j))
  enddo
enddo
```

**(b)** Apsi

```
specul_par_do i
  while (sptr > 0) do
    ... = stack(sptr)
    do j
      if (cond)
        stack(sptr) = ...
        Compute sptr
    enddo
  enddo
enddo
```

**(c)** Tree

**Fig. 1.2.** Examples of non-analyzable loops that exhibit quasi-privatization access patterns. The outermost loop is speculatively run in parallel.

### 1.2.3 Application Behavior

There are many applications that require individual processors to buffer multiple speculative versions of the same variable. These applications tend to exhibit quasi-privatization access patterns. Under this pattern, tasks create new versions of a given variable without reading older versions. Of course, the pattern should not be fully analyzable since, otherwise, the compiler would have privatized the variable and the tasks could execute in parallel without speculation.

Figure 1.2 shows simplified loops taken from the applications used later in this paper that exhibit quasi-privatization access patterns. The arrays accessed with quasi-privatization patterns are *xdt()*, *work()*, and *stack()*. These loops are speculatively parallelized because the compiler cannot prove that iterations are independent.

Applications with these patterns have a higher tendency to require the buffering of multiple versions per processor if the tasks exhibit load imbalance. In this case, processors that execute short tasks accumulate many speculative versions. Note that the imbalance may be intrinsic to the application or can come from external causes that selectively delay the execution of some tasks. Such causes can be operating system interrupts or tasks from a different application grabbing a processor.

Table 1.1 estimates the weight of quasi-privatizable data and the load imbalance in our applications (discussed in Section 1.5.2) running on our simulated 16-processor architecture of Section 1.5.1. Column *Priv* shows how much data generated by task $i-1$ was speculatively written by task $i$ without reading it first. This quantity estimates the amount of quasi-privatizable data in task $i$. The table also shows the rest of the data speculatively written by task $i$ (*Other*), and the sum of the two items (*Total*). From the table, we see that tasks in *P3m*, *Tree*, *Apsi*, and *Bdna* write a lot of quasi-privatizable data, both in absolute and relative terms. In *Dsmc3d* and *Track*, they do not.

| Appl | Spec Written Footprint per Task (Kbytes) | | | # Uncommitted Tasks per Proc | | Need Multiple-Version Buffer? |
|---|---|---|---|---|---|---|
| | Priv | Other | Total | Maximum | Average | |
| *P3m* | 1.5 | 0.2 | 1.7 | 100 | 50.0 | Yes |
| *Tree* | 0.9 | 0.0 | 0.9 | 8 | 1.5 | Yes |
| *Apsi* | 12.0 | 8.0 | 20.0 | 4 | 1.8 | Yes |
| *Bdna* | 23.5 | 0.2 | 23.7 | 4 | 1.6 | Yes |
| *Dsmcd3d* | 0.0 | 0.8 | 0.8 | 3 | 1.1 | No |
| *Track* | 0.0 | 2.3 | 2.3 | 4 | 1.3 | No |
| Average | 6.3 | 1.9 | 8.2 | 20.5 | 9.5 | – |

**Table 1.1.** Application characteristics that affect the need for individual processors to buffer multiple versions of the same variable. The data corresponds to 16-processor runs.

Columns 5-6 show the number of uncommitted tasks that keep their speculative state buffered per individual processor at a time. The table shows the maximum and average values for the execution of the applications. Large values in these two columns suggest load imbalance. These columns show that each processor may need to buffer the state of more than 1 speculative task at a time. Even if we redesigned a set-associative cache to keep each version of a given variable in a different bank, the cache would require too high an associativity in *P3m* and *Tree*. *Apsi* and *Bdna* are less imbalanced, although they can still require up to 4 versions of the same variable to be buffered per processor. Thus, we expect that *P3m*, *Tree*, *Apsi*, and *Bdna* will benefit from multiple-version buffering supports.

## 1.3 Speculation Protocol Used

In [23], speculative accesses are marked with special load and store instructions that trigger the protocol. In each node, the first speculative access to a shared data page prompts the OS to allocate a page of *local time stamps* in the local memory of the node. These time stamps will record, for each word, the ID of the youngest local task that writes the word (*PMaxW*), and the ID of the youngest local task that reads it without writing it first (*PMaxR1st*). The latter operation is also called *exposed load*. These local time stamps are needed by the protocol, and are automatically updated by dependence-detecting hardware with small overhead [23].

## 1.4 Efficient Software Logging

### 1.4.1 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 useless for retrieval or recovery (*Recycle*).

Figure 1.3 shows simple per-processor software structures that we use for logging. The log buffer is broken down into fixed-sized sectors that will be used to log individual tasks. The compiler estimates the size of the sectors and log buffer based on the number of writes in a task and the number of tasks per processor that are likely to be uncommitted at a time, respectively.

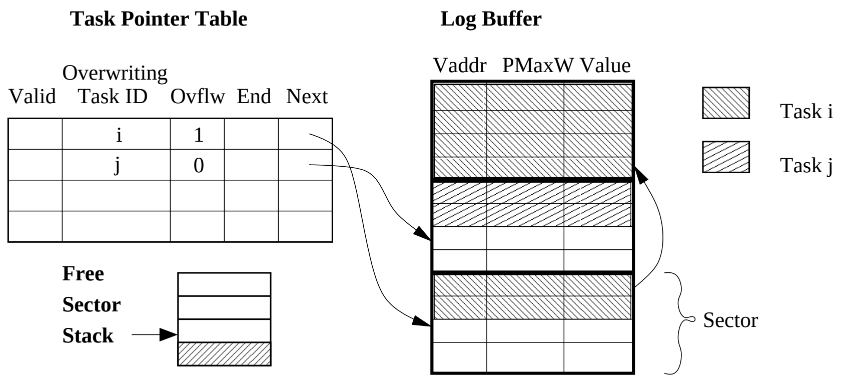

**Fig. 1.3.** Simple per-processor software structures that we use for logging.

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. Each entry in the Task Pointer Table has two pointer fields: Next that points to the next entry to fill, and End that points to 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. If the Free Sector Stack runs out of entries, we resize the Log Buffer and Stack accordingly.

**Insertion.** Insertion is the most overhead-sensitive operation since it occurs frequently. At compile time, the compiler instruments stores in the code with instructions to save a log record. As shown in Figure 1.3, a record includes the following information about the variable that is about to be updated: its virtual address (the only one the software knows), its value before the update, and its producer Task-ID (*PMaxW*). *PMaxW* is obtained from the local time-stamp page (Section 1.3). After the record is inserted at run time, the Next pointer is incremented. At the end of a task, all the records that it generated are in contiguous locations in one or more sectors, easily retrievable through the Task Pointer Table with the task ID.

**Recycle.** When a processor finishes a task, it tries to commit it [23]. Based on the resulting value of the commit point, it can identify which of the entries in its Task Pointer Table correspond to committed tasks. The data in those tasks' sectors will not be needed in the future and, therefore, the sectors can be recycled. Consequently, we invalidate the corresponding Task Pointer Table entries and return the sectors to the Free Sector Stack.

**Recovery** and **Retrieval.** Recovery occurs when we need to undo tasks after the detection of an out-of-order RAW dependence. Retrieval occurs when an in-order RAW dependence cannot simply be satisfied by the underlying coherence protocol because the requested version is not in a cache: another task running on the producer processor has overwritten the variable, pushing the desired version into the log. Since these two cases happen infrequently, we solve them with software exception handlers that access the logs.

### 1.4.2 Insertion Overhead

Inserting a record in the log of Figure 1.3 involves collecting the items to save, saving them in sequence using the *Next* pointer, and advancing the pointer. Figure 1.4 shows the MIPS assembly instructions added before every speculative store that we log. All memory accesses in the figure are non-speculative. Overall, 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 time stamp.

```
            ; r1 = upper limit of the sector. r5 = ID of the executing task
            ; r2 = address in memory to insert the log record.
            ; offset(r3) = address of the variable to update.
            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_ts  r4, offset(r3)             ; load the 16-bit PMaxW time stamp
            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_ts  r5, offset(r3)             ; update cached PMaxW
```

**Fig. 1.4.** Instructions added before an instrumented speculative store.

Figure 1.4 shows two special instructions, namely *load half-word time stamp (lh_ts)* and *store half-word time stamp (sh_ts)*. These are special instructions that load the 16-bit *PMaxW* local time stamp of the variable, and update it (the cached copy only), respectively. *lh_ts* loads the time stamp so that it can be saved in the log. *sh_ts* updates the cached copy with the ID of the executing task. This is done to prevent the cached copy from becoming stale. The reason is that the time stamp pages in memory are read and updated in hardware by the speculation protocol as it tries to detect dependence violations (Section 1.3). They cannot be updated in software.

We can reduce the overhead of the instrumentation by noting that the log only needs to save the value overwritten by the *first store* to the variable in the task. Consequently, for variables accessed with speculative accesses, we can modify the instrumentation in Figure 1.4 to dynamically test whether or not a store is a first store in the task, and log only if it is. Such testing can be done by comparing the *PMaxW* of the variable with the ID of the executing task. If they are the same, the store is not a first store. A detailed discussion of this topic is beyond the scope of this paper.

### 1.4.3 Alternative: Hardware-Only Logging

In the evaluation section, we will compare our software logging system to a hardware-only implementation of logging described in [25]. The latter uses a logging module embedded in the directory controller of each node. Log record insertion is done in the background with no overhead visible to the program. Similarly, recycling has practically no overhead. The log is kept in memory, thereby avoiding cache pollution.

## 1.5 Evaluation Methodology

### 1.5.1 Simulation Environment

We use an execution-driven simulation system based on MINT [22] to model in detail a CC-NUMA with 16 nodes. Each node contains a fraction of the shared memory and directory, as well as a 4-issue dynamic superscalar. The processor has a 32-entry instruction window and 4 Int, 2 FP, and 2 Ld/St units. It supports 8 pending loads and 16 stores. It also has a 2K-entry BTB with 2-bit saturating counters. Each node has a 2-way 32-Kbyte L1 D-cache and a 4-way 2-Mbyte L2, both with 64-byte lines and a write-back policy. Contention is accurately modeled. 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 that is 2 and 3 protocol hops away are 1, 12, 60, 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 virtually impossible to optimize shared data allocation at compile time. Private data are allocated locally.

For speculation, we use the protocol of Section 1.3. 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. In addition, a processor that allocates a page of time stamps is penalized with 4,000 cycles.

### 1.5.2 Workload

| Appl | Non-Analyzable Sections (Loops) | % of Tseq | # of Invoc | Iters per Invoc | Instruc per Iter |
|---|---|---|---|---|---|
| *P3m* | *pp_do100* | 56.5 | 1 | 97336 | 69165 |
| *Tree* | *accel_do10* | 79.1 | 41 | 1024 | 28746 |
| *Apsi* | *run_do[20,30,40,50,60,100]* | 29.3 | 900 | 63 | 102639 |
| *Bdna* | *actfor_do240* | 44.2 | 1 | 1499 | 103339 |
| *Dsmc3d* | *move3_goto100* | 41.2 | 80 | 46777 | 5442 |
| *Track* | *nlfilt_do300* | 58.1 | 56 | 502 | 5577 |
| Average | | 51.4 | 180 | 24533 | 52484 |

**Table 1.2.** Application characteristics. 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. Finally, in *Dsmc3d*, the data corresponds to unrolling the loop 15 times.

We use a set of scientific applications where a large fraction of the code is not analyzable by a parallelizing compiler. These applications are: *Apsi* from SPECfp2000 [11], *Track* and *Bdna* from Perfect [2], *Dsmc3d* from HPF-2 [5], *P3m* from NCSA, and *Tree* from Univ. of Hawaii [1]. We use the Polaris parallelizing compiler [3] 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 doubly-subscripted accesses to arrays. The code also has sections that have complex control flow, with conditionals that depend on array values and jump to code sections that modify the same or other arrays. In these sections, Polaris marks the speculative references, which will trigger speculation protocol actions. Polaris also identifies store instructions that may need to be logged, and we instrument them according to Section 1.4.

Table 1.2 shows the non-analyzable sections in each application. These sections are loops. The table lists the weight of these loops relative to the total *sequential* execution time of the application (%Tseq), with the I/O excluded. This value, which is obtained on a single-processor Sun Ultra 5 workstation, is on average 51.4%. The table also shows the number of invocations of these loops during program execution, the average number of iterations per invocation, and the average number of instructions per iteration. Note that all the data presented in the evaluation, including speedups, refer only to the code sections in the table.

## 1.6 Evaluation

We compare multiprocessors that support no logging, software logging, or hardware logging. Under no logging, a node can only hold in its cache hierarchy a single version for each variable; if the processor is about to overwrite a local version produced by an uncommitted task, the processor stalls. For software logging, we use our scheme of Sections 1.4.1 and 1.4.2. Finally, for hardware logging, we use the support of Section 1.4.3.

Figure 1.5 compares the execution time of these three systems, called *NoLog*, *Sw*, and *Hw*, respectively. They run on 16 processors. For each application, the bars are normalized to *NoLog* and broken down into execution of instructions (*Useful*), waiting on data, control, and structural pipeline hazards (*Hazard*), synchronization (*Sync*), waiting on data from the memory system (*Memory*), and stall when attempting to overwrite an uncommitted version in *NoLog* (*Stall*). A sixth category, measuring the execution of software handlers for data recovery and retrieval, is too small to be seen. 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 on the local memory of the single active processor.

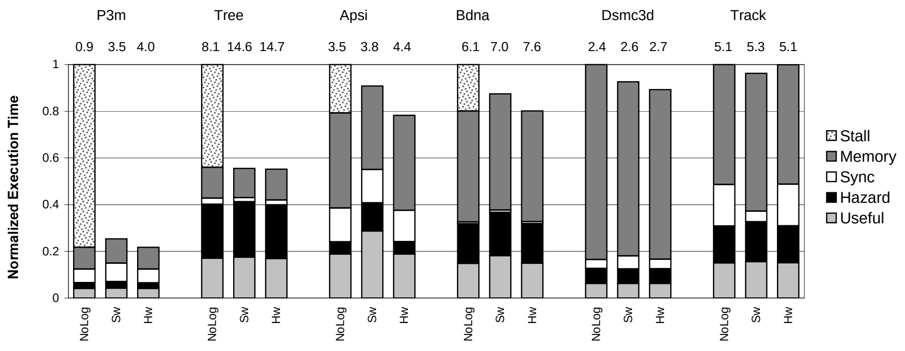

**Fig. 1.5.** Execution time on a 16-node multiprocessor with different logging supports.

A comparison between *NoLog* and *Sw* reveals the benefits of software logging. With software logging, processors do not stall when they overwrite local uncommitted versions. Thus, *Stall* disappears. However, software logging introduces extra instructions and memory system accesses to generate and maintain the log. As a result, it tends to have higher *Useful* and *Memory* times. Indirectly, the other times (*Hazard* and *Sync*) may also increase.

To understand these results, note that logging is most beneficial in applications that have both quasi-privatization access patterns and load imbalance. *P3m*, *Tree*, *Apsi*, and *Bdna* have quasi-privatization patterns and, of them, *P3m* and *Tree* have the largest imbalance. These observations are consistent with Figure 1.5. The figure shows that *P3m*, *Tree* and, to a lesser extent, *Apsi* and *Bdna* have *Stall* under *NoLog*. *Sw* removes all *Stall* and speeds up these applications, especially *P3m* and *Tree*. For *Dsmc3d* and *Track*, the difference between *NoLog* and *Sw* is an indirect effect of the different data layouts, which cause different cache conflicts, and the different execution timings, which result in different dependence violations found at run time. Overall, software logging is effective: *Sw* is on average 36% faster than *NoLog* for the four applications with quasi-privatization patterns. If we take the average of all the applications, the speedup of the 16-processor execution increases from 4.3 under *NoLog* to 6.1 under *Sw*.

The *Hw* system also eliminates the *Stall* time like the *Sw* system. Furthermore, it induces negligible overheads. The cost, of course, is special hardware support. Comparing *Sw* to *Hw*, we see the overhead of software logging. From the figure, we see that this overhead is very modest. Indeed, for the four applications with quasi-privatization patterns, the average overhead is only 9% of the *Sw* execution time. Therefore, we conclude that software logging is efficient as well as effective.

## 1.7 Related Work

Different schemes handle speculative state buffering differently. For instance, some schemes do not buffer speculative state because they do not allow on-the-fly repair on a violation [9, 16, 24]. If a violation occurs, the state rolls back to the beginning of the speculative section. The schemes that support on-the-fly repair typically buffer the state in caches [4, 8, 13, 20], write buffers [10, 21], or special buffers [7, 15]. In some schemes, a new task cannot start on a processor until the task that previously ran there commits [7, 8, 13, 14, 21]. In this case, there is no need to support speculative state from multiple tasks in the buffer.

Several schemes support several uncommitted tasks on a processor, including Run-Time Speculation (RTS) [25], Hydra [10], Stampede [20], and MDT [4]. In RTS, they handle multiple versions with hardware logs. In Hydra, a new buffer is allocated when a new task starts to execute. The applications that they use only need up to 2 Kbytes of buffering storage. In Stampede, they suggest either to exploit cache associativity to hold multiple versions, or to stall or squash the task [19]. However, since their applications do not need support for multiple writers [20], they do not evaluate this issue. In MDT, they also exploit cache associativity, aided by a victim cache, to keep multiple versions [4].

Finally, logging is also used in the schemes presented in SUDS [6] and RTS [25]. RTS uses hardware logs as described in Section 1.4.3. The SUDS system, proposed for RAW, uses software logs that only keep a single version per variable [6].

## 1.8 Conclusion

A good solution to buffer speculative state with multi-version variables is to enhance a cache hierarchy with logs. In this paper we show that software logging is inexpensive and delivers high performance for applications with quasi-privatization patterns and load imbalance. Using simulations of a 16-processor CC-NUMA, we show that the execution time of such applications on a system with software logging is on average 36% shorter than on a system where caches can only hold a single version of any given variable. Furthermore, execution takes only 10% longer than in a system with hardware support for logging.

## Acknowledgments

This work was supported in part by the National Science Foundation under grants CCR-9970488, EIA-9975018, EIA-0081307, and EIA-0072102; by the CICYT of Spain under grant TIC 2001-0995-C02-02; and by gifts from IBM and Intel.

## References

1. J. E. Barnes. ftp://hubble.ifa.hawaii.edu/pub/barnes/treecode/. *University of Hawaii*, 1994.
2. M. Berry et al. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.
3. W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Advanced Program Restructuring for High-Performance Computers with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.
4. M. Cintra, J. F. Martinez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 13–24, June 2000.

5. I. Duff, R. Schreiber, and P. Havlak. HPF-2 Scope of Activities and Motivating Applications. Technical Report CRPC-TR94492, Rice University, November 1994.
6. M. Frank, W. Lee, and S. Amarasinghe. A Software Framework for Supporting General Purpose Applications on Raw Computation Fabrics. Technical Report MIT/LCS Technical Memo 619, July 2001.
7. M. Franklin and G. S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.
8. S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 195–205, February 1998.
9. M. Gupta and R. Nim. Techniques for Speculative Run-Time Parallelization of Loops. In *Proceedings of Supercomputing 1998*, pages 1–12, November 1998.
10. L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, October 1998.
11. J.L. Henning. SPEC CPU2000: Measuring Performance in the New Millenium. *IEEE Computer*, 33(7):28–35, July 2000.
12. T. Knight. An Architecture for Mostly Functional Languages. In *ACM Lisp and Functional Programming Conference*, pages 500–519, August 1986.
13. 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.
14. P. Marcuello and A. González. Clustered Speculative Multithreaded Processors. In *Proc. of the 1999 Int'l Conference on Supercomputing (ICS'99)*, pages 365–372, June 1999.
15. M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas. Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 204–215, July 2001.
16. L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In *Proceedings of the SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 218–232, June 1995.
17. P. Rundberg and P. Stenström. Low-Cost Thread-Level Data Dependence Speculation on Multiprocessors. In *Fourth Workshop on Multithreaded Execution, Architecture and Compilation*, December 2000.
18. G. S. Sohi, S. Breach, and S. Vajapeyam. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.
19. J.G. Steffan, C. B. Colohan, and T. C. Mowry. Architectural Support for Thread-Level Data Speculation. Technical report, CMU-CS-97-188, School of Computer Science, Carnegie Mellon University, November 1997.
20. J.G. Steffan, C.B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 1–12, June 2000.
21. J.Y. Tsai, J. Huang, C. Amlo, D. Lilja, and P.C.Yew. The Superthreaded Processor Architecture. *IEEE Trans. on Computers, Special Issue on Multithreaded Architectures*, 48(9):881–902, September 1999.
22. J. Veenstra and R. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the Second International*

*Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'94)*, pages 201–207, January 1994.

23. Y. Zhang. Hardware for Speculative Run-Time Parallelization in DSM Multiprocessors. Ph.D. Thesis, University of Illinois at Urbana-Champaign, Department of Electrical and Computer Engineering, May 1999.
24. Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 162–174, February 1998.
25. Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Parallelization of Partially-Parallel Loops in DSM Multiprocessors. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 135–139, January 1999.