Speculative Versioning Cache

Sridhar Gopal[†] gsri@cs.wisc.edu

T.N.Vijaykumar[*] vijay@ecn.purdue.edu

James E. Smith[‡] jes@ece.wisc.edu

Gurindar S. Sohi[†] sohi@cs.wisc.edu

[†]Computer Sciences Department University of Wisconsin

[*]School of Electrical and Computer Engineering Purdue University

[‡]Department of Electrical and Computer Engineering University of Wisconsin

Abstract

Dependences among loads and stores whose addresses are unknown hinder the extraction of instruction level parallelism during the execution of a sequential program. Such ambiguous memory dependences can be overcome by memory dependence speculation which enables a load or store to be speculatively executed before the addresses of all preceding loads and stores are known. Furthermore, multiple speculative stores to a memory location create multiple speculative versions of the location. Program order among the speculative versions must be tracked to maintain sequential semantics. A previously proposed approach, the Address Resolution Buffer(ARB) uses a centralized buffer to support speculative versions. Our proposal, called the Speculative Versioning Cache(SVC), uses distributed caches to eliminate the latency and bandwidth problems of the ARB. The SVC conceptually unifies cache coherence and speculative versioning by using an organization similar to snooping bus-based coherent caches. A preliminary evaluation for the Multiscalar architecture shows that hit latency is an important factor affecting performance, and private cache solutions trade-off hit rate for hit latency.

1. Introduction

Modern microprocessors extract instruction level parallelism (ILP) from sequential programs by issuing instructions from an active instruction window. Data dependences among instructions, and not the original program order, determine when an instruction may be issued from the window. Dependences involving register data are detected easily because register designators are completely specified within instructions. However, dependences involving memory data (e.g. between a load and a store or two stores) are ambiguous until the memory addresses are computed.

A straightforward solution to the problem of ambiguous memory dependences is to issue loads and stores only after their addresses are determined. Furthermore, a store is not allowed to complete and commit its result to memory until all preceding instructions are known to be free of exceptions. Each such store to a memory location creates a *speculative version* of that location. These speculative versions are held in buffers until they can be committed. Multiple speculative stores to the same location create multiple versions of the location. To improve performance, loads are allowed to bypass buffered stores, as long as they are to different addresses. If a load is to the same address as a buffered store, it can use data bypassed from the store when the data becomes available. An important constraint of this approach is that a load instruction *cannot* be issued until the addresses of all the preceding stores are determined. This approach may diminish ILP unnecessarily, especially in the common case where the load is not dependent on preceding stores.

More aggressive uniprocessor implementations issue load instructions as soon as their addresses are known, even if the addresses of all previous stores may not be known. These implementations employ memory dependence speculation [8] and predict that a load does not depend on previous stores. Furthermore, one can also envision issuing and computing store addresses out of order. Such memory dependence speculation enables higher levels of ILP, but more advanced mechanisms are needed to support this speculation. These aggressive uniprocessors dispatch instructions from a single instruction stream, and issue load and store instructions from a common set of hardware buffers (e.g. reservation stations). Using a common set of buffers allows the hardware to maintain program order of the loads and stores via simple queue mechanisms, coupled with address comparison logic. The presence of such queues provides support for a simple form of *speculative versioning*.

However, proposed next generation processor designs use replicated processing units that dispatch and/or issue instructions in a distributed manner. These future approaches partition the instruction stream into sub streams called tasks [11] or traces [10]. Higher level instruction control units distribute the tasks to the processors for execution and, the processors execute the instructions within each task leading to a hierarchical execution model. Proposed next generation multiprocessors [9, 12] that provide hardware support for dependence speculation also use such execution models. A hierarchical execution model naturally leads to memory address streams with a similar hierarchical structure. In particular, each individual task generates its own address stream, which can be properly ordered (disambiguated) within the processor that generates it, and at the higher level, the multiple address streams produced by the processors must also be properly ordered. It is more challenging to support speculative versioning for this execution model than a superscalar execution model because a processor executes loads and stores without knowing those executed by other processors.

The Address Resolution Buffer [3] (ARB) provides speculative versioning support for such hierarchical execution models. Each entry in the ARB buffers all versions of the same memory location. However, there are two significant performance limitations of the ARB:

1. The ARB is a single shared buffer connected to the multiple processors and hence, every load and store incurs the latency of the interconnection network. Also, the ARB has to provide sufficient bandwidth for all the processors.

2. When a task completes all its instructions, the ARB commits its speculative state into the architected storage (or copies all the versions created by this task to the data cache). Such write backs generate bursty traffic and can increase the time to commit a task, which delays the issue of new task to that processor and lowers the overall performance.

We propose a new solution for speculative versioning called the Speculative Versioning Cache [2, 5] (SVC), for hierarchical execution models. The SVC comprises a private cache for each processor, and the system is organized similar to a snooping bus-based cache coherent Symmetric Multiprocessor (SMP). Memory references that hit in the private cache do not use the bus as in an SMP. Task commits do *not* write back speculative versions *en masse*. Each cache line is individually handled when it is accessed the next time.

Section 2 introduces the hierarchical execution model briefly and identifies the issues in providing support for speculative versioning for such execution models. Section 3 presents the SVC as a progression of designs to ease understanding. Section 4 gives a preliminary performance evaluation of the SVC to highlight the importance of a private cache solution for speculative versioning. We derive conclusions in section 5.

2. Speculative versioning

First, we discuss the issues involved in providing support for speculative versioning for current generation processors. Second, we describe the hierarchical execution model used by the proposed next generation processors. Third, we discuss the issues in providing support for speculative versioning for this execution model and use examples to illustrate them. Finally, we present similarities between multiprocessor cache coherence and speculative versioning for the hierarchical execution model and use this unification to motivate our new design, the speculative versioning cache.

Speculative versioning involves tracking the program order among the multiple buffered versions of a location to guarantee the following sequential program semantics:

- A load must *eventually* read the value created by the most recent store to the same location. This requires that (i) the load must be squashed and re-executed if it executes before the store and incorrectly reads the previous version and, (ii) all stores (to the same location) that follow the load in program order must be buffered until the load is executed.

- A memory location must *eventually* have the correct version independent of the order of the creation of the versions. Consequently, the speculative versions of a location must be committed to the architected storage in program order.

2.1. Hierarchical execution model

In this execution model, the dynamic instruction stream of a program is partitioned into fragments called *tasks*. These tasks form a *sequence* corresponding to their order in the dynamic instruction stream. A higher level control unit *predicts* the next task in the sequence and assigns it for execution to a free processor. Each processor executes the instructions in the task assigned to it and buffers the speculative state created by the task. The Wisconsin Multiscalar [11] is an example architecture that uses the hierarchical execution model.

When a task misprediction is detected, the speculative state of all the tasks in the sequence including and after the incorrectly predicted task are invalidated[1] and the corresponding processors are freed. This is called a *task squash*. The correct tasks in the sequence are then assigned for execution. When a task prediction has been validated, it *commits* by copying the speculative buffered state to the architected storage. Tasks commit one by one in the order of the sequence. Once a task commits, its processor is free to execute a new task. Since the tasks commit in program order, tasks are assigned to the processors in program order.

[1] An alternative model for recovery invalidates only the dependent

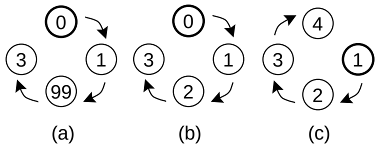

Figure 1: Task commits and squashes: example.

Figure 1 illustrates task commits and task squashes. Initially, tasks 0, 1, 99 and 3 are predicted and speculatively executed in parallel by the four processors as shown in Figure 1(a). When the misprediction of task 99 is detected, tasks 99 and 3 are squashed and their buffered states are invalidated. New tasks 2 and 3 are then executed by the processors as show in Figure 1(b). Tasks that are currently executing are said to be *active*. When task 0 completes execution, the corresponding processor is freed and task 4 is assigned for execution as shown in Figure 1(c). The program order, represented by the sequence among the tasks, enforces an *implicit* total order among the processors; the arrows show this order. When the tasks are speculatively executed in parallel, the multiple speculative load/store streams from the processors are merged in arbitrary order. Providing support for speculative versioning for such execution models requires mechanisms that establish program order among these streams. The following subsections outline how the order is established using the sequence among the tasks.

2.1.1. Loads A task executes a load as soon as its address is available, speculating that stores from previous tasks in the sequence do not write to the same location. The closest previous version of the location is supplied to the load; this version could have been created either by the same task or by a previous task. A load that is supplied a version from a previous task is recorded to indicate a use before a potential definition. If such a definition (a store to the same location from a previous task) occurs, the load was supplied with an incorrect version and memory dependence was violated.

2.1.2. Stores When a task executes a store to a memory location, it is communicated to all later active tasks in the sequence[2]. When a task receives a new version of a location from a previous task, it squashes if a use before definition is recorded for that location — a memory dependence violation is detected. All tasks after the squashed task are also squashed as on a task misprediction (simple squash model).

2.1.3. Task commits and squashes The oldest active task is non-speculative and can commit its speculative memory state (versions created by stores from this task) to architected storage. Committing a version involves logically copying the versions from the speculative buffers to the architected storage (data cache). As we assume the simple task squash model, the speculative state associated with a task is invalidated when it is squashed.

2.2. Examples for speculative versioning

Figure 2 illustrates the issues involved in speculative versioning using an example program and a sample execution of the program on a four processor hierarchical system. We use the same example in the later sections to explain the SVC design. Figure 2(a) shows the loads and stores in the example program and the task partitioning. Other instructions are not of direct relevance here. Figure 2(b) shows two snapshots of the memory system during a sample execution of the program. Each snapshot contains four boxes, one for each active task and shows the load or store that has been executed by the corresponding task. The program order among the instructions translates to a sequence among the tasks which imposes a total order among the processors executing them; *solid* arrowheads show the program order and *hollow* arrowheads show the execution time order in all the examples.

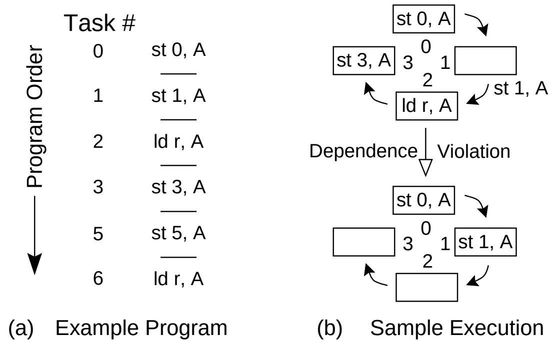

Figure 2: Speculative versioning example.

The first snapshot is taken just before task 1 executes a store to address A. Tasks 0 and 3 have already stored 0 and 3 to A and task 2 has executed a load to A. The load is supplied the version created and buffered by task 0. But, according to the original program, this load *must* be supplied the value 1 created by the store from task 1, i.e., the store to load dependence has been violated. This violation is detected when task 1 stores to address A and all the tasks including and after task 2 are squashed and re-executed. The second snapshot is taken after the tasks have been squashed and re-started.

2.3. Coherence and speculative versioning

The actions performed on memory accesses and task commits and squashes are summarized in Table 1. The functionality in this table requires the hardware to track the

chains of instructions by maintaining information at a finer granularity. This paper assumes the simpler model.

[2] In reality, the store has to be communicated only until the task that has created the next version, if any, of the location.

active tasks or processors that executed a load/store to a location and the order among the different copies/versions of this location. Cache coherent Symmetric MultiProcessors (SMP) use similar functionality to track the caches that have a copy of every memory location. SMPs, however, need not track the order among these copies since all the copies are of a single version.

Event	Actions
Load	Record use before definition by the task; supply the closest previous version.
Store	Communicate store to later tasks; later tasks look for memory dependence violations.
Commit	Write back buffered versions created by the task to main memory.
Squash	Invalidate buffered versions created by the task.

Table 1: Versioning: events and actions.

SMPs typically use snooping [4] to implement a Multiple Reader/Single Writer protocol, which uses a coherence directory that is a collection of sets, each of which tracks the sharers of a line. In a snooping bus based SMP, the directory is typically implemented in a distributed fashion comprising state bits associated with each cache line. On the other hand, the Speculative Versioning Cache (SVC) implements a Multiple Reader/Multiple Writer protocol that tracks copies of multiple speculative versions of each memory location. This protocol uses a version directory that maintains *ordered* sets for each line, each of which tracks the program order among the multiple speculative versions of a line. This ordered set or list, called the Version Ordering List (VOL), can be implemented in several different ways — the SVC, proposed in this paper, uses explicit pointers in each line to implement it as a linked list (like in SCI [1]). The following sections elaborate on a design that uses pointers in each cache line to maintain the VOL.

The *private* cache organization of the SVC makes it a feasible memory system for proposed next generation single chip multiprocessors that execute sequential programs on tightly coupled processors using automatic parallelization [9, 12]. Previously, ambiguous memory dependences limited the range of programs chosen for automatic parallelization. The SVC provides hardware support to overcome ambiguous memory dependences and enables more aggressive automatic parallelization of sequential programs.

3. SVC design

In this section, we present the Speculative Versioning Cache (SVC) as a progression of designs to ease understanding. Each design improves the performance over the previous one by tracking more information. We begin with a brief review of snooping bus-based cache coherence and then present a base SVC design which provides support for speculative versioning with minimal modifications to the cache coherence scheme. We then highlight the performance bottlenecks in the base design and introduce optimizations one by one in the rest of the designs.

3.1. Snooping bus based cache coherence

Figure 3 shows a 4-processor SMP with private L1 caches that uses a snooping bus to keep the caches consistent. Each cache line comprises an address tag that identifies the data that is cached, the data that is cached, and two bits (valid and store) representing the state of the line. The valid (V) bit is set if the line is valid. The store (S) or dirty bit is set when a processor stores to the line.

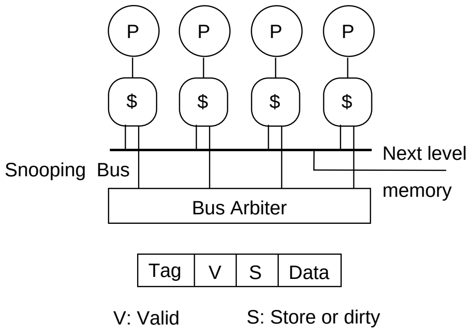

Figure 3: SMP coherent cache.

A cache line is in one of three states: Invalid, Clean and Dirty. A request (load or store) from a processor to its L1 cache *hits* if a valid line with the requested tag is in an appropriate state; otherwise, it *misses*. Cache misses issue bus requests while cache hits do not. More specifically, a load from a clean or dirty line and a store to a dirty line result in cache hits. Otherwise, the load(store) misses and the cache issues a *BusRead*(*BusWrite*) request. The L1 caches and the next level memory snoop the bus on every request. If a cache has a valid line with the requested tag, it issues an appropriate response according to a *coherence protocol*. A store to a clean line misses and the cache issues a *BusWrite* request. An invalidation-based coherence protocol invalidates copies of this line in all other caches, if any. This protocol allows a dirty line to be present in only one of the caches. However, a clean line can be present in multiple caches simultaneously. The cache with the dirty line supplies the data on a *BusRead* request. A cache issues a *BusWback* request to cast out a dirty line on a replacement. This simple protocol can be extended by adding an *exclusive* bit to the state of each line to cut down traffic on the shared bus. If a cache line has the exclusive bit set, then it has the only valid copy of the line and can perform a store to that line locally. The SVC designs we discuss in the following sections also use an invalidation-based protocol.

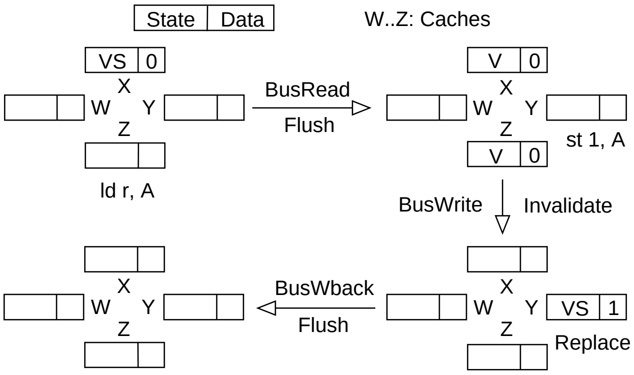

Figure 4: Cache coherence example.

Figure 4 shows snapshots of the cache lines with tag or address A in an SMP with four processors, W, X, Y, and Z. The state of the cache line is shown in a box corresponding to that cache. An empty box corresponding to a cache represents that the line is not present in that cache. The first snapshot is taken before processor Z issues a load from A and misses in its private cache. The cache issues a *BusRead* request and cache X supplies the data on the bus. The second snapshot shows the final state of the lines; they are clean. Later, processor Y issues a *BusWrite* request to perform a store to A. The clean copies in caches X and Z are invalidated and the third snapshot shows the final state. Now, if cache Y chooses to replace this line, it casts out the line to memory by issuing a *BusWback* request; the final state is shown in the fourth snapshot; only the next level memory contains a valid copy of the line.

3.2. Base SVC design

The organization of the private L1 caches in the SVC design is shown in Figure 5; all the SVC designs use the same organization. The base design minimally modifies the memory system of the snooping bus-based cache coherent SMP to support speculative versioning for processors based on the hierarchical execution model. We assume that memory dependences among loads and stores executed by an individual processor are ensured by a conventional load-store queue; our design guarantees program order among loads and stores from different processors. The base design also assumes that the cache line size is one word; a later design relaxes this assumption. First, we introduce the modifications to the SMP coherent cache, and then discuss how the individual operations listed in Table 1 are performed.

1. Each cache line maintains an extra state bit called the load (L) bit, as shown in Figure 6. The L bit is set when a task loads from a line before storing to the line — a potential violation of memory dependence in case a previous task stores to the same line.

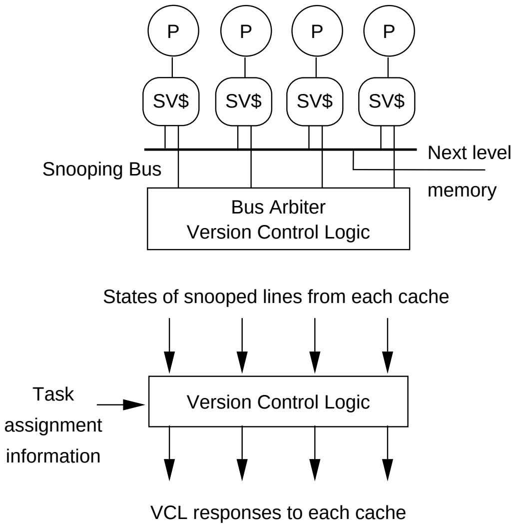

Figure 5: Speculative versioning cache.

Figure 6: Base SVC design: structure of a line.

2. Each cache line maintains a pointer that identifies the processor (or L1 cache) that has the next copy/version, if any, in the Version Ordering List (VOL) for that line. Thus, the VOL for a line is stored in a distributed fashion among the private L1 cache lines. It is important to note that the pointer identifies a processor rather than a task. Storing the VOL explicitly in the cache lines using pointers may not be necessary for the base design. However, it is necessary to explicitly store the VOL for the advanced designs and we introduce it in the base design to ease the transition to the advanced designs.

3. The SVC uses combinational logic called the Version Control Logic (VCL) that provides support for speculative versioning using the VOL. A processor request that hits in the private L1 cache does not need to consult the VOL and hence does not issue a bus request; the VCL is also not used. Cache misses issue a bus request that is snooped by the L1 caches and the next level memory. The states of the requested line in each L1 cache and the VOL are supplied to the VCL. The VCL uses the bus request, the program order among the tasks, and the VOL to compute appropriate responses for each cache. Each cache line is updated based on its initial state, the bus request and the VCL response. A block diagram of the Version Control Logic is shown in Figure 5. For the base design, the VCL responses are similar to that of the dis-

ambiguation logic in the ARB [3]. The disambiguation logic searches for previous or succeeding stages in a line to execute a load or store, respectively.

3.2.1. Loads Loads are handled in the same way as in an SMP except that the L bit is set if the line was initially invalid. On a *BusRead* request, the VCL locates the closest previous version by searching the VOL in the reverse order beginning from the requestor; this version, if any, is supplied to the requestor. If a previous version is not buffered in any of the L1 caches, the next level memory supplies the data. Task assignment information is used to determine the position of the requestor in the VOL. The VCL can search the VOL in reverse order because it has the entire list available and the list is short.

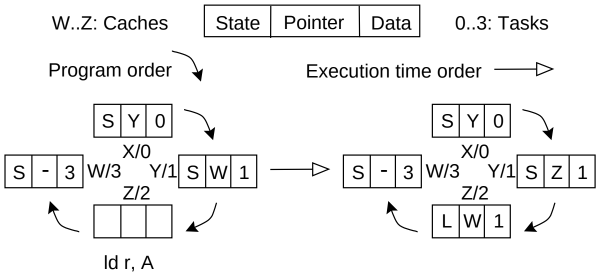

Figure 7: Base SVC design: example load.

We illustrate the load executed by task 2 to address A in the example program. Figure 7 shows two snapshots: one before the load executes and one after the load completes. Each box shows the line with tag or address A in an L1 cache (the valid bit is not explicitly shown). The number adjacent to a box gives a processor/cache identifier and a task identifier. The processor identifiers are used by the explicit pointers in each line to represent the VOL, whereas, the task identifiers serve only to ease the explanation of the examples. Task 2 executes a load that misses in cache Z and results in a bus request. The VCL locates cache Z in the VOL for address A using program order and then searches the VOL in the reverse order to find the correct version to supply, which is the version in cache Y (the version created by task 1).

3.2.2. Stores The SVC performs more operations on a store miss as compared to a cache coherent SMP. When a *BusWrite* request is issued on a store miss, the VCL sends invalidation responses to the caches beginning from the requestor's immediate successor (in task assignment order) to the cache that has the next version (including it, if it has the L bit set). This invalidation response allows for multiple versions of the same line to exist and also serves to detect memory dependence violations. A cache sends a task squash signal to its processor when it receives an invalidation response from the VCL and the L bit is set in the line.

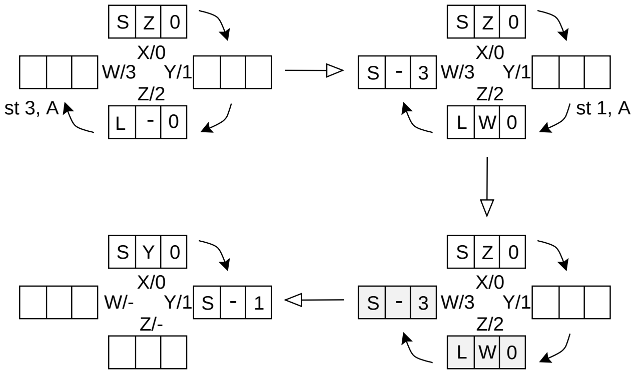

Figure 8: Base SVC design: example stores.

We illustrate the stores executed by tasks 1 and 3 in the example program. Figure 8 shows four snapshots of the cache lines with address A. The first snapshot is taken before task 3 executes a store that results in a *BusWrite* request. Since task 3 is the most recent in program order, the store by task 3 does not result in any invalidations. Note that a store to a line does not invalidate *all* other cache lines (unlike an SMP) to allow for multiple versions of the same line. The second snapshot is taken after the store from task 3 completes and before task 1 executes its store. Based on task assignment information, the VCL sends an invalidation response to each cache from the one after cache Y until the one before cache W, which has the next version of the line (cache W is not included since it does not have the L bit set) — VCL sends an invalidation response to cache Z. But, the load executed by task 2, which follows the store by task 1 in program order, has already executed. Cache Z detects a memory dependence violation since the L bit is set when it receives an invalidation response from the VCL. Tasks 2 and 3 are squashed as shown in the third snapshot by shaded boxes. The final snapshot is taken after the store by task 1 has completed.

3.2.3. Task commits and squashes The base SVC design handles task commits and squashes in a naive manner. When a processor commits a task, all dirty lines in its L1 cache are *immediately* written back to the next level memory and all other lines are invalidated. To write back all the dirty lines immediately, a list of the stores executed by the task is maintained by the processor. When a task is squashed, *all* lines in the corresponding cache are invalidated.

3.3. Base design performance drawbacks

The base design just described has two significant performance limitations that make it less desirable: (i) write backs performed when a processor commits a task lead to bursty bus traffic that may increase the time to commit the task and delay issuing a new task to that processor, (ii) clean lines are also invalidated when a task commits or squashes because the buffered versions could be stale for the new task allocated on the same processor; the correct version may be present in other caches. Consequently, *every* task begins execution with a *cold* L1 cache, increasing the bandwidth demand. The following advanced designs eliminate these problems by tracking additional information.

1. The first advanced design, the *ECS design* (section 3.5), makes task commits and squashes more efficient. To ease the understanding of this design, we first present an intermediate design, the EC design (section 3.4), that makes task commits efficient by distributing the write backs of dirty lines over time. Also, it retains read-only data in the L1 caches across task commits by careful book-keeping. However, it assumes that mispredictions do not occur. Then, we present the ECS design that extends the EC design to allow task squashes. Task squashes are as simple as in the base design, but are more efficient as they retain non-speculative data in the caches across task squashes.

2. The second advanced design (section 3.6) boosts the hit-rate of the ECS design by allowing requests to *snarf* [6] the bus to account for reference spreading. Snarfing involves copying the data supplied on a bus request issued by another processor in an attempt to combine bus requests indirectly.

3. The final design (section 3.7) is realistic and allows the size of a cache line to be more than one word.

3.4. Implementing efficient task commits (EC)

The EC design avoids expensive cache flushes on task commits by maintaining an extra state bit, called the commit bit, in each cache line. Task commits do not stall until all lines with speculative versions are written back. The EC design eliminates write back bursts on the bus during task commits. Also, no extra hardware is necessary to maintain a list of stores performed by each task. Further, the EC design improves cache utilization by keeping the L1 caches warm across tasks.

Figure 9: EC design: structure of a line.

The structure of a cache line in the EC design is shown in Figure 9. When a processor commits a task, the C bit is set in *all* its cache lines. This operation is entirely local to the L1 cache and does not issue a bus request. A dirty committed line is written back, if necessary, when it is accessed the next time either on a processor request or on a bus request. Therefore, committed versions could remain in the caches until much later in time since the task that created the version committed. The order among committed and uncommitted versions is still maintained by the explicit pointers in the line. This order among the versions is necessary to write back the correct committed version and to supply the correct version on a bus request. The EC design uses an additional state bit, the sTale (T) bit, to retain read-only data across tasks. First, we discuss how loads and stores are handled when caches have both committed and uncommitted versions and then discuss the stale bit.

3.4.1. Loads and stores Loads to committed lines are handled like cache misses and issue a bus request. The VCL searches the VOL in the reverse order beginning from the requestor for the closest previous *uncommitted* version; this version, if any, is supplied to the requestor. If no such version is found, the VCL supplies the most recent *committed* version, if any. This version is the first committed version that is encountered on the reverse search. All other committed versions need not be written back and are invalidated. On a store miss, committed versions are purged in a similar fashion.

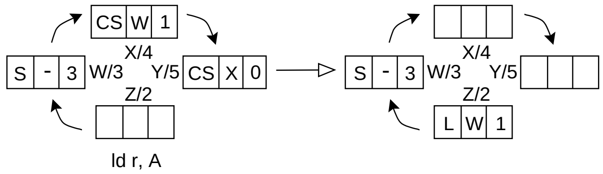

Figure 10: EC design: example load.

We illustrate the load executed by task 2 in the example program. Figure 10 shows two snapshots: one before the load executes and one after the load completes. Versions 0 and 1 have been committed (the C bit is set in the lines in caches X and Y). Task 2 executes a load that misses in cache Z and results in a bus request. The VCL knows that task 2 is the head task and determines that cache X has the most recent committed version. Cache X supplies the data which is also written back to the next level memory. Other committed versions (version 0) are invalidated and are never written back to memory. The VCL also inserts the new copy of version 1 into the VOL by modifying the pointers in the lines accordingly — the second snapshot shows the modified VOL.

Figure 11 illustrates the actions performed on a store miss. The first snapshot is taken before a store is executed by task 5. Versions 0 and 1 have been committed. Task 5 executes a store that misses in cache Y and results in a *BusWrite* request even though the line has a committed version. The VCL purges all committed versions of this line — it determines that version 1 has to be written back to the next level memory and the other versions (version 0) can be invalidated. Purging the committed versions also makes space for the new version (version 5). The modified VOL shown in the second snapshot contains only the two uncommitted versions.

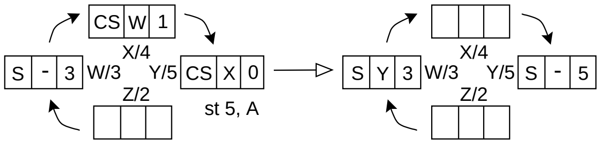

Figure 11: EC design: example store.

3.4.2. Stale copies The EC design makes task commits efficient by delaying to commit each cache line until a later time. Therefore, a cache line could have a *stale copy* because versions more recent than the version buffered by the committed task could be present in other caches. The base SVC design does not introduce stale copies because it invalidates all non-dirty lines whenever a task commits. The EC design uses the stale (T) bit to distinguish stale copies from correct copies and avoids issuing a bus request on accesses to correct copies. This additional information allows the EC design to retain read-only data (correct copies) across task commits. First, we illustrate that stale and correct copies are indistinguishable without the T bit and then show how the T bit is used.

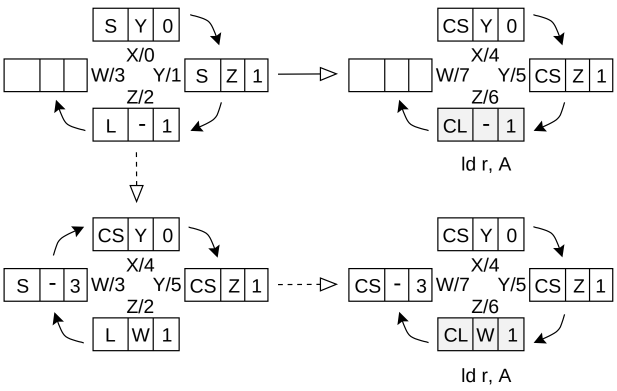

Figure 12: EC design: correct and stale copies.

Figure 12 shows two execution time lines — one that leaves a correct copy of address A (shown using solid lines) in cache Z and another that leaves a stale copy of address A in the same cache (shown using dashed lines). The first time line shows a sample execution of a modified version of our example program — task 3 in Figure 2 does not execute the store. The second time line shows an execution of our original program. The first snapshot is the same for both time lines. The second snapshot in the second time line is taken after tasks 0 and 1 have committed. The C bit is set in their caches and new tasks 4 and 5 have been allocated. The final snapshot in both time lines are taken when tasks 4 to 7 are active and before task 6 executes a load. In the first time line, the data in cache Z is a correct copy, since no versions were created after version 1; the load can be supplied data by just resetting the C bit and without issuing a bus request. In the second time line, the copy in cache Z is stale since the creation of version 3 and hence the load misses resulting a bus request. However, cache Z cannot distinguish between these two scenarios and has to issue a request in both cases to consult the VOL and obtain a copy of the correct version.

The EC design uses the stale (T) bit to distinguish between these two scenarios and avoids the bus request whenever a copy is not stale. The design maintains the invariant: the most recent version of an address and its copies have the T bit reset and the other copies and versions have the T bit set. This invariant is easily guaranteed by resetting the T bit in the most recent version, or a copy thereof, when it is created and setting the T bit in the previous versions, if any. The T bits are updated on the *BusWrite* request issued to create a version or a *BusRead* request issued to copy a version and hence do not generate additional bus traffic. Since stores in different tasks can be executed out of program order, an active task could execute a store to a copy that has the T bit set (the copy is not stale for this task, but is stale for the next task allocated to the same processor). Figure 13 shows the two time lines in our example with the status of the T bit. Cache Z can distinguish between the correct copy (T bit is not set) and the stale copy (T bit is set). The load hits if a correct copy is present and no bus request is issued.

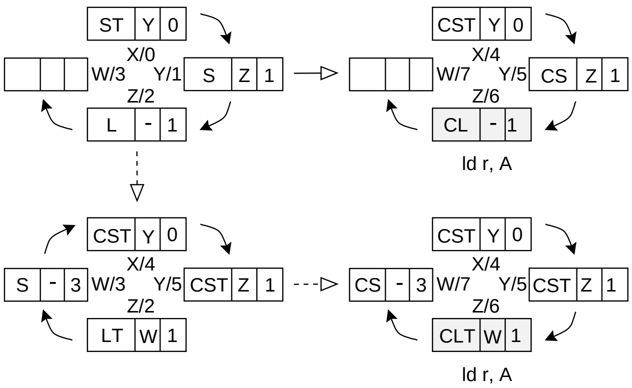

Figure 13: EC design: Using the stale bit.

The EC design eliminates the serial bottleneck in flushing the L1 cache on task commits by using the commit (C) bit. Also, this design retains non-dirty lines after task commits as long as they are not stale. More generally, read-only data used by a program is fetched *only once* into the L1 caches and never invalidated unless chosen to be replaced on a cache miss. Further a task commits by just setting the C bit in all lines in its L1 cache.

3.5. Implementing efficient task squashes (ECS)

The ECS design extends the EC design to allow task squashes for the EC design. Also, the ECS design makes the task squashes more efficient than in the base design by retaining non-speculative data in the caches across squashes using another state bit, the Architectural (A) bit. The structure of a line in the ECS design is shown in Figure 14.

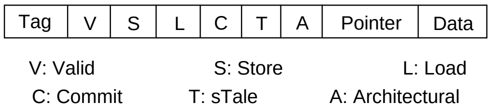

Figure 14: ECS design: structure of a line.

When a task squashes, all uncommitted lines (lines with the C bit reset) are invalidated by resetting the valid (V) bit. The invalidation makes the pointers in these lines and their VOLs inexact. The VOL has a (dangling) pointer in the last valid (or unsquashed) copy or version of the line and the status of the T bit in the lines are incorrect. The ECS design repairs the VOL of such a line when the line is accessed later either on a processor request or on a bus request. Updating the T bits is not necessary because it is only a hint to avoid a bus request and a squash would not incorrectly reset a stale version to be correct. However, the ECS design updates the T bit on this bus request by consulting the repaired VOL.

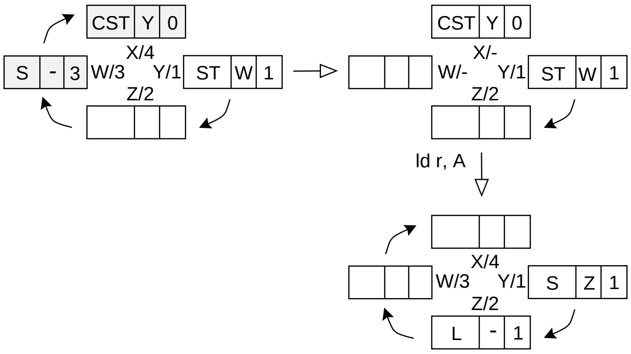

Figure 15: ECS design: VOL repair.

Figure 15 illustrates VOL repair with an example time line with three snapshots. The first snapshot is taken just before the task squash occurs. Tasks 3 and 4 are squashed; only version 3 is invalidated. The VOL with incorrect T bits and the dangling pointer are shown in the second snapshot. Task 2 executes a load that misses in cache W and results in a bus request. The VCL resets the dangling pointer and the T bit in cache Y. The VCL then determines the version to supply the load. Also, the most recent committed version (version 0) is written back to the next level memory. The third snapshot is taken after the load has completed.

3.5.1. Squash invalidations The base design invalidates non-dirty lines in the L1 cache on task squashes. This includes both speculative data from previous tasks and architectural data from the next level memory (or the committed tasks). The base design invalidates these lines because it does not track the creator of a speculative versions for each line and hence cannot determine whether the version in a line has been committed or squashed. Squashing non-speculative data leads to higher miss rates for tasks that are squashed and restarted multiple times.

To distinguish between copies of speculative and architectural versions, we add the architectural (A) bit to each cache line as shown in Figure 14. The A bit is set in a copy if either the next level memory or a committed version supplies data when a bus request issued to obtain the copy; else the A bit is reset. One of the VCL responses on a bus request specifies whether the A bit should be set or reset. Copies of architectural versions are not invalidated on task squashes, i.e., the ECS design only invalidates lines that have both the A and C bits reset. Further, a copy of a speculative version used by a task becomes an architectural copy when the task commits. However, the A bit is not set until the line is accessed by a later task, when the C bit is reset and the A bit is set.

3.6. Hit rate optimizations

The base and ECS designs incur severe performance penalties due to *reference spreading*. When a uniprocessor program is executed on multiple processors with private L1 caches, successive accesses to the same line that hit after missing once in a shared L1 cache could result in a series of misses. This phenomenon is also observed for parallel programs where the miss rate for read-only shared data with private caches is higher than that with a shared cache. We use *snarfing* [6] to mitigate this problem. Our SVC implementation snarfs data on the bus if the corresponding cache set has a free line available. However, an active task's cache can only snarf the version that the task can use unlike an SMP coherent cache. The VCL determines whether a task can copy a particular version or not and informs caches of an opportunity to snarf data on a bus request.

3.7. Realistic line size

The base and ECS designs assume that the line size of the L1 caches is one word. The final SVC design however allows lines to be longer than a word. Similar to an SMP coherent cache, we observe effects due to false sharing. In addition to causing higher bus traffic, false sharing leads to more squashes when a store to a cache line from a task is executed out-of-order with a load from a different byte or word in the same line from a later task. We mitigate the effects of false sharing by using a technique similar to the sector cache [7]. Each line is divided into sub-blocks and the L and S bits are maintained for each sub-block. The size of a sub-block or versioning block is less than that of the address block (storage unit for which an address tag is maintained). Also, when a store miss results in a *BusWrite* request, mask bits that indicate the versioning blocks modified by the store are also made available on the bus.

4. Performance evaluation

We report preliminary performance results for the SVC using the SPEC95 benchmarks. The goal of our implementation and evaluation is to prove the SVC design not just to analyze its performance. We underline the importance of a private cache solution by first showing how performance degrades rapidly as the hit latency for a shared cache solution is increased; the Address Resolution Buffer (ARB) is the shared cache solution we use for this evaluation. We mitigate the commit time bottlenecks in the ARB (by using an extra stage that contains architectural data) to isolate the effects of pure hit latency from other performance bottlenecks.

4.1. Methodology and configuration

All the results in this paper were collected on a simulator that faithfully models a Multiscalar processor. The simulator dynamically switches between a functional and a detailed cycle-by-cycle model to provide accurate and fast simulation of a program. The memory system model includes a fair amount of detail including an off chip cache, DRAM banks and interconnects between the different levels of memory hierarchy. The Multiscalar processor used in the experiments has 4 processors each of which can issue 2 instructions out-of-order. Each processor has 2 simple integer ALUs, 1 complex integer unit, 1 floating point unit, 1 branch unit and 1 address calculation unit, all of which are assumed to be completely pipelined. Inter-processor register communication latency is 1 cycle and each processor can send as many as two registers to its neighbor in every cycle. Loads and stores from each processor are executed in program order by using a load/store queue of 16 entries each.

The ARB is a fully-associative set of 32-byte lines with a total of 8KB storage per stage and five stages; the shared data cache that backs up the ARB is 2-way set associative and 64KB in size. The off chip cache is 4MB in size with a total peak bandwidth of 16 bytes per processor clock to the L1 data, instruction and task caches. Main memory access time for the first word is 24 processor clocks and has a RAMBUS-like interface that operates at half the speed of the processors to provide a peak bandwidth of 8 bytes every bus clock. All the caches and memory are 4-way interleaved. Both the ARB and the L1 data cache have 16 MSHRs/writebuffers each; each buffer can combine up to 8 accesses to the same line. Disambiguation is performed at the byte-level. The base ARB hit time is varied from 1 to 3 cycles in the experiments. Both the tags and data RAMs are single ported in all the caches.

The private caches that comprise the SVC are connected together and with the off chip cache by an 8-word split-transaction snooping bus where a typical transaction requires 3 processor cycles[3]. Each processor has its own private L1 cache with 16KB of 4-way set-associative storage in 32-byte lines. Both loads and stores are non-blocking with 8 MSHRs/writebuffers per cache. Each buffer can combine up to 4 accesses to the same line. Disambiguation is performed at the byte-level. L1 cache hit time is fixed at 1 cycle. The tag RAM is dual ported to support snooping while the data RAM is single ported.

4.2. Benchmarks

We used the following programs from the SPEC95 benchmark suite with train inputs except in the cases listed: *compress*, *gcc (ref/jump.i)*, *vortex*, *perl*, *ijpeg (test/specmun.ppm)*, *mgrid (test/mgrid.in)*, *apsi*, *fpppp*, and *turb3d*. All programs were stopped after executing 1 billion instructions. From past experience, we know that for these programs performance change is not significant beyond 1 billion instructions.

4.3. Experiments

Figure 16 presents the instructions per cycle (IPC) for a Multiscalar processor with either the ARB or the SVC. The configurations keep total data storage of the SVC and ARB/cache storage roughly the same. The percentage miss rates for the ARB and the SVC are shown on top of the IPC bar clusters (in that order). For the SVC, an access is counted as a miss if data is supplied by the next level memory; data transfers between the L1 caches are not counted as misses.

From these preliminary experiments, we make three observations: (i) the hit latency of data memory significantly affects ARB performance, (ii) the SVC trades-off hit rate for hit latency and the ARB trades-off hit latency for hit rate to achieve performance, and (iii) for the same total data storage, the SVC performs better than the ARB having a hit latency of 2 or more cycles as shown in Figure 16. The graphs in these figures show that performance improves in the range of 5% to 20% when decreasing the hit latency of the ARB from 3 cycles to 1 cycle. This improvement indicates that techniques that use private caches to improve hit latency are an important factor in increasing overall performance, even for latency tolerant processors like a Multiscalar processor.

[3]Bus arbitration occurs only once for cache to cache data transfers. An extra cycle is used to flush a committed version to the next level memory.

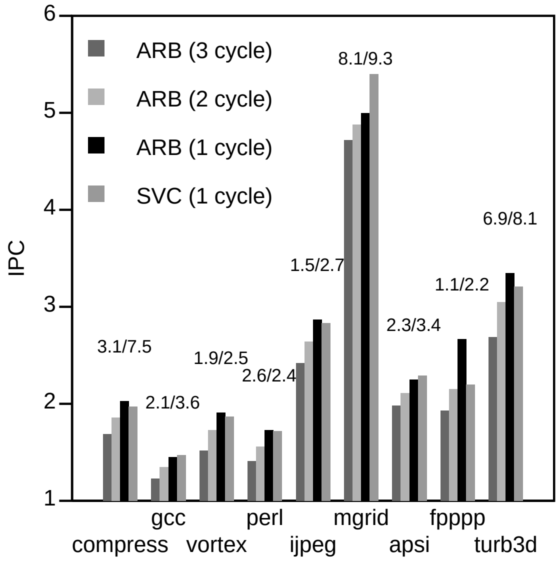

Figure 16: SPEC95 IPCs for ARB and SVC.

The distribution of storage for the SVC produces higher miss rates than for the ARB. We attribute the increase in miss rates for the SVC to two factors. First, distributing the available storage results in reference spreading [6] and replication of data reduces available storage. Second, a latest version of a line that caches fine-grain shared data between Multiscalar tasks constantly moves from one L1 cache to another (migratory data). Such fine-grain communication may increase the number of total misses as well.

5. Conclusion

Speculative versioning is important to overcome limits on Instruction Level Parallelism (ILP) due to ambiguous memory dependences in a sequential program. Our proposal, called the Speculative Versioning Cache(SVC), uses distributed caches to eliminate the latency and bandwidth problems of a previous solution, the Address Resolution Buffer, which uses a centralized buffer. The SVC conceptually unifies cache coherence and speculative versioning by using an organization similar to snooping bus-based coherent caches. A preliminary evaluation for the Multiscalar architecture shows that hit latency is an important factor affecting performance, and private cache solutions tradeoff hit rate for hit latency. The SVC provides hardware support to break ambiguous memory dependences allowing proposed next generation multiprocessors to use aggressive parallelizing software for sequential programs.

Acknowledgements

We thank Scott Breach, Andreas Moshovos, Subbarao Palacharla and the anonymous referees for their comments and valuable suggestions on earlier drafts of the paper.

This work was supported in part by NSF Grants CCR-9303030 and MIP-9505853, ONR Grant N00014-93-1-0465, and by U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346 and a donation from Intel Corp. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U. S. Army Intelligence Center and Fort Huachuca, or the U.S. Government.

References

[1] IEEE Standard for Scalable Coherent Interface (SCI) 1596-1992. IEEE 1993.

[2] S. E. Breach, T. Vijaykumar, S. Gopal, J. E. Smith, and G. S. Sohi. Data memory alternatives for multiscalar processors. Technical Report CS TR-1344, University of Wisconsin, Madison, Nov. 1996.

[3] 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.

[4] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 124–131, 1983.

[5] S. Gopal, T.N.Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. Technical Report CS TR-1334, University of Wisconsin, Madison, July 1997.

[6] D. Lilja, D. Marcovitz, and P.-C. Yew. Memory reference behavior and cache performance in a shared memory multiprocessor. Technical Report 836, CSRD, University of Illinois, Urbana-Champaign, Dec. 1988.

[7] J. S. Liptay. Structural aspects of the system/360 model 85 part II: The cache. *IBM Systems Journal*, 7(1):15–21, 1968.

[8] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 2–4, 1997.

[9] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang. The case for a single-chip multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, October 1–5, 1996.

[10] J. E. Smith and S. Vajapeyam. Trace processors: Moving to fourth-generation microarchitectures. *Computer*, 30(9):68–74, Sept. 1997.

[11] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 22–24, 1995.

[12] J. G. Steffan and T. C. Mowry. The potential for thread-level data speculation in tightly-coupled multiprocessors. Technical Report CSRI-TR-350, Computer Systems Research Institute, University of Toronto, Feb. 1997.