Speculative Versioning Cache

Sridhar Gopal, T. N. Vijaykumar, James E. Smith and Gurindar S. Sohi
gsri,vijay@cs.wisc.edu, jes@ece.wisc.edu, sohi@cs.wisc.edu
University of Wisconsin-Madison, USA

July 14 1997

Abstract

During the execution of a sequential program, dependences involving loads and stores are ambiguous until their addresses are known. Ambiguous memory dependences impede the extraction of instruction level parallelism. Memory dependence speculation is a hardware technique to overcome ambiguous memory dependences. This technique enables processors to execute loads and stores before the addresses of preceding loads and stores are known. When a store address is known, if a later load to the same address is found, a misspeculation is posted. The misspeculation is handled similar to a branch misprediction. Store values are buffered and committed in program order. Multiple uncommitted stores to a memory location create multiple versions of the location. Program order among the versions is tracked to maintain sequential semantics, i.e., the versions are committed in the correct order and loads are supplied with correct versions. The problem of maintaining sequential semantics in the presence of multiple speculative versions of memory locations is called speculative versioning of memory.

Most modern microprocessors dispatch instructions from a single dynamic instruction stream and maintain a time-ordering of the versions using load-store queues. The queues support a simple form of speculative versioning. Several recent proposed processors including the Multiscalar processor and single chip multiprocessors, however, partition the single dynamic instruction stream into multiple fragments and dispatch these fragments to multiple processing units (PUs). Such hierarchical processor designs naturally lead to hierarchical memory organizations. But load-store queues are not designed to support speculative versioning in hierarchical organizations. The Address Resolution Buffer (ARB) has been proposed to provide this support. The ARB uses a single shared buffer connected to all the PUs and hence, every memory access incurs the latency of the interconnection network. The shared buffer is a potential bandwidth bottleneck.

Our proposal, called the Speculative Versioning Cache (SVC), contains a private cache with each PU organized similar to a cache coherent Symmetric Multiprocessor. The SVC conceptually unifies cache coherence and memory dependence speculation. The SVC eliminates the latency and bandwidth problems of the ARB. A preliminary experimental evaluation of SPEC95 benchmarks shows that (i) hit latency is an important factor affecting performance (even for a latency tolerant processor like the multiscalar) and, (ii) private cache solutions trade-off hit rate for hit latency to achieve performance. The experiments show that the SVC performs up to 8% better than an ARB without any bank contention.

Keywords: Dependence Speculation, Memory Disambiguation, Multiscalar, Memory Versioning, Multiple Reader Multiple Writer protocol.

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 execute memory instructions in program order, as soon as their addresses can be determined. Store instructions (with their addresses) are then held in buffers. Buffered store instructions may have to wait for their data to become available, and to implement precise interrupts, store instructions are not allowed to complete and commit their results to memory until all preceding instructions are known to be free of exceptions.

For performance reasons, however, loads are allowed to pass 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. A key element of this straightforward approach is that a load instruction is not 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 [8, 6, 2] issue load instructions as soon as their addresses are known, even though the addresses of all previous stores may not be known. By issuing a load in this manner, these implementations speculate that it does not depend on previous stores; this form of speculation is called memory dependence speculation. Furthermore, one can envision issuing and computing store addresses out of order. Memory dependence speculation can enable higher levels of ILP, but it means there must be more advanced mechanisms to support this speculation and to recover from the misspeculations that inevitably occur. And it means that a significant amount of store data must be buffered and committed in correct program order.

Prior to being committed, each store to a memory location creates a new speculative *version* of that memory location. Multiple speculative stores to a memory location create multiple speculative versions of the location. Program order between the multiple versions of a location has to be tracked to (i) commit versions in program order, and (ii) supply (bypass) the correct version as per program order to loads. We call this problem *speculative versioning*. We illustrate the issues in speculative versioning with an example program containing two loads and two stores to the same memory location, A.

load R_1, A
store 2, A
load R_2, A
store 3, A

Sequential program semantics dictate that

- the load to R_1 should *not* read the value 2 created by the store even if it is executed after the store. This requires that the store must not overwrite memory location A until the load has executed and hence must be buffered.
- the load to R_2 should *eventually* read the value 2 created by the store. This requires that the load must be squashed and re-executed if it executes before the store and incorrectly reads the version stored in memory.
- memory location A should *eventually* have the value 3 stored independent of the order of execution of the stores. This requires that multiple stores to the same location have to be buffered before they are committed.
- program order between the loads and stores has to be tracked to guarantee the above three semantics.

Most modern microprocessors dispatch instructions from a single instruction stream, and issue load and store instructions from a common set of hardware buffers (e.g. reservation stations). This allows the hardware to maintain a time-ordering of loads and stores via simple queue mechanisms, coupled with address comparison logic. The presence of store queues provides 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 tasks [15] or traces [18] and use higher level instruction control units to distribute them to the processing units for execution. These approaches are aimed at higher levels of ILP and use replicated processing units with hierarchical control for a number of practical engineering reasons [12]. Proposed next generation multiprocessors [11, 17] that provide hardware support for dependence speculation use such hierarchical orchestration of the multiple processing units.

These highly parallel, hierarchical designs naturally lead to memory address streams with a similar hierarchical structure. In particular, each individual task or trace generates its own address stream, which can be properly ordered (disambiguated) within the processing unit that generates it, and at the higher level, the multiple address streams produced by the processing units must also be properly ordered. Consequently, there is a need for speculative versioning mechanisms that operate within this hierarchical organization.

The Address Resolution Buffer [4] (ARB) provides speculative versioning for such hierarchical organization of instructions. The ARB uses a shared buffer with each entry comprising multiple versions of the same memory location. However, there are two important problems with the ARB:

1. the ARB uses a single shared buffer connected to the multiple processing units (PUs) and hence, every load and store incurs the latency of the interconnection network. Also, the ARB has to provide sufficient bandwidth for all the PUs.

2. when each processing unit commits instructions, the ARB copies speculative state (versions created by stores) into the architected storage (main memory); this generates bursty traffic to memory and can increase the time to commit, which lowers the overall performance.

We propose a new solution for speculative versioning, the Speculative Versioning Cache (SVC), for processors that have hierarchical organizations. The SVC comprises a private cache for each processing unit, 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. The speculative memory state is committed in one clock cycle and is written back to main memory in a distributed manner using a lazy algorithm.

Section 2 introduces the hierarchical execution model in more detail and discusses the issues in providing support for speculative versioning for such execution models. Section 3 presents the Speculative Versioning Cache as a progression of designs to ease understanding. Section 4 gives a preliminary performance evaluation of the SVC to underline the importance of a private cache solution to speculative versioning for hierarchical execution models. Finally, section 5 concludes.

2 Problem

We first describe the execution model used by processors, that hierarchically organize the instructions in the active window, called the hierarchical execution model. Then, we discuss the issues involved in providing speculative versioning for such processors.

2.1 Hierarchical Execution Model

In processor designs with hierarchical organizations, resources like the register file, instruction issue queue and functional units are distributed among multiple processing units (PUs) for a variety engineering reasons, and a higher level unit controls the multiple PUs. The dynamic instruction stream is partitioned into fragments called *tasks*. These tasks form a *sequence* corresponding to their order in the dynamic instruction stream. The higher level control unit *predicts* the next task in the sequence and assigns it for execution to a free PU. Instructions in each task are executed by its PU, and its speculative state is buffered. Thus the predicted tasks are executed speculatively by the multiple PUs in parallel.

When a misprediction of a task in the sequence is detected, the buffered state of all the tasks starting from the incorrectly predicted task and beyond are invalidated[1] and the PUs are freed. The correct tasks in the sequence are then assigned for execution. This is called a *task squash*. When the prediction of a task is known to be correct, 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, the PU is free to execute a new task.

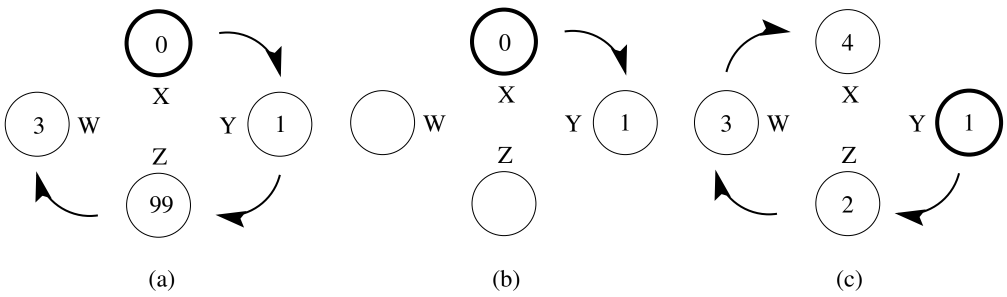


Figure 1: Hierarchical Systems: Task commits and squashes

In figure 1, we illustrate task squashes and task commits with an example. The system consists four PUs, W, X, Y and Z, shown using circles. The task sequence is $0,\ldots,4$. Initially, tasks 0, 1, 99 and 3 are predicted and executed in parallel as shown in part (a). When the misprediction of task 99 is detected, tasks 99 and 3 are squashed and their buffered states are invalidated. Now, new tasks 2 and 3 are executed by the PUs as shown in part (b). If one of the tasks complete execution, the corresponding PU is freed and task 4 is assigned for execution as shown in part (c). The sequence of the tasks assigned to the PUs enforces an *implicit* total order among the PUs; the solid arrowheads show this order among the PUs.

2.2 Speculative Versioning for Hierarchical Execution Models

Now, we discuss how loads and stores are handled in hierarchical processors to provide speculative versioning. A task executes a load as soon as its address is available, predicting that stores from previous tasks in the sequence do not write to the same address. A store to a memory location creates a new version of that location. The sequence of tasks imposes an order among the versions of a location from multiple tasks. This order among the versions of a location is key to provide support for speculative versioning.

2.2.1 Loads

The value to be supplied for a load is the closest previous version in the sequence starting from the task that executed the load. If a task loads from an address before storing to that address, this is recorded to detect potential violations of memory dependence

[1] An alternative model for recovery invalidates only the dependent chains of instructions by maintaining information at a finer granularity. This paper assumes the simpler model

Operation	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	Writeback buffered versions created by the task to main memory.
Squash	Invalidate buffered versions created by the task.

Table 1: Operations and Actions in Speculative Versioning

— if a store from a task with a previous sequence number writes to the same location at a later time, the load was supplied with an incorrect version.

2.2.2 Stores

When a task stores to a memory location, it is communicated to all later currently executing tasks in the sequence[2]. A task that receives a store from a previous task in the sequence squashes if a use before definition is recorded for the task — a memory dependence violation is detected. All tasks in the sequence starting from the incorrect one and beyond are squashed as on a task misprediction. However, a more aggressive squash model could squash only the computation dependent on the incorrectly executed load.

2.2.3 Task Commits and Squashes

When a task becomes the oldest executing task in the sequence, it becomes non-speculative and can commit its speculative memory state (versions created by stores from this task) to main memory. Committing a version involves copying the stores from the speculative buffers to the architected storage. Since the task squash model we assume is simple, the speculative buffered state associated with a task is invalidated when it is squashed.

2.3 Coherence and Speculative Versioning

The above operations are summarized in Table 1. The functionality in this table requires the hardware to track the PUs that have performed load/store to each address and the order among the different copies/versions of each address. Similar functionality is required of a cache coherent Symmetric Multiprocessor (SMP), which tracks the caches that have a copy of every address. SMPs, however, need not track the order among these copies since all the copies are of a single version.

SMPs typically use snooping-bus based cache coherence [5] to implement a Multiple Reader Single Writer (MRSW) protocol that tracks copies of a single version of each memory location. This MRSW protocol 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 (MRMW) protocol[3] that tracks copies of multiple

[2] In reality, the store has to be communicated only until the task that has created the next version, if any, of the location.
[3] MRMW protocols have been implemented in software by distributed shared memory systems like the Midway [3] and the TreadMarks [14].

speculative versions of each memory location. This MRMW 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 using explicit pointers (for example, 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 PUs using automatic parallelization [11, 17]. 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 give a base SVC design that provides support for speculative versioning with minimal modifications to the cache coherence scheme. We then highlight the performance problems in the base design and introduce optimizations one by one in the rest of the subsections.

3.1 Snooping-bus based Cache Coherence

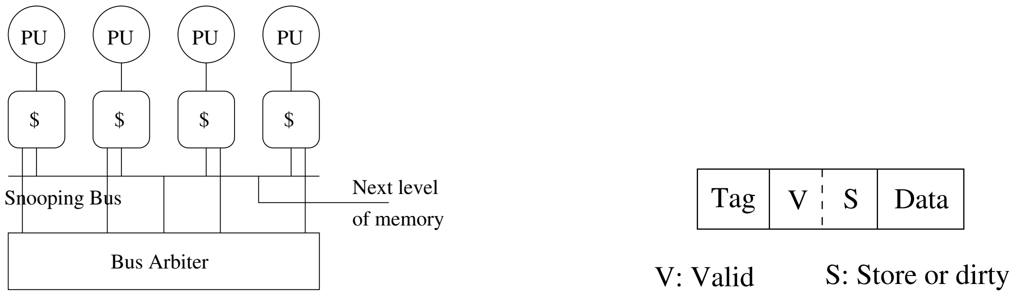
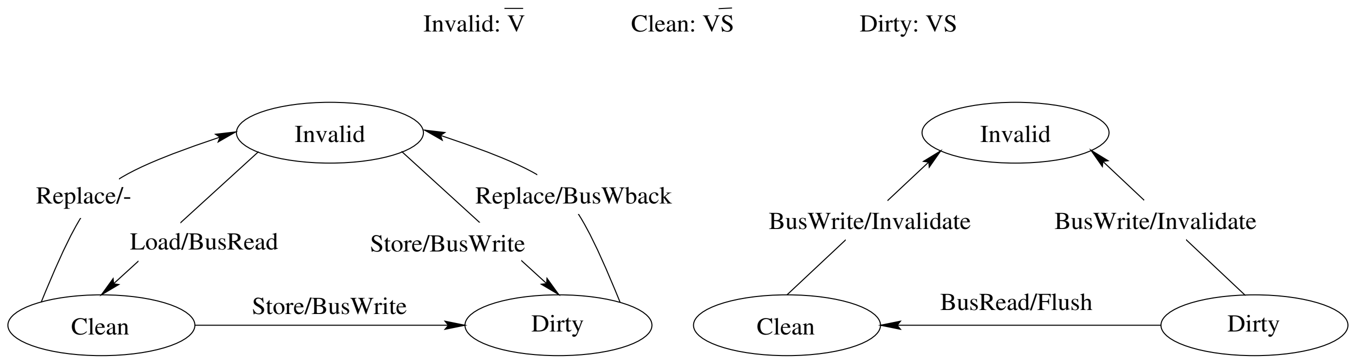


Figure 2: Snooping-bus based Cache Coherence in an SMP

Figure 2 shows a 4-processor SMP with private L1 caches that uses a snooping-bus to keep them consistent. The structure of a cache line is shown in Figure 2. Each line comprises an address tag that is used to identify the data that has been cached, the data that is cached, and two bits (valid and store) representing the state of the line. The valid (V) bit represents the validity of the data in the line. The store (S) or dirty bit is set if the processing unit stores to any word in the data portion of the line.

Each L1 cache has a controller that determines whether a request from a PU (load or store) hits or misses by consulting a tag directory which comprises the tag portions of all lines. Each cache line is in one of three states: Invalid, Clean and Dirty and the controller implements a finite state machine that determines the operations performed on a PU request (Figure 3). Loads to a clean/dirty line and stores to a dirty line result in cache hits. Cache hits do not generate bus requests. A load(store) to an invalid line results in a miss and a *BusRead*(*BusWrite*) request is issued. A request on the bus is snooped by the L1 caches



(a) PU REQUEST

(b) BUS REQUEST

Figure 3: An Invalidation-based Cache Coherence Protocol

and the next level memory, and the requested line, if present, is accessed. If a valid copy is present, an appropriate response is generated as shown in Figure 3(b). A *BusWrite* request is generated on a store miss to a clean line to invalidate copies of the line in the other caches, if any. The finite state machine in these two figures implements an invalidation-based MRSW protocol that invalidates all but the requestor's copy of a line on a *BusWrite* request. The protocol thus ensures that at most one copy of a line is dirty. A dirty line is flushed on the bus on a *BusRead* request; a *BusWback* request is generated 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 coherence 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.

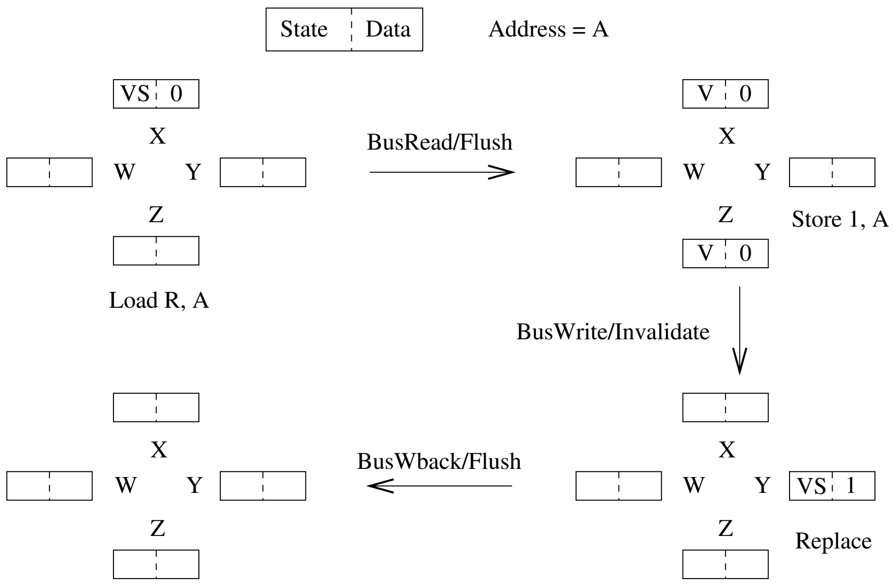


Figure 4: Snooping-bus based Cache Coherence: Example

Figure 4 shows four snapshots of the cache lines corresponding to address A in a 4-PU SMP. The four PUs are designated as X, Y, Z and W. An invalid line for address A is shown as an empty box, while a valid line for address A shows the state bits that are set. The first snapshot is taken before PU Z executes a load to address A. The cache miss results in a *BusRead* request and cache X supplies the line on the bus. The second snapshot shows the final state of the lines; they are clean. PU

Y issues a *BusWrite* request to perform a store to address 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 the line, it casts out the line to memory by issuing a *BusWback* request; the final state is shown in the fourth snapshot; only memory contains a valid copy of the line.

3.2 Base SVC Design

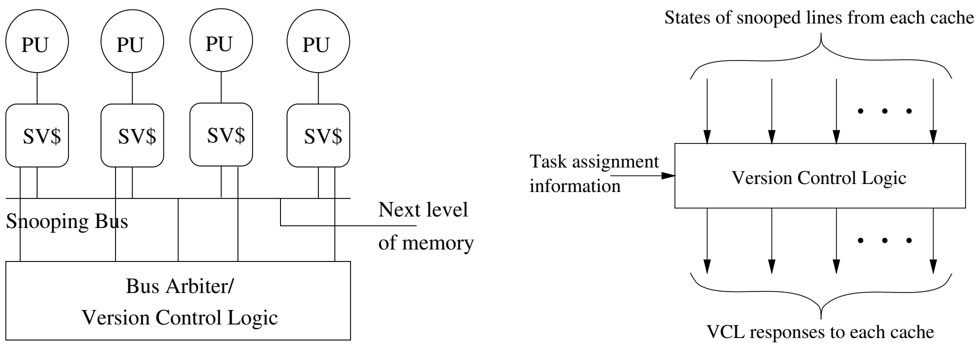


Figure 5: Speculative Versioning Cache: Organization and Version Control Logic

Tag	V	S	L	Pointer	Data

V: Valid
S: Store
L: Load

Figure 6: Base SVC Design: Structure of a line

A base SVC design is shown in Figure 5; it minimally modifies the memory system of the snooping-bus based cache coherent SMP to support speculative versioning for hierarchical systems. We assume that loads and stores to the *same* address from the *same* PU are executed in program order[4]. This design also assumes that the cache line size is one word; later, we relax this assumption. We first introduce the modifications, and then discuss how the individual operations listed in Table 1 are performed. The modifications are:

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 it — a potential violation of memory dependence in case a previous task stores to the same line.

2. Each cache line maintains a pointer that identifies the PU (or L1 cache) that has the next copy/version, if any, in the Version Ordering List (VOL) for that line. It is important to note that the pointer identifies a PU rather than a task because identifying a dynamic task would require infinite number of tags.

3. The base design uses combinational logic called the Version Control Logic (VCL) to provide speculative versioning. When a PU request hits in the L1 cache, it is satisfied locally without generating a bus request as in an SMP; the VCL is also not used. When a PU request misses in the cache, the cache controller generates a bus request and is snooped by

[4]This can be performed by a conventional load-store queue.

the L1 caches and the next level of memory. The states of the requested line in each L1 cache are supplied to the VCL. The VOL for this line is given by the pointers in the lines. The VCL uses the bus request, the program order among the tasks, and the VOL to generate appropriate responses for each cache. The cache controllers make state transitions based on the initial state of the line, the bus request and the VCL response. A schematic of the Version Control Logic is shown in Figure 5. For the base design, the VCL responses are similar to that of the disambiguation logic in the ARB [4]; the disambiguation logic searches for previous/succeeding stages in a line to satisfy a PU access.

The program order among the tasks assigned to the PUs enforces an *implicit* total order among the PUs and hence the L1 caches. In addition to the VOL, this implicit order is required by the VCL to generate cache responses for a bus request. There is no such total order among the PUs/caches in a multiprocessor[5]. This order is illustrated in our examples by the presence of arrows between the caches.

3.2.1 Examples

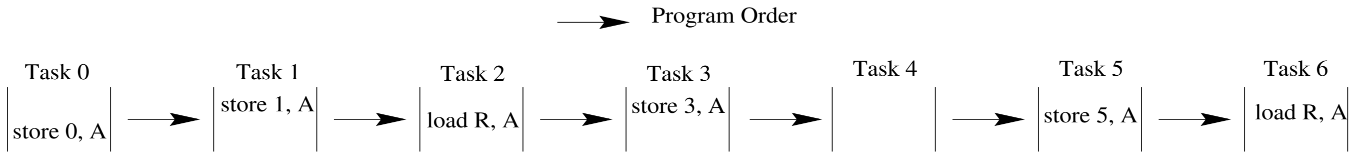


Figure 7: SVC: Example Program with Loads and Stores

We present the SVC designs by discussing the operations performed on loads, stores, task commits and task squashes, and illustrate them with example executions of a program. Each example shows a time line of snapshots of the cache lines corresponding to address A in a 4-PU system. The state and data in each line are shown in a box whereas the pointers are shown in the VOL beside the snapshot. The pointers in the VOL are shown using hollow arrowheads. The absence of the line in a cache is represented by an empty box; if the line is present, the state bits that are set (except the V bit) are shown in the box. The cache line corresponding to the head (non-speculative) task is shown using a bold box. The line in a cache being squashed is shown using a double box.

The four PUs in the hierarchical system are designated as W, X, Y and Z; the pointer in a cache line uses one of these designators to point to the next copy/version of the line. Tasks can be allocated to the PUs in an arbitrary fashion. Hence, we number tasks (beginning from 0) to show the program order among them; smaller numbers indicate older tasks. The hardware does *not* assign numbers to tasks; they are shown in the example figures for illustration purposes only. The PUs and the tasks they execute are shown alongside each box. The total order among the PUs is shown using solid arrowheads between the boxes. Loads and stores to address A in the example program and its tasks are shown in Figure 7. In all our examples, we use a convention that the version of address A created by task i has a value i. Figure 8 shows an example time line.

3.2.2 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 requestor in the VOL and searches the VOL in the reverse order beginning from the requestor.

[5]Memory ordering may impose a total order among the individual memory accesses but not among the caches.

The first version that is encountered is the required one. If no version is found, then the data is supplied by the next level of memory. The implicit order among the PUs 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 length of the list is small.

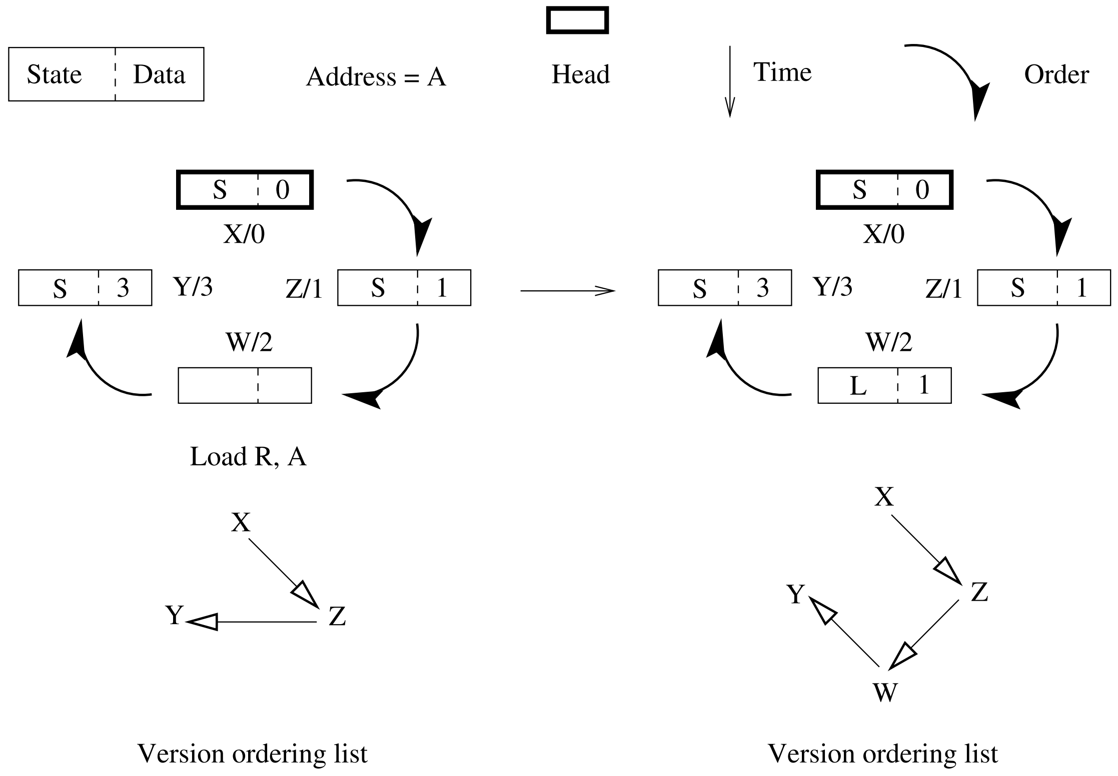


Figure 8: Base SVC Design: Example Load

We illustrate the load executed by task 2 in the example program. Figure 8 shows two snapshots: one before the load executes and one after the load completes. PU W executing task 2 issues a load to cache W which issues a *BusRead* request. The VCL searches the VOL in the reverse order beginning from cache W and finds the correct version to supply, the version in cache Z (that was created by task 1).

3.2.3 Stores

The SVC performs more operations on a store miss than are required by a conventional SMP to provide coherence. 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 (inclusive, 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 PU when it receives an invalidation response from the VCL and the L bit is set in the line.

We illustrate the stores executed by task 1 and 3 in the example program. Figure 9 shows four snapshots. The first snapshot is taken before PU Y executing task 3 issues a store to cache Y, resulting 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 the line in the other caches (unlike an SMP memory system) to allow for multiple versions of the same line. The second snapshot is

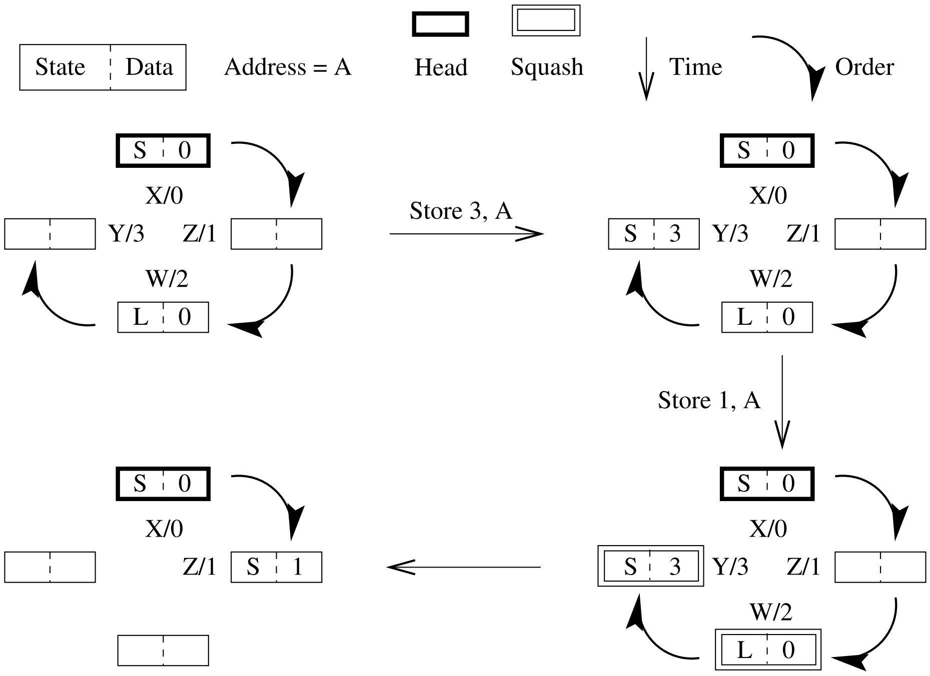


Figure 9: Base SVC Design: Example Stores

taken after the store completes.

The first snapshot is taken before PU Z executing task 1 issues a store to cache Z, resulting in a *BusWrite* request. Based on task assignment information, the VCL sends an invalidation response to each cache from the one after cache Z until the one before cache Y, which has the next version of the line (cache Y is not included since it does not have the L bit set) — VCL sends an invalidation response to cache W. But, the load executed by task 2, which follows the store by task 1 in program order, has already executed. Cache W 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 and restarted (the squash model we assume squashes all tasks until the tail). The final snapshot is taken after the store by task 1 has completed.

3.2.4 Task Commits and Squashes

The base SVC design handles task commits and squashes in a simple manner. When a PU 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 writeback all the dirty lines of a task immediately, a list of the stores executed by the task is maintained by the PU. When a task is squashed, all the lines in the corresponding cache are invalidated.

3.2.5 Finite State Machine

Support for the SVC is similar to an SMP cache; a cache line is in one of three states: Invalid, Clean and Dirty. The state encodings and the finite state machine are shown in Figure 10. The L bit is not shown in the figure. The finite state machine used by the L1 cache controllers in the base SVC design includes the VCL responses and a few modifications to those used by an SMP. A *BusWrite* request is generated on a store to a clean or invalid line. Caches that receive the *Invalidate* response from the VCL invalidate their cache line and send a squash signal to the PU if the L bit is set in their line. The VCL allows

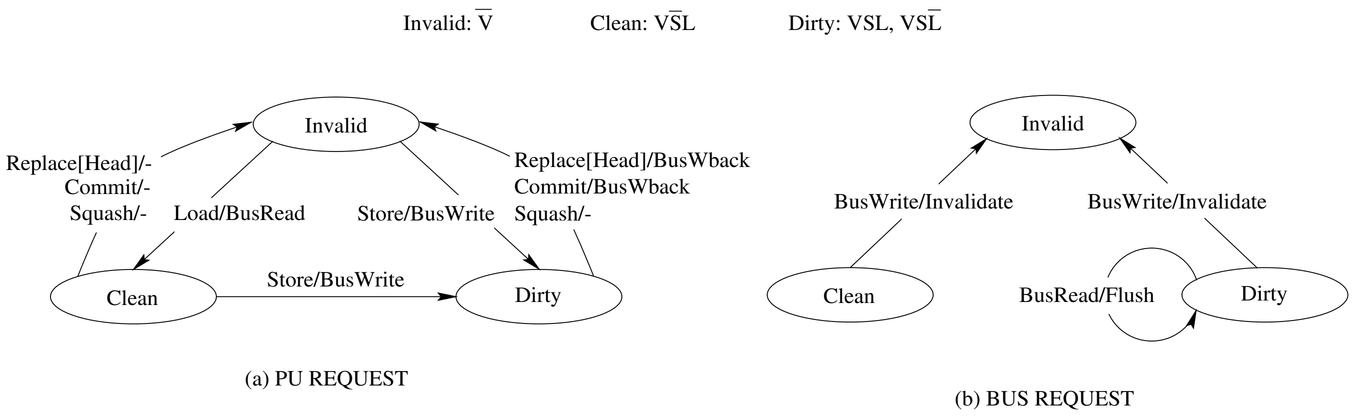


Figure 10: Base SVC Design: Finite State Machine used by the L1 Cache Controllers

multiple speculative versions to exist by not invalidating all lines as an MRSW protocol would. Another difference between the state machines in Figures 3 and 10 is that a dirty line that supplies data on a *BusRead* request remains dirty since the next level memory can not store the speculative data yet.

A valid line can be replaced by the head task's cache. Other caches cannot replace a valid line because it contains information necessary to guarantee correct execution. Hence, either a different replacement victim is chosen or the PU request stalls until the task becomes non-speculative (i.e., the head task). On task commits, all clean lines are invalidated and dirty lines are written back to the next level memory by issuing *BusWback* requests. All lines are invalidated on task squashes.

3.2.6 Performance Problems

The base design has two significant performance problems:

1. Writebacks performed during task commits lead to bursty traffic on the bus and increased task commit time that could make task commits a serial bottleneck.

2. Invalidating all lines after task commits and squashes implies that *every* task begins execution with a *cold* cache; read-only data is also invalidated on task commits and squashes.

3.3 Road map

In the following sections, we discuss advanced SVC designs that redress the performance problems with the base design, especially the ones due to task commits and squashes.

1. *ECS design* provides efficient task commits and squashes. To ease the understanding of this design, we describe an intermediate design, the EC design, that provides efficient task commits and assumes tasks are not squashed. Then, we present the ECS design that adds efficient task squashes to the EC design. The ECS design makes task commits efficient by distributing the writebacks of dirty lines over time with a lazy algorithm. Also, it retains read-only data in the caches across task commits. Even though the ECS design does more book-keeping, task squashes are as simple as the base design but are more efficient as they retain non-speculative data in the caches across task squashes.

2. *HR design* boosts the hit-rate of the ECS design by allowing requests to snarf the bus to account for reference spreading.

3. *RL design* makes the above designs more realistic by allowing the size of a cache line to be more than one word.

3.4 Implementing Efficient Task Commits (EC)

The EC design avoids the expensive cache flush on task commits by maintaining an extra state bit, called the commit bit, in each cache line. Task commits are made lightweight and writebacks are distributed over time with a lazy algorithm. The EC design eliminates the writeback bursts on the bus during task commits. Also, no hardware is necessary to maintain a list of stores performed by each task. Further, the EC design keeps the L1 caches warm across tasks, thereby improving their utilization.

Tag	V	S	L	C	T	Pointer	Data

V: Valid
S: Store
L: Load
C: Commit
T: sTale

Figure 11: EC Design: Structure of a line

The structure of a cache line in the EC design is shown in Figure 11. The EC design sets the commit (C) bit in *all* lines in the L1 cache associated with a task when it commits. This operation is entirely local to that cache and does not involve the bus. A dirty committed line is written back, if necessary, when it is accessed the next time (either on a PU request or on a bus request). Since the same PUs and caches are reallocated for new tasks, committed versions remain in the caches across tasks. The order among committed and uncommitted versions is still maintained by the explicit pointers in the line. The stale (T) bit is used by the EC design 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. The EC design is an intermediate design; it assumes that tasks are not squashed. The ECS design adds support for efficient task squashes to the EC design.

3.4.1 Loads

Loads to committed lines are treated like misses and generate a bus request. On a *BusRead* request, the VCL searches the VOL in the reverse order beginning from the task that issued the bus request for an uncommitted version. The first uncommitted version that is encountered is the version to be supplied to the requestor. If no uncommitted version is found, then the most recent committed version is the required version. The most recent *committed* version is the first committed version encountered while searching the VOL in the reverse order beginning from the most recent version. All other committed versions need not be written back and hence are invalidated.

We illustrate the load executed by task 2 in the example program. Tasks 0 and 1 have committed and the C bit is set in the lines in caches X and Z. Figure 12 shows two snapshots: one before the load executes and one after the load completes. PU W executing task 2 issues a load to cache W which issues a *BusRead* request. The VCL determines that the task on cache W is the head task from the task assignment information and, from the VOL, it determines that cache Z has the most recent committed version. Cache Z supplies its version (version 1) to cache W and the data is also written back to 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

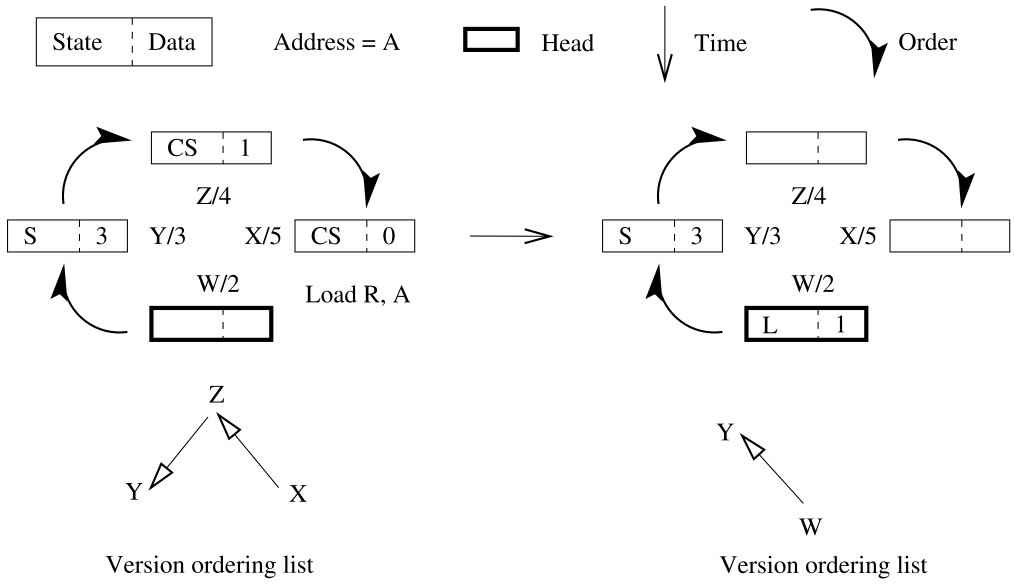


Figure 12: EC Design: Example Load

the VOL by modifying the pointers in the lines accordingly — the modified VOL is shown alongside the second snapshot.

3.4.2 Stores

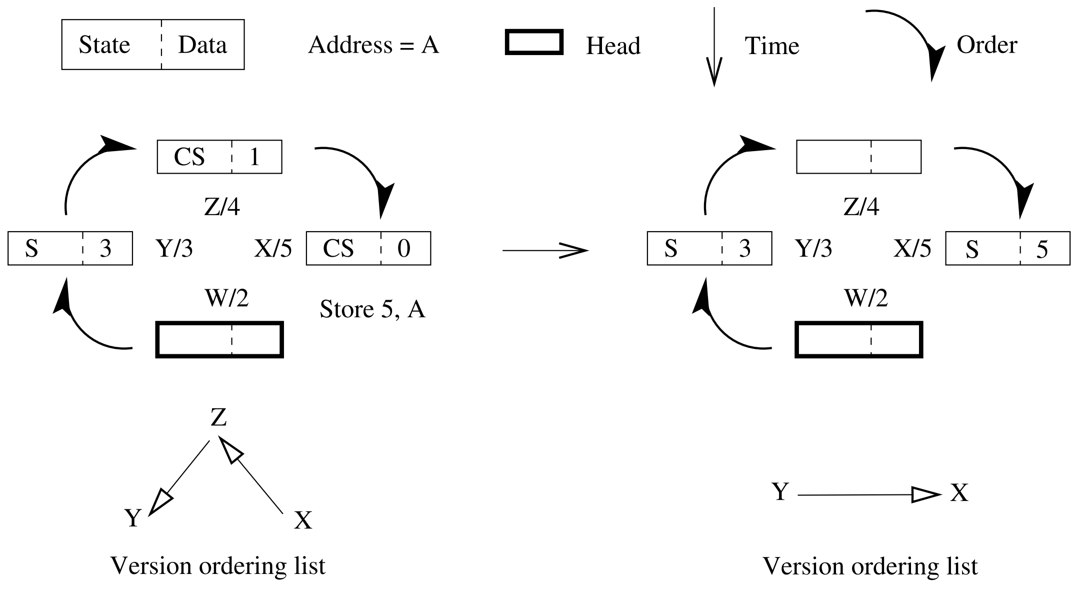


Figure 13: EC Design: Example Store

Similarly on a store miss, committed versions are purged as the example in figure 13 illustrates. The first snapshot is taken before the store executes. Versions 0 and 1 have been committed. The VOL corresponding to this snapshot is shown below the snapshot. Note that the tasks need *not* be assigned in a circular fashion to the PUs. Now, PU X executing task 5 issues a store to cache X which issues 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 contains only the two uncommitted versions.

3.4.3 Stale Copies

The base design invalidates all non-dirty lines and hence read-only data, whenever a task commits. This has serious implications on the bandwidth requirement of the snooping-bus since every task that accesses a read-only data issues a *BusRead* request. The EC design eliminates this problem by exploiting an opportunity to retain read-only data in a cache across multiple tasks. We highlight this opportunity using the example in Figure 14.

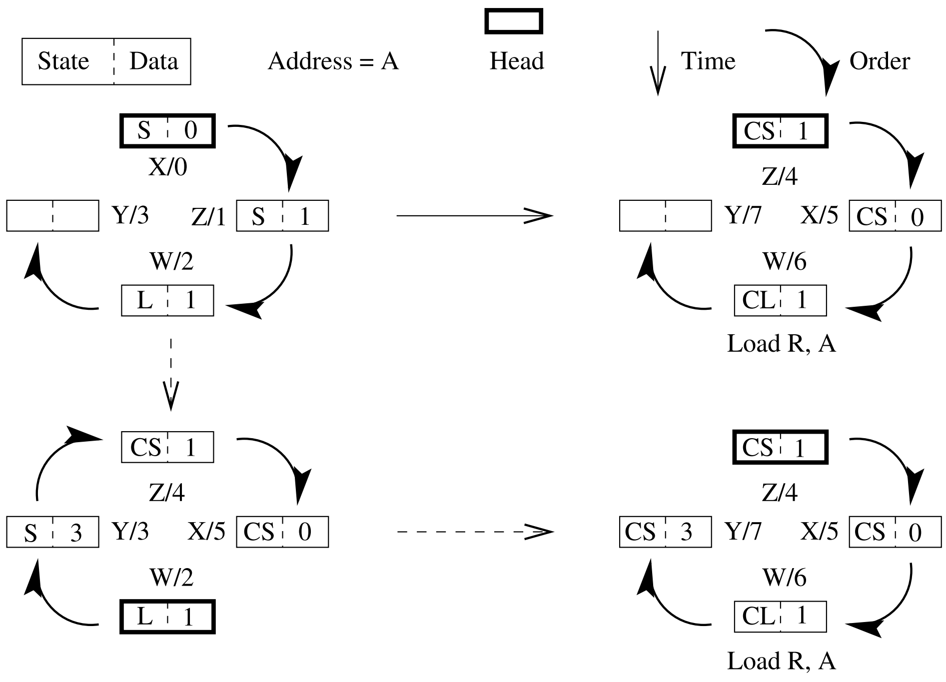


Figure 14: EC Design: Distinguishing between Correct and Stale Copies

The figure shows two time lines both of which start with the first snapshot. The first time line (shown using a regular arrow) corresponds to a modified version of our example program — task 3 in Figure 7 does not have a store. The second time line (shown using dashed arrows) corresponds to our example program. The second snapshot in this time line is taken after task 3 has executed its store and only tasks 0 and 1 have committed. The final snapshot in both time lines are taken when tasks 4 to 7 are executing and before task 6 executes its load. In the first time line, this load can safely reuse the data in cache W by just resetting the C bit since no versions were created after version 1. In the second time line, the load can *not* reuse the data since cache W has a copy of a version that has become stale since the creation of version 3. Cache W can not distinguish between these two situations and hence has to consult the VOL to obtain a copy of the correct version. However, the VOL can be obtained only on a bus request and we would like to avoid the associated overhead.

The EC design uses the sTale (T) bit to distinguish between these two situations and to avoid the bus request whenever a copy is not stale. This is performed by maintaining the following 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 guaranteed by resetting the T bit when the most recent version or a copy thereof is created and/or setting the T bit in the copies of 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 does not generate any extra bus traffic. Since stores in different tasks can be executed out of program order, a cache issuing a *BusWrite* request may reset/set the T bit depending on whether it is correct or already stale.

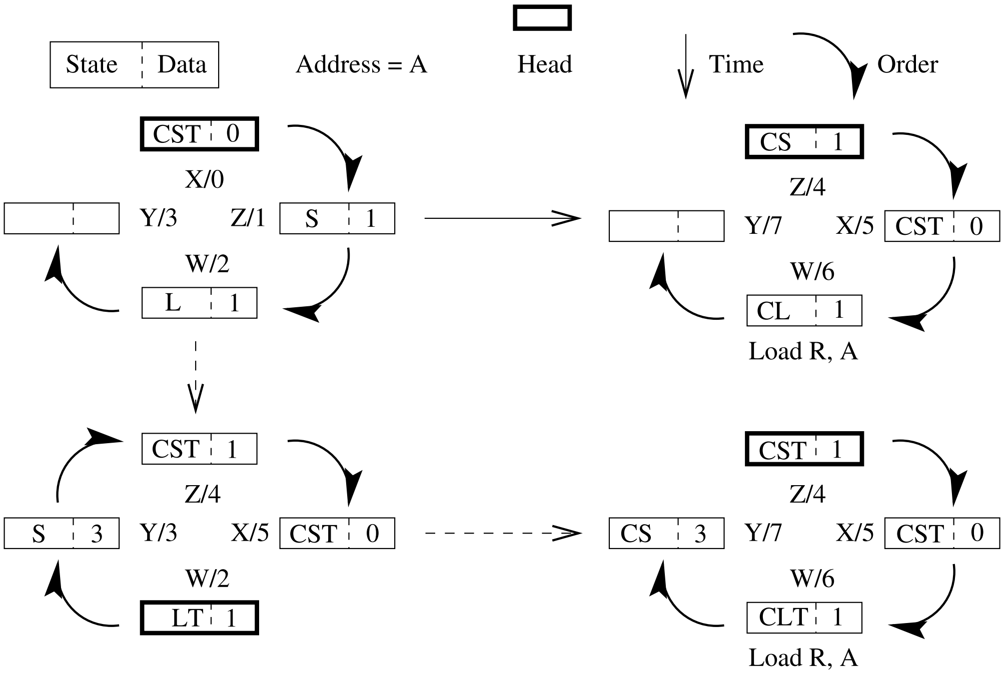


Figure 15: EC Design: Stale bit

Figure 15 shows snapshots of the two time lines discussed earlier with the status of the T bit. In the first time line (regular arrow), the T bit is reset in cache W since it has a copy of the most recent version. The load executed by task 6 can then safely reuse the copy of the line by just resetting the C bit. In the second time line (dashed arrows), the T bits are set in caches W and Z when task 3 executes its store. This state is shown in the second snapshot in this time line. The final snapshot is taken before task 6 executes its load. When the load executes, cache W has the T bit set indicating a stale copy, and hence it issues a *BusRead* request.

The EC design eliminates the serial bottleneck in flushing the L1 cache on task commits by using the commit (C) bit. Also, it retains read-only copies 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 as a replacement victim. Further these operations are done by just setting the C bit in all lines when a task commits and processing the individual lines in a lazy manner. Next, we discuss the ECS design, which completes the EC design by supporting task squashes.

3.5 Implementing Efficient Task Squashes (ECS)

The ECS design provides simple task squashes for the EC design although it keeps track of more information than the base design. Also, the ECS design makes the task squashes efficient by retaining non-speculative data in the caches across squashes using the Architectural (A) bit. The structure of a line in the ECS design is shown in Figure 16.

When a task squashes, all uncommitted lines (lines with the C bit reset) are invalidated. The invalidation makes the pointers in these lines and their VOLs inexact. The inconsistencies in the VOL are the (dangling) pointer in the last valid (or unsquashed)

Tag	V	S	L	C	T	A	Pointer	Data

V: Valid   L: Load   T: sTale
S: Store   C: Commit   A: Architectural

Figure 16: ECS Design: Structure of a line

version/copy of the line and the status of the T bit in each line. The ECS design fixes[6] the VOL of such a line when the line is accessed later either on a PU request or on a bus request. Fixing the T bit 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 fixes the T bit anyway by consulting the repaired VOL.

Figure 17 illustrates VOL repair with an example time line showing three snapshots. The first snapshot is taken just before the task squash occurs. Tasks 3 and 4 are squashed; only the uncommitted version (version 3) is invalidated. The VOL with incorrect T bits and the dangling pointer are shown in the second snapshot. PU W executing task 2 issues a load to cache W which issues a *BusRead* request. The VCL resets the dangling pointer in cache Z and resets the T bit in this version. The VCL determines the version to supply the load after fixing the VOL and T bits. 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.

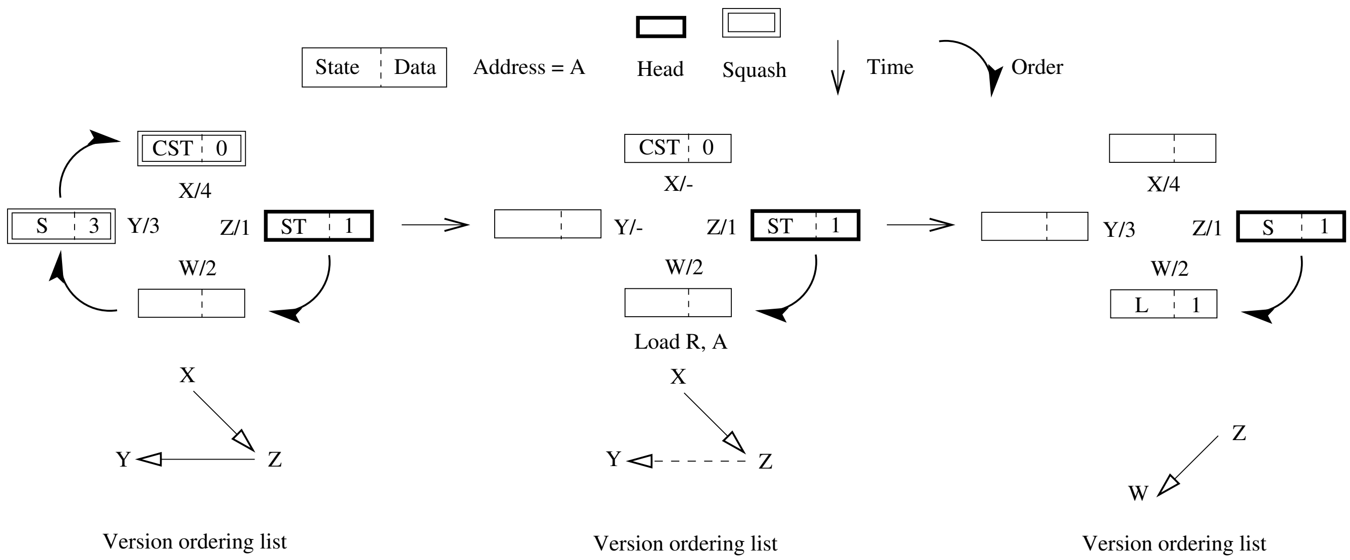


Figure 17: ECS Design: Repairing Version Ordering List after a Squash

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 main memory (or the committed tasks). The base design invalidates these lines because it is not known whether the creator of a speculative version has committed or squashed. This leads to higher miss rates for tasks that are squashed and restarted multiple times.

[6] It is possible to repair the VOL since the squash model we consider in this paper always results in the tail portion of the VOL to be invalidated.

To distinguish between copies of speculative and architectural versions, we add the architectural (A) bit to each cache line as shown in Figure 16. The A bit is set in a copy if either main memory or the head task supply data on the *BusRead* request issued to obtain the copy; else the A bit is reset. The VCL response on a *BusRead* 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. When this copy is reused by a later task, the C bit is reset and the A bit is set to remember that the version is architectural.

3.6 Hit Rate Optimizations (HR)

This design fixes a specific performance problem in the previous SVC designs, viz., reference spreading. When a uniprocessor program is executed on multiple PUs with private L1 caches, successive accesses to the same line that hit after a single miss in a shared L1 cache could result in a series of misses. This phenomenon is observed even for parallel programs where misses for read-only shared data is higher with private caches than a shared one. We borrow the technique proposed by [9], *snarfing*, to mitigate this problem. Our SVC implementation snarfs data on the bus if the corresponding cache set has a free line available. However, an L1 cache in an SVC can snarf only the version of data that a task executing on the corresponding PU can use unlike the caches that implement a MRSW protocol. The VCL is able to determine whether a cache can copy a particular version or not and hence can inform the caches of an opportunity to snarf on a data transfer during a bus transaction.

3.7 Realistic Line size (RL)

The RL design allows the line size of the L1 caches to be more realistic: line sizes greater than a word are allowed, unlike the previous designs. With line sizes greater than a word, false sharing effects are observed [13]. In addition to causing higher bus traffic, false sharing leads to more squashes when a store from a task shares a cache line with a load (to a different address) from a later task and they are executed out-of-order. We mitigate the effects of false sharing by using a technique similar to the sector cache [10]. Each line is divided into sub-blocks and the L and S bits are maintained for each sub-block. The sub-block or versioning block size is less than the address block size (storage unit for which an address tag is maintained). Also, *BusWrite* requests are issued with mask bits that indicate the versioning blocks modified by the store that caused the bus request.

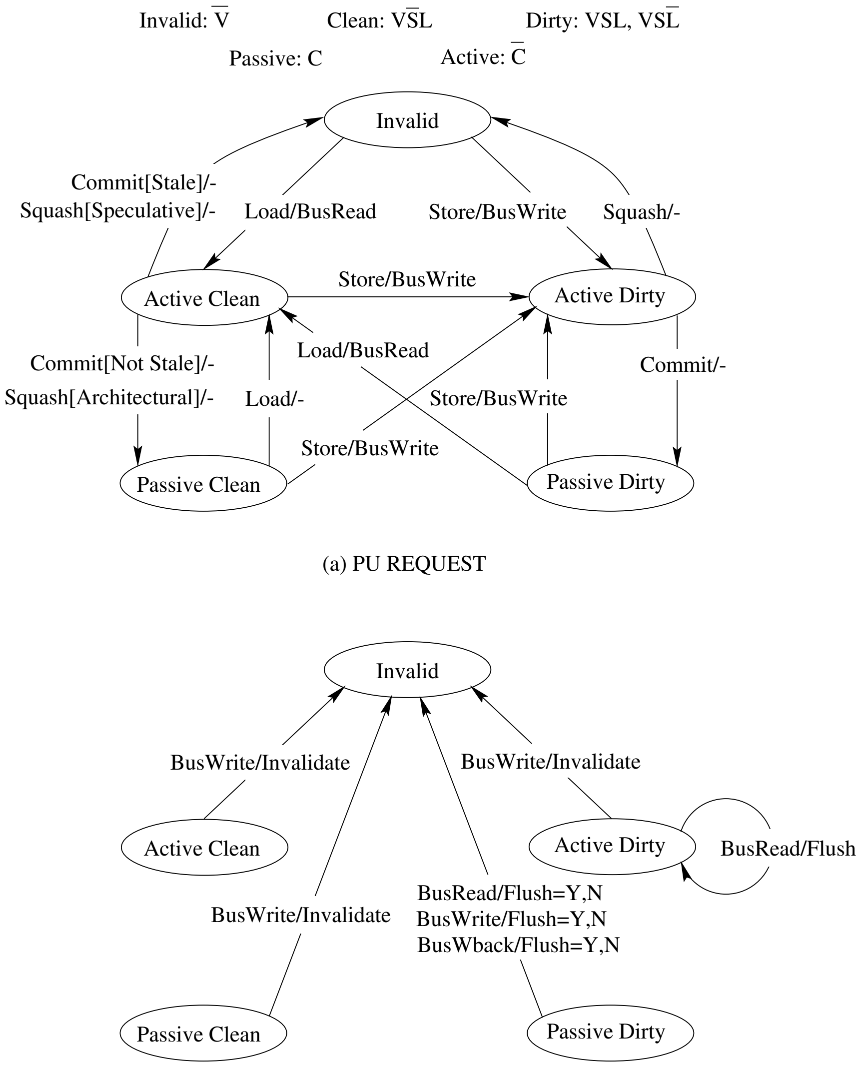
3.8 Final SVC design

The final Speculative Versioning Cache (SVC) design is shown in Figure 5 and the structure of each SVC line is shown in Figure 16. In addition, the final design uses a hybrid update–invalidate coherence protocol that dynamically selects either write-update or write-invalidation for each bus request. The write-update protocol helps to reduce the inter-task communication latency through memory, i.e., the latency between a store from a task and a dependent load from one of the later tasks, that executes concurrently. On the other hand, the write-invalidate protocol could reduce the bus bandwidth requirement.

Our solution for speculative versioning for hierarchical execution models resolves complexity by decoupling the directory from the disambiguation logic. The directory is distributed as in an SMP and the disambiguation logic is the Version Control Logic. We summarize the final SVC design by first describing the finite state machine used by the cache controllers and then the

VCL. The finite state machine has been intentionally made simple to project the important features of the design; specifically, it builds on an invalidation-based coherence protocol.

3.8.1 Finite State Machine



(a) PU REQUEST

(b) BUS REQUEST

Figure 18: Finite State Machine for the Final SVC Design L1 Cache Controllers

The finite state machine used by the L1 cache controllers of the final SVC is shown in Figure 18. Each line in this SVC is in one of five states: Invalid, Active Clean, Active Dirty, Passive Clean and Passive Dirty. The two active states correspond to the two valid states in Figure 10 — lines in these two states have been accessed by the task that is executing on the corresponding PU. Adding the C bit to the base SVC adds the two passive states; committed versions and their copies are in the passive states. The bits set/reset in each state are shown in the figure. To simplify the exposition, the L, T, A and X bits are not discussed unless of significance. Also, replacements and cache to cache transfers of read-only data are not shown. Lines in the clean states can be replaced without a bus request. Dirty lines generate a *BusWback* request on a replacement. Similar to the base design, active lines can be replaced only by the head task.

On a task commit, an active dirty line becomes passive and hence its writeback is postponed until later. However, an active clean line is either invalidated or retained on a task commit depending on whether it is stale or not; again this is postponed until later by just setting the C bit. On a task squash, an active dirty line is invalidated resulting in a temporary inconsistency in the line's VOL which is fixed by the VCL later. An active clean line is either invalidated or retained on a task squash depending on whether it is a copy of a speculative version or the architectural version.

Load/store requests to invalid and active lines are handled similar to the base design. A load to a line in the passive clean state does not result in a bus request unlike the base design which would generate a *BusRead* request for the load (as the line would be invalid). On a *BusRead* request, the data could be supplied either by a line that is in the active dirty state or the passive dirty state. On a bus request, a line in passive dirty state is invalidated whether it is flushed or not. Similar to the base design, the VCL determines the copies/versions to be invalidated on a *BusWrite* request. However, an invalidate response received by a line in one of the passive states will not generate a squash unlike a line in one of the active states. The most recent committed version is written back to the next level memory.

A further optimization is to retain lines in the passive dirty state that are flushed on bus requests. These lines can be retained until they are cast out by the cache. This could reduce the writeback traffic to the next level of memory by delaying the writebacks long enough for versions to become stale. Also, average miss penalty could improve since cache to cache transfers incur less latency than transfers from the next level memory. The following section summarizes the actions of the Version Control Logic.

3.8.2 Version Control Logic

The version control logic or the VCL uses the information in the L1 cache lines on a bus request to

- provide cache coherence among the multiple copies of a version.
- track the order among the multiple versions of a line based on the VOL represented explicitly by the pointers in these lines.
- supply the correct version for a load by using the VOL and the PU allocation order.
- write back committed versions in order.
- repair the VOL after task squashes.
- maintain the stale (T) bit.
- maintain the architectural (A) bit.
- inform a cache when it can snarf data to be transferred during a bus transaction.

Generating appropriate responses for each cache on a bus request is easy once the VOL is available. This is obtained in a straightforward manner from the explicit pointers in each line. Searching the VOL is *not* difficult since the entire VOL is available and the maximum length of the list is small — combinational logic can be used to perform searches.

4 Performance Evaluation

We report preliminary performance results of the SVC design using the SPEC95 benchmarks. The goal of our implementation and evaluation is to prove the SVC design more than analyzing its performance, as the SVC is the *first* attempt at a private cache solution for speculative versioning for hierarchical execution models. 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 assume unlimited bandwidth to the ARB to isolate the effects of pure hit latency from other performance bottlenecks. Also, we mitigate the commit time bottlenecks by using an extra stage, that contains architectural data.

4.1 Methodology

All the results in this paper were collected on a simulator that faithfully models a multiscalar processor using a hierarchical execution model. The simulator accepts annotated big endian MIPS instruction set binaries produced by the multiscalar compiler, a modified version of gcc. To provide results that reflect reality as accurately as possible, the simulator performs all the operations of a multiscalar processor and executes all the program code, except system calls, on a cycle-by-cycle basis. System calls are handled by trapping to the operating system of the simulation host.

4.2 Configuration

The multiscalar processor used in the experiments has 4 processing units (PUs), each of which can issue 2 instructions out-of-order. Each PU has 2 simple integer functional units (FUs), 1 complex integer FU, 1 floating point FU, 1 branch FU and 1 address calculation unit. All the FUs are assumed to be completely pipelined. Inter-PU register communication latency is 1 cycle and each PU can send as many as two registers to its neighbor in every cycle. Each PU has its own instruction cache: 32KB, 2-way set associative storage. Instruction cache hits return a line in 1 cycle and misses to the next level memory cause a penalty of 10 cycles. Loads and stores from each PU are executed in program order by using a load/store queue.

The higher level control unit that dispatches tasks to the individual PUs maintains a 1024 entry 2-way set associative cache of task descriptors. The control flow predictor of this control unit uses a dynamic path-based scheme that selects from up to 4 task targets per prediction and tracks 7 path histories XOR-folded into a 15-bit path register [7]. The predictor storage consists of both a task target table and a task address table, each with 32K entries indexed by the path register. Each target table entry is a 2-bit counter and a 2-bit target. Each address table entry has a 2-bit counter and a 32-bit address. The control flow predictor includes a 64 entry return address stack.

The PUs are connected to the ARB and the data cache by a crossbar. The ARB comprises a fully-associative cache with 256 rows and five stages; a shared data cache of 32KB or 64KB direct-mapped storage in 16-byte lines backs up the ARB. Both loads and stores are non-blocking with 32 MSHRs [16] and a 32-entry writeback buffer each in the ARB and the data cache. An MSHR can combine up to 8 accesses to the same line. The data cache has a 32-entry writeback buffer. The MSHRs and the writeback buffer are equally divided among 4 banks of storage. Disambiguation is performed at the byte-level. Hit time is varied from 1 to 4 cycles in the experiments, with an additional penalty of 10 cycles for a miss supplied by the next level of the data memory (plus any bus contention). Contention in the crossbar between the PUs and the ARB/data cache is not modeled.

Benchmark	ARB – 32KB	SVC – 4x8KB
compress	0.031	0.075
gcc	0.021	0.036
vortex	0.019	0.025
perl	0.026	0.024
ijpeg	0.015	0.027
mgrid	0.081	0.093
apsi	0.023	0.034

Table 2: Miss Ratios for ARB and SVC

The private caches that comprise the SVC are connected together and with the next level memory by a 4-word split-transaction snooping-bus where a typical transaction requires 3 processor cycles[7]. Each PU has its own private L1 cache with 8KB or 16KB of 4-way set-associative storage in 16 byte lines (32KB or 64KB total). Both loads and stores are non-blocking with 8 MSHRs and a 8-entry writeback buffers per cache. An MSHR can combine up to 4 accesses to the same line. Disambiguation is performed at the byte-level. Hit time is assumed to be 1 cycle, with an additional penalty of 10 cycles for a miss supplied by the next level of the data memory (plus any contention).

4.3 Benchmarks

We used the following programs from the SPEC95 benchmark suite with inputs given in parentheses: compress (train/test.in), gcc (ref/jump.i), vortex (train/vortex.in), perl (train/scrabble.pl), ijpeg (test/specmun.ppm), mgrid (test/mgrid.in) and apsi (train/apsi.in). The programs were stopped after executing 200 million instructions, if they ran that long. From past experience we know that for these programs performance change is not significant beyond 200 million instructions.

4.4 Experiments

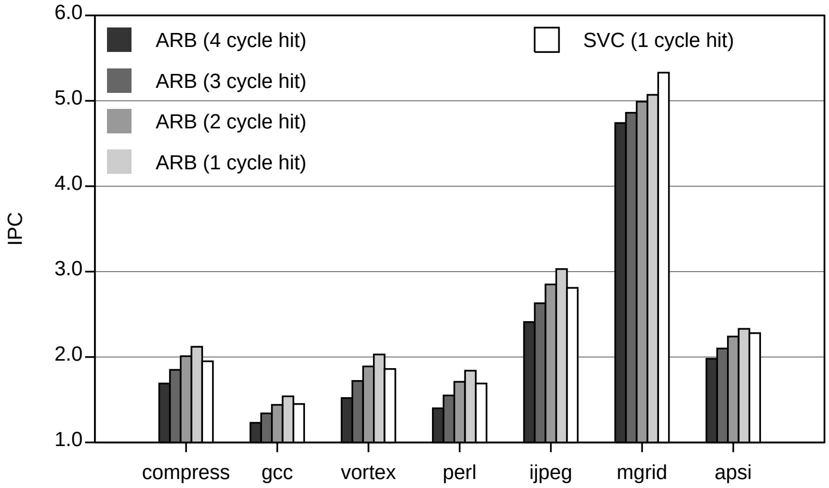


Figure 19: SPEC95 IPCs for ARB and SVC — 32 KB total data storage

[7]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.

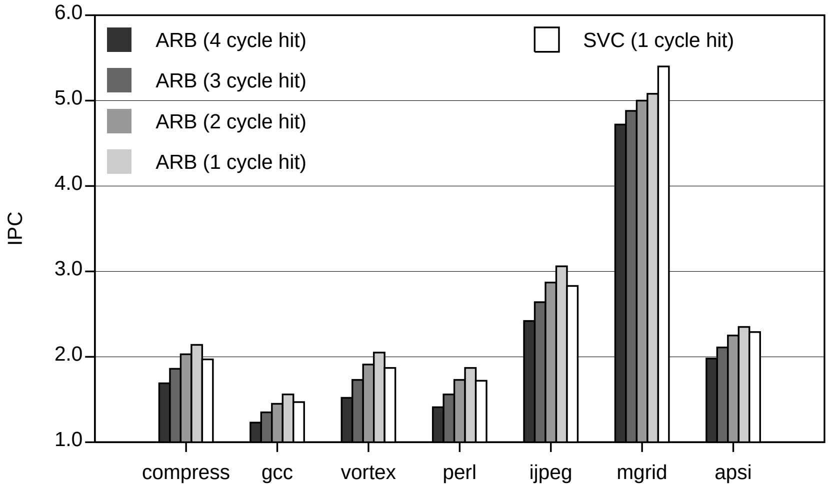


Figure 20: SPEC95 IPCs for ARB and SVC — 64KB total data storage

Benchmark	4x8KB	4x16KB
compress	0.348	0.341
gcc	0.219	0.203
vortex	0.360	0.354
perl	0.313	0.291
ijpeg	0.241	0.226
mgrid	0.747	0.632
apsi	0.276	0.255

Table 3: Snooping Bus Utilization for SVC

Figures 19 and 20 present the instructions per cycle (IPC) for a multiscalar processor configured with an ARB and an SVC. The configurations keep total data storage of the SVC and ARB/cache storage roughly the same, since the amount of ARB storage is rather modest compared to its data cache. The miss rates for the ARB and the SVC for a total storage of 32KB are shown in Table 2. 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. Table 3 presents the bus utilization for the SVC.

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 a contention-free ARB having a hit latency of 3 or more cycles. The graphs in these figures show that even for a 64KB data cache, performance improves in the range of 8% to 35% when decreasing the hit latency from 4 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.

Comparing the same total amounts of storage, the distribution of storage for the SVC produces higher miss rates than for the ARB (perl is an exception). We attribute the increase in miss rates for the SVC to two factors. Firstly, distributing the available storage results in reference spreading [9] and replication of data reduces available storage. Secondly, the fine-grain sharing of data between multiscalar tasks causes the latest version of a line to constantly move from one L1 cache to another (migratory data). Such fine-grain communication may increase the number of total misses as well. Such communication is evident from the fairly high bus utilization of the SVC ranging from 22% to 36%. The high bus utilization for mgrid are mostly due to misses to the next level memory.

From Figures 19 and 20, we find that the IPCs for SVC are better than that for ARB with a hit time of 2 or more cycles for gcc, apsi and mgrid. Both schemes perform nearly as well for compress, vortex, perl and ijpeg. However, the ARB has been modeled without any bank contention and assumes a high bandwidth commit path from any stage to the architectural stage. With these in mind, we see that for a total storage of 64KB, the SVC outperforms the ARB by as much as 8% for mgrid.

5 Conclusion

Speculative versioning is important for overcoming limits on Instruction Level Parallelism (ILP) due to ambiguous memory dependences in a sequential program. A solution to speculative versioning must track the program order between loads and stores to the same address to (i) commit the stores in program order to the architected memory and, (ii) supply the correct value for loads. Conventional solutions like load–store queues are well suited for modern superscalar microprocessors that dispatch instructions from a single stream. Proposed next generation processors use replicated processing units (PUs) that dispatch and/or issue instructions in a distributed manner. These future approaches use a hierarchical execution model and generate memory address streams with a hierarchical structure.

The Address Resolution Buffer (ARB), the previous solution to speculative versioning for such hierarchical processor designs, uses a shared L1 cache. Hence, every load/store access incurs the latency of the interconnection network between the PUs and the cache. Further, the ARB can lead to serial bottlenecks if the time to copy the speculative state of a task to the next level memory is large.

We proposed the Speculative Versioning Cache (SVC) that comprises a private L1 cache with each PU making the processor organization similar to a Symmetric Multiprocessor (SMP). Memory references that hit in the private cache do not use the bus and committing the speculative state of a task is done in one clock cycle. The speculative state is written back to next level memory in a distributed manner using a lazy algorithm. Based on preliminary performance evaluation, we find that the SVC performs up to 8% better than the ARB (with 2 cycle hit latency) for SPEC95 benchmarks. Also, the private cache organization of the SVC makes it a feasible memory system for proposed next generation multiprocessors that execute sequential programs on tightly coupled PUs using aggressive automatic parallelization. The SVC provides hardware support to overcome ambiguous memory dependences and hence the parallelizing software can be less conservative on sequential programs. Further, the SVC organization facilitates execution of parallel and multi-threaded programs on the same hardware.

Acknowledgements

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] PowerPC 620 RISC microprocessor technical summary. IBM order number MPR620TSU-01, October 1994.

[3] B.N.Bershad, M.J.Zekauskas, and W.A.Sawdon. The Midway distributed shared memory system. In *Compcon 93*, pages 528–537. CS Press, 1993.

[4] Manoj Franklin and Gurindar S. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[5] James 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.

[6] Doug Hunt. Advanced performance features of the 64-bit PA-8000. In *Compcon 95 Digest of Papers*, pages 123–128. CS Press, March 1995.

[7] Quinn Jacobson, Steve Bennett, Nikhil Sharma, and James E. Smith. Control flow speculation in multiscalar processors. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, 1997.

[8] Jim Keller. The 21264: A superscalar Alpha processor with out-of-order execution. Presentation at the 9th Annual Microprocessor Forum, San Jose, California, October 1996.

[9] 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, December 1988.

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

[11] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kun-Yung 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.

[12] Subbarao Palacharla, Norman P. Jouppi, and James E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 2–4, 1997.

[13] David A. Patterson and John L. Hennessy. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, 1996.

[14] P.Keleher et al. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of Usenix Winter Conference*, pages 115–132, Berkeley, California, 1994. Usenix Association.

[15] Gurindar S. Sohi, Scott 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.

[16] Gurindar S. Sohi and Manoj Franklin. High-bandwidth data memory systems for superscalar processors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, April 8–11, 1991.

[17] J. Gregory Steffan and Todd 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, February 1997.

[18] Sriram Vajapeyam and Tulika Mitra. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 1–12, June 2–4, 1997.