Data Memory Alternatives for Multiscalar Processors

Scott E. Breach, T. N. Vijaykumar, Sridhar Gopal, James E. Smith, Gurindar S. Sohi

Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, Wisconsin 53706

Abstract

This work considers data memory alternatives for multiscalar processors that can support the aggressive control and data speculative execution of loads and stores. We discuss the key issues that must be dealt with for such a data memory design and partition the design space of alternatives on the basis of composition, i.e. whether the storage for speculative and architectural versions is separate or aggregate, and on the basis of organization, i.e. whether the storage for speculative and architectural versions is shared or private. Moreover, we attempt to address a broad spectrum of solutions by considering two schemes in terms of centralized and distributed designs: a known scheme, the address resolution buffer which provides distinct speculative and architectural storage; and a novel scheme, the time-sequence cache which merges the speculative and architectural storage. We have performed a preliminary experimental evaluation of designs from opposite ends of the spectrum of solutions. Our experimental evidence from a simulation of a multiscalar processor with a centralized address resolution buffer and a distributed time-sequence cache shows (i) that hit latency is an important performance factor (even for a latency tolerant processor like a multiscalar processor) and (ii) that distributed schemes may trade-off hit rate for hit latency to improve performance over centralized schemes.

1 Introduction

A multiscalar processor extracts instruction-level parallelism (ILP) using multiple sequencers, coupled with aggressive control speculation, to build a vast active window of instructions. From this window, multiple processing units are used to execute multiple instructions in a given cycle; multiple loads and stores in this window are executed in a data speculative fashion. Supporting this mode of operation requires a data memory that can: (i) buffer multiple speculative versions of a memory location, (ii) detect violations of true (store to load) memory data dependences, (iii) convey correct data from stores to loads within the active window, and (iv) commit the results of correct speculative execution and squash the results of incorrect speculative execution.

In this paper, we consider data memory alternatives that can support the aggressive execution model of a multiscalar processor. We consider both a known scheme, the *address resolution buffer* [3] which provides distinct speculative and architectural storage, and a novel scheme, the *time-sequence cache* which merges speculative and architectural storage. Moreover, we consider both centralized and distributed designs for each scheme. After a brief background on the concepts and memory requirements of the multiscalar paradigm in section 2, our investigation of this subject follows. The progression is an exploration of the key issues for any design in section 3, a consideration of fundamental aspects that partition the design space in section 4, a description of a broad spectrum of data memory alternatives in section 5, and an experimental evaluation of data memory alternatives from opposite ends of this spectrum in section 6. This study is followed by a discussion in section 7 of how data memory designs such as those described here deal with precise interrupts in a uniprocessor system and sequential consistency in a multiprocessor system. The paper closes in section 8 with a summary of this work.

2 Multiscalar Concepts and Memory Requirements

In the multiscalar model of execution, the control flow graph (CFG) is partitioned into regions called tasks. A multiscalar processor assigns tasks to one of a collection of processing units (PUs) for execution by passing an initial program counter to the PU. Multiple tasks execute in parallel on the PUs, resulting in an overall execution rate of multiple instructions per cycle. The order among tasks is maintained by treating the PUs as a circular queue with head and tail pointers, indicating the *active tasks* (those in concurrent execution) from oldest to newest respectively. The execution of instructions in tasks may be both control and data speculative. Correct execution causes tasks to be committed; whereas incorrect speculation causes tasks to be squashed. To maintain sequential semantics, tasks are committed in the original program order and squashed from the task at point of incorrect speculation onwards. More details of this entire process may be found in [12].

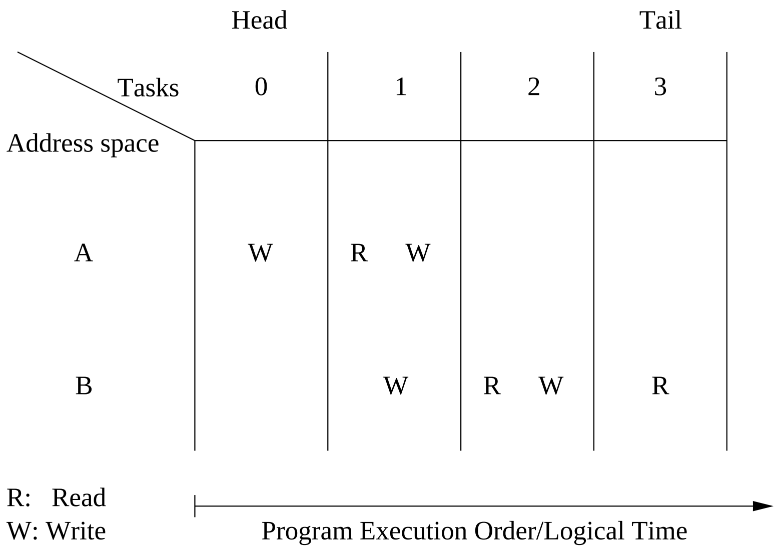


Figure 1: Memory requirements for a multiscalar processor.

To facilitate an understanding of the memory requirements of multiscalar execution, consider Figure 1. In the figure, memory locations are shown along the Y-axis (memory locations A and B are singled out). Logical time, corresponding to sequential program execution order, is along the X-axis. The logical time has been divided into 4 ”epochs” corresponding to 4 sequential tasks: 0, 1, 2, and 3. In the figure, task 0, corresponding to the head task, is non-speculative; tasks 1-3 are (control and/or data) speculative.

In multiscalar, the logically-sequential tasks 0-3 execute speculatively and in parallel. As tasks execute, they produce and consume memory values corresponding to their logical order in the program execution. Thus, when task 1 reads location A, it would read the latest version corresponding to that logical time, i.e, the (architectural) version created by task 0. Likewise, when it stores into location A, it creates a new (speculative) version of A, which is the version used by succeeding tasks. Similarly, task 1 creates a (speculative) version of location B; the speculative version is used by task 2, which in turn produces another speculative version of location B. Continuing further, when task 3 reads location B, it should access the latest (logical and speculative) version of B – the version produced by task 2.

Since all but the head task are speculative, the memory system must provide sufficient storage to allow all the speculative values of a task to be buffered (the number of such values created corresponds to the number of memory locations written by each task). In addition, the memory system needs to provide storage to support multiple versions of a memory location.

As the speculative tasks are executing in parallel, and produce/consume memory values in a real time that is different from the logical time, the memory system must ensure that the dependences dictated by the logical time are not violated. For example, if task 2 had read the value of location B before task 1 had created it, task 1 must convey to task 2 that it has performed an incorrect data speculation, and must squash (and recover).

Next, as tasks complete and commit, the speculative state that they created must be merged with the architectural state at that point to create the architectural state for the next epoch. For example, when task 1 is committed, its speculative state gets merged with the existing architectural state (the architectural state corresponding to the end of task 0), creating a version of the architectural state at a (logical) time corresponding to the completion of task 1.

3 Key Issues

From the above discussion, we can summarize that there are four key issues that need to be dealt with in the design of the memory system of a multiscalar processor: (i) the allocation/deallocation of speculative storage, (ii) the access of speculative and/or architectural storage, (iii) the ordering of load and store instructions to respect memory data dependences, and (iv) the transfer of state among PUs, speculative storage, and architectural storage. Without loss of generality, this discussion is restricted to inter-task (in lieu of intra-task as this problem is the same as for a superscalar processor) memory accesses and to the top level of the multiscalar data memory hierarchy, as its proximity to the PUs make it a natural choice, and all relevant loads and stores are easily "seen" at this level.

3.1 Allocation/Deallocation

The allocation/deallocation of speculative storage provides space to maintain the different *versions* of a memory location. It is important to realize that the allocation/deallocation of this storage does not actually create a version. Instead versions are only created as a result of the speculative execution of loads and stores on this storage. A fundamental choice is whether to allocate/deallocate speculative storage all-at-once or on-demand. For an all-at-once approach, speculative storage is allocated en masse (on task invocation). For an on-demand approach, speculative storage must be allocated when a task performs loads or stores. The advantage of the all-at-once approach is that the association between tasks and/or instructions and speculative storage may be straightforward on commit or squash. The primary drawback of this approach is that it is not possible in general to know precisely how much speculative storage is needed ahead of time. If too little is allocated, it may lead to stalls or squashes when it runs out. The advantage of the on-demand approach is that the internal fragmentation problems of the all-at-once approach may be eliminated. It is still possible to run out of speculative storage (as in the all-at-once approach), but this scenario is less likely to occur due to the elimination of internal fragmentation. The primary drawback of this approach is that the association between a task and/or instruction and speculative storage may be difficult to establish on commit or squash.

3.2 Access

There are two modes of access for the speculative storage: (i) given a particular memory address, select loads and stores performed on the memory address by all active tasks or (ii) given a particular task, select loads and stores performed by the task. The first mode of access is instruction oriented, triggered by the execution of loads or stores. The second mode of access is task oriented, triggered by the commit or squash of tasks. A significant concern is how the method to allocate/deallocate speculative storage interacts with the two modes of access. For the all-at-once approach, the physical management of speculative storage may be performed to match the logical management, favoring one or both modes of access. In contrast, for the on-demand approach, the physical management of speculative storage likely has no correspondence with the logical management as it is done *a priori*, favoring neither mode of access.

3.3 Ordering

The purpose of ordering is to *disambiguate* loads and stores such that their execution abides by the memory data dependences of the program. The appearance that must be provided for loads and stores of a particular memory address may described as follows: (i) the value read by a load is the value written by the logically preceding store, and (ii) the value written by a store is the value that persists until the logically succeeding store. Conceptually, a load checks the speculative storage of earlier tasks via a backward scan in the direction of the head to ensure the read condition; a store checks the speculative storage of later tasks via a forward scan in the direction of the tail to ensure the write condition. As such a scan may be a serial process, its timing impact on the execution of loads and stores is a concern. Two orthogonal techniques that may be used to reduce the timing impact associated with ordering are (i) to avoid performing the ordering on every load and store, and (ii) to speedup performing the ordering on loads and stores for which it cannot be avoided.

3.4 Transfer

The issue of transfer involves conveying state from one place to another (physically or logically). This transfer may occur due to (i) the execution of loads and stores, or (ii) the commit or squash of tasks. The transfers due to the execution of loads and stores are to ensure the production and consumption of values according to the memory data dependences of the program. As speculative loads may depend on speculative stores, some means must be provided to transfer data between such instructions in the active window via speculative rather than architectural storage. The transfers due to commits ensure that architectural storage is properly updated from loads and stores in speculative storage; whereas, the transfers due to squashes ensure that architectural storage is not improperly updated by loads and stores in speculative storage. A task does not yield its PU on a commit or squash until the full transfer of its speculative loads and stores. A means must be provided to perform this action in an expeditious manner so that a new task may be invoked.

4 Axes of the Design Space

The fundamental axes that characterize the data memory design space are its composition and its organization. Composition determines whether the storage for speculative and architectural versions are *separate* or *aggregate*. Organization determines whether the storage for speculative and architectural versions is *shared* or *private* between the various PUs.

4.1 Composition

The choice between separate and aggregate composition presents different trade-offs with respect to handling the overhead of disambiguation. Separate composition fixes the amount of storage of each type. As a result, speculative may not displace architectural storage to adjust to the needs of execution and to avoid the exhaustion of speculative storage which otherwise leads to stalls or squashes. Nevertheless, the storage of each type may be optimized for its particular role when separate: different capacity, block size, associativity, tagging, etc as needed. Aggregate composition incurs some overhead due to the tagging required to distinguish between the distinct types. Likewise, additional overhead in the form of extra state for disambiguation burdens all storage, though it is relevant only for speculative and not for architectural versions. In addition, for straightforward designs, separate composition implies transfer of data from speculative to architectural storage on a commit and simple annulling of speculative storage on a squash; whereas aggregate composition implies tagging of data on commits and squashes without any data transfer.

4.2 Organization

The choice between shared and private organization presents different trade-offs in terms of handling the bandwidth and the latency of data memory access. Private organization naturally offers high bandwidth and low latency due to its proximity and dedicated fast, wide access paths but shared organization may use the plethora of well developed high bandwidth techniques like multi-porting, interleaving and multi-level designs. The private approach poses the well-known and complex problem of how to maintain coherence, in a manner that is transparent to a running program, among multiple copies of storage. For a small number of PUs, coherence protocols based on a snooping bus [9] have been well studied. Much effort has been devoted to the minimization of the overhead traffic required of such schemes as the bus is often a performance bottleneck.

5 Data Memory Alternatives

In order to address a broad spectrum of solutions rather than focus on a narrow band of the design space, we consider two approaches to perform the required functionality. We consider a known scheme, the *address resolution buffer* [3], and a novel scheme, the *time-sequence cache*. Moreover, we consider *centralized* and *distributed* designs for each.

5.1 Address Resolution Buffer

In terms of the axes of the design space, an address resolution buffer (ARB) provides a data memory that is composed of separate speculative and architectural storage. The ARB provides only speculative storage that is coupled with a data cache to provide complementary architectural storage. The idea behind the ARB is to maintain different versions (on a word granularity) as a compact unit such that speculative versions are readily discerned among one another yet need not be distinguished from architectural versions; versions are allocated all-at-once primarily on task basis and secondarily on an instruction basis.

This scheme may be organized as a centralized ARB (CARB) with shared storage for all PUs or a distributed ARB (DARB) with private storage for each individual PU. (We use the term ARB to refer in general to both designs, and CARB or DARB to refer in particular to one design.) The ARB uses state bits to track the state of different versions of a word (in a manner similar to hardware cache coherence for multiprocessors [5]). The CARB uses centralized version control logic to trigger finite state machine transitions. In contrast, the DARB uses distributed version control logic to serve the same function.

5.1.1 Structure

An ARB is similar to a normal cache, except that a line is termed a row, as it contains storage for a fixed number of versions for an address to which it is assigned; it may be direct-mapped, set-associative, fully-associative, etc where each set contains a number of rows. Each row of an ARB consists of an address tag, multiple data items (each a single word), and sets of state bits matched one per data item. The state bits consist of a *valid bit* (V) as well as *load and store bits* (L and S) used to determine if the version is valid and if loads and/or stores have been performed on the word (tracking clean and dirty characteristics). In addition, a *mirror bit* (M) exists as a special state bit for the DARB but not the CARB (and is described below). Any speculative version that is committed or squashed is always removed immediately from the ARB. This action is a requirement for the ARB, since speculative and architectural versions must not coexist.

Figure 2(a) shows the overall picture of the CARB, while Figure 2(b) shows a row. A crossbar connects PUs to interleaved banks of storage. Each bank consists of an ARB coupled to a data cache which are connected to the next level of the data memory by a pipelined bus. With a centralized storage, different versions of a word are readily accessible. Figure 3(a) shows the overall picture of the DARB, while Figure 3(b) shows a row. Each PU is connected to a private copy of storage that consists of an ARB coupled to a data cache. The private ARBs are connected to one another by a broadcast bus. The private data caches are connected to next level of the data memory by a pipelined bus. With a distributed storage, different versions of a word are not readily accessible, since each version is private. To remedy this situation, each PU mirrors the stores of other PUs as seen on the broadcast bus. Though a row of the DARB appears similar to a row of the CARB, it is not the same since other data items in the row are not actual versions. Instead, other data items are store mirrors, as indicated by a mirror bit rather than valid, load, and store bits for the data items.

5.1.2 Operation

The operation of the ARB may be defined in terms of (i) events, (ii) states, and (iii) transitions betweens states due to events. In general, the operation is concerned with what happens to versions of a word on events. In particular,

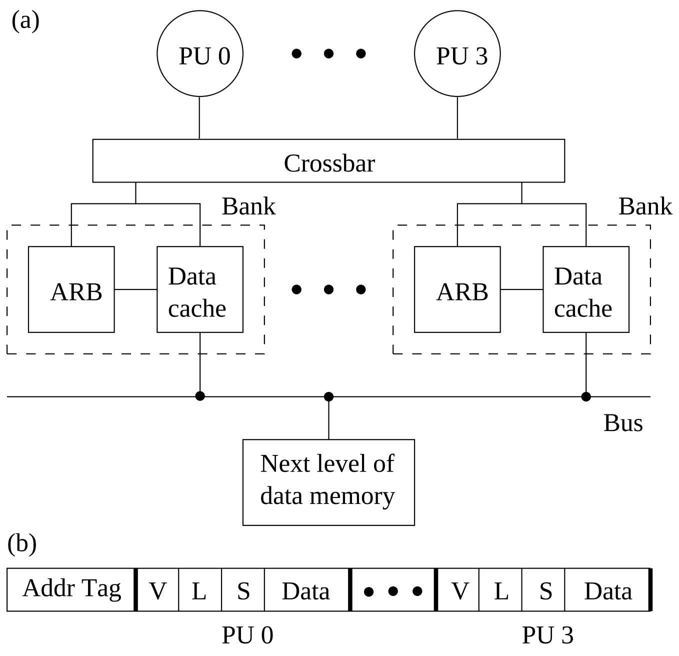


Figure 2: Structure of the centralized address resolution buffer (CARB).

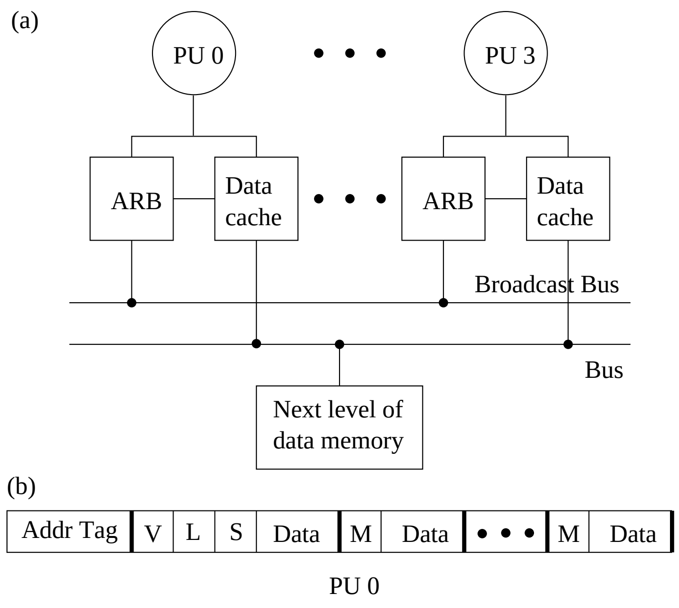


Figure 3: Structure of the distributed address resolution buffer(DARB).

the operation is concerned with how a version is effected by events from the PU with which the version is associated and by events from other PUs. A PU event is directed at a bank of the shared storage for a CARB or at the private storage for a DARB. In some cases, the event may act on a particular version without an effect on other versions. However, in the cases where other versions may be involved, version control logic (VCL) dictates the proper course of action.

The role of the VCL is to handle the disambiguation of loads and stores. In response to an event, the state bits of all relevant versions are provided to the VCL which in turn triggers state changes among these versions and delivers control signals to the relevant PUs. In the CARB, the VCL exists inside the shared storage, where it manipulates versions in a centralized manner. In the DARB, the VCL resides inside the private storage, where it performs the same function in a distributed manner via the broadcast bus. The actual details of the VCL depend on the specifics of particular centralized or distributed designs and is not a real concern.

5.1.3 Versions and Ordering

The VCL is able to disambiguate a load in a straightforward manner among all versions in the active window. The correct version for a load can be found using the load and store bits of each version and the implicit cyclic order imposed by the head and the tail. For the CARB, the VCL inspects versions in a backward scan from the accessed version to the head to find the nearest store which supplies the data. For the DARB, each private VCL performs the same process, using the mirrored stores (previously broadcast).

The VCL is able to disambiguate a store in a straightforward manner (as for a load) among all versions in the active window. A memory data dependence violation caused by later loads that have been performed before an intervening store can be found using the load and store bits of each version and the implicit cyclic order imposed by the head and the tail. For the CARB, the VCL inspects versions in a forward scan from the accessed version to the tail. Any load encountered before an intervening store identifies a version that has a memory data dependence violation. For the DARB, each private VCL performs the same process, using the mirrored stores (previously broadcast).

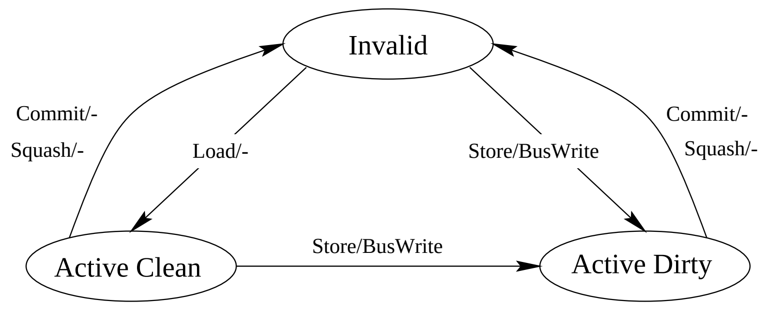
5.1.4 Events

A PU access to the CARB or DARB may correspond to a load, store, commit, or squash PU event. A load or store event corresponds to the instruction oriented mode of access described in section 3.2 and involves versions for all PUs that match on the address tag of the load or store. In contrast, a commit or squash event corresponds to the task oriented mode of access described in section 3.2 and involves versions for a particular PU regardless of address. In either case, an involved version exists in a shared row that is accessed by all PUs for the CARB and in a private row that is accessed by only one PU for the DARB. In addition to PU events, a PU access for the DARB unlike the CARB, may generate a related bus event. A bus write event occurs for a store PU event and ensures that private ARBs remain consistent with one another. A bus event caused for a store allows it to be broadcast and mirrored in each private ARB. A commit or squash PU event also effects other private ARBs, but this effect pertains to a mirror of some other version rather than an actual version per se.

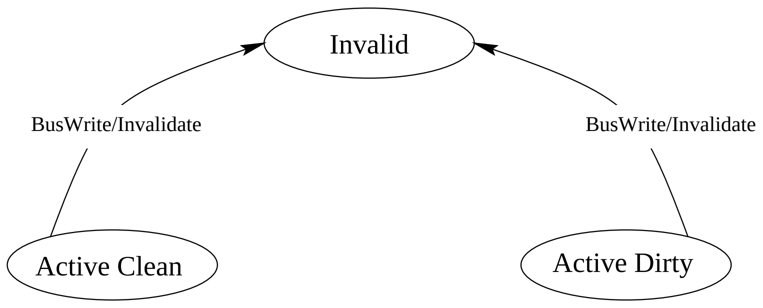
5.1.5 States

To keep this description of version states concise, we group all physical states (many) along with the relevant transitions into categories which represent logical states (few). Thus, the operation of the ARB is given in terms of these logical states rather than the actual physical states – transitions are from categories of states to categories of states instead of from individual states to individual states. A version may fall in one of three categories (heretofore referred to as states): Invalid, ActiveClean, or ActiveDirty. Below is a description which gives the meaning of the states which a version may occupy.

An Invalid version is one for which the tags, state, and data of the word has no context. A version in this state in not considered a part of the speculative storage, as it is not involved in the operation of the ARB until it is accessed (and made valid). An ActiveClean version is one for which a task in the active window has performed only loads (unmodified version). An ActiveDirty version is one for which a task in the active window has performed stores and possibly loads (modified version). Moreover, a version in either of these states is considered part of the speculative storage, as it is involved in the operation for disambiguation amongst other versions.



(a) Version Tag = PU



(b) Version Tag <> PU

Figure 4: State transitions for ARB.

5.1.6 Transitions

The transitions between version states are caused by the events and by the directives of the VCL as depicted in Figure 4 (self transitions are not shown for conciseness).

Consider the top of the state transition diagram in Figure 4(a). The arcs between states are given in the form of [PU event]/[bus event] (where a – means no corresponding event). The transitions in this half of the diagram indicate the effects on a version for accesses performed by its PU. A load or store by a PU always transits a version to ActiveClean or ActiveDirty indicating it is contained in the active window of execution. Likewise, a commit or squash by a PU only effects a version in ActiveClean or ActiveDirty. A version always goes to Invalid on commit or squash. In addition, a commit of an ActiveDirty version requires a writeback from speculative to architectural storage.

Consider the bottom of the state transition diagram in Figure 4(b). The arcs between states are given in the form of [bus event]/[VCL response] (where a – means no corresponding event). The arcs between states are given in the form of bus/VCL event (where a – means no corresponding event).

The transitions in this half of the diagram indicate the effects on a version for accesses performed by other PUs (a bus event for the DARB which sees only its private version but not for CARB which sees all shared versions). A broadcast store performed by some other PU, leaves a version in its existing state, unless the VCL detects a memory data dependence violation and delivers an invalidate signal, in which case a version always transits to Invalid.

5.1.7 Replacement and Writeback

A load or store cause a miss if no valid corresponding version exists. In the CARB, a load or store miss is handled individually by a shared ARB bank. In the DARB, a load miss is handled individually by a private ARB, but a store miss must be handled collectively by each private ARB in unison (regardless of which PU missed). The data to be supplied for a miss comes from the data cache which complements the ARB. The row enters the ARB with the accessed version in ActiveClean or ActiveDirty, as the case may be. While handling the miss is straightforward, selecting a row to replace and the possible writeback may be involved.

The best option for a row to replace is one in which all versions are Invalid. If a row has all Invalid versions, it can be replaced with no writeback. The only other option is to replace a row in which some versions are ActiveClean or ActiveDirty, but this action can only be performed under restricted conditions. When the PU that caused the miss is not the head, it may be stalled until a row with all Invalid versions becomes available or until it becomes the head (whichever occurs first). When the PU that caused the miss is the head, it may writeback versions in the row associated with it or it may squash other PUs to invalidate other versions in the row. (This approach guarantees forward progress, since a PU may always replace rows as the head; and as the head, a PU at some point yields to the next PU to be the head.)

5.2 Time-Sequence Cache

In terms of the axes of the design space, a time-sequence cache (TSC) provides a data memory that is composed of aggregate speculative and architectural storage. The TSC does not need complementary storage like an ARB since it provides both types. The idea behind the TSC is to maintain different versions (on a cache line granularity) such that both speculative and architectural versions coexist but can be readily distinguished; versions are allocated on-demand as per the execution of a load or store in the active window.

If the TSC only contained speculative versions, then it would be the same as the ARB. However, both committed and squashed versions are allowed to remain in the TSC to avoid the potential overhead of removing them as in the ARB. As a result, a mix of both speculative and architectural versions may be present. In particular, there may be many "architectural" versions (each of which corresponds to the true architectural state at different points in time) in contrast to a normal cache (which has only one version of architectural state). Among these versions only one actually corresponds to the most recent version of the true architectural state at any given point in time. Therefore, it is necessary to distinguish this version from the others.

The TSC uses state bits to track the state of different versions of a line (in a manner similar to hardware cache coherence for multiprocessors). This scheme may be organized as a centralized TSC (CTSC) with shared storage for all PUs or a distributed TSC (DTSC) with private storage for each individual PU. (We use the term TSC to refer in general to both designs, and CTSC or DTSC to refer in particular to one design.) The CTSC uses centralized version control logic to trigger finite state machine transitions. In contrast, the DTSC uses snooping version control logic (like snooping bus-based protocols found in ubiquitous SMPs) to serve the same function.

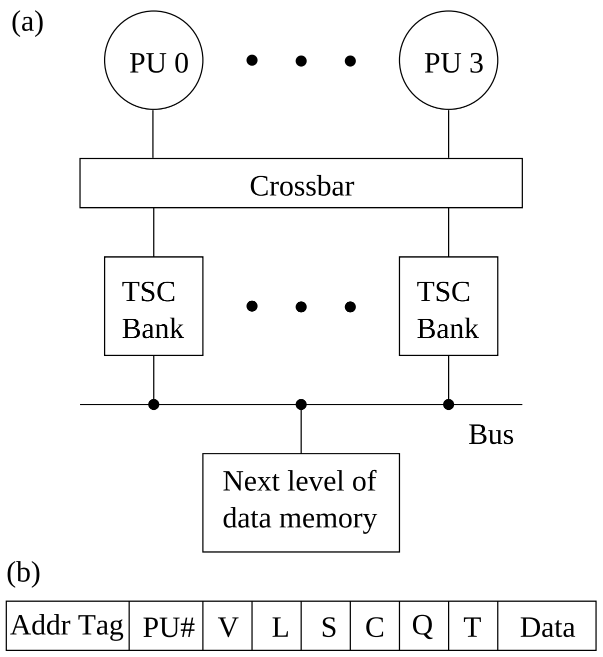


Figure 5: Structure of the centralized time-sequence cache (CTSC).

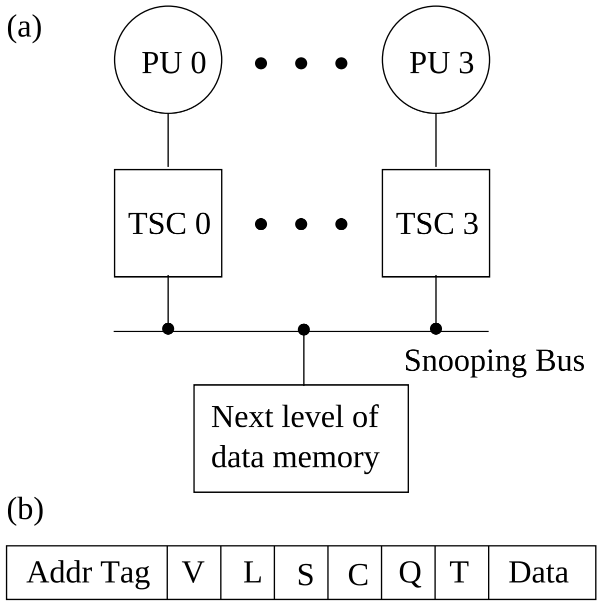


Figure 6: Structure of the distributed time-sequence cache (DTSC).

5.2.1 Structure

A TSC is much like a normal cache, but with an extended tag space to maintain different versions of an address; it may be direct-mapped, set-associative, fully-associative, etc where each set contains a number of lines. Each line of a TSC consists of an address tag, possibly a PU tag, a single data item (usually multiple words), and a single set of state bits matched to the data item. The state bits consist of a *valid bit* (V) as well as *load and store bits* (L and S) used to determine if the line is valid and if loads and/or stores have been performed on the line (tracking clean and dirty characteristics) as in the ARB. In addition, a *commit bit* (C) and a *squash bit* (Q) distinguish committed and squashed versions, while a *token bit* (T) identifies the most recent version in existence among all versions.

Figure 5(a) shows the overall picture of the CTSC, while Figure 5(b) shows a line. The lines of the CTSC contain an explicit PU tag to distinguish different versions in shared storage. A crossbar connects PUs to interleaved banks of storage. Each bank consists of a TSC connected to the next level of the data memory by a pipelined bus. With a centralized storage, different versions of a line are readily accessible. Figure 6(a) shows the overall picture of the DTSC, while Figure 6(b) shows a line. The lines of the DTSC are private to a PU and hence have an implicit PU tag. Each PU is connected to a private copy of storage that consists of a TSC. The private TSCs are connected to one another and to the next level of the data memory by a pipelined snooping bus. With a distributed storage, different versions of a line are not readily accessible, since each version is private.

5.2.2 Operation

The operation of the TSC may be defined in terms of (i) events, (ii) states, and (iii) transitions betweens states due to events. In general, the operation is concerned with what happens to versions of a line on events. In particular, the operation is concerned with how a version is affected by events from the PU with which the version is associated and by events from other PUs. A PU event is directed at a bank of the shared storage for a CTSC or at the private storage for a DTSC. In some cases, the event may act on a particular version without an effect on other versions. However, in the cases where other versions may be involved, version control logic (VCL) dictates the proper course of action.

The role of the VCL is to handle the disambiguation of loads and stores. In response to an event, the state bits of all relevant versions are provided to the VCL which in turn triggers state changes among these versions and delivers control signals to the relevant PUs. In the CTSC, the VCL exists inside the shared storage, where it manipulates versions in a centralized manner. In the DTSC, the VCL resides on the pipelined snooping bus, where it performs the same function. The actual details of the VCL depend on the specifics of particular centralized or distributed designs and is not a real concern.

5.2.3 Versions and Ordering

The VCL is able to disambiguate a load in a straightforward manner among all versions in the active window. However, this approach is not sufficient to support disambiguation in the presence of squashed and/or committed versions (no longer associated with the active window) that may exist in a TSC. The problem is that the order between versions is lost without the relationship of the head and tail that existed when the versions were active. The solution to this problem is to impose an invariant on the coexistence of versions which permits disambiguation to be performed only on active versions as in the ARB. The invariant is that each time a new version is created (first load or store to the address for a task in the active window), only versions for other active tasks may coexist.

This condition implies that the creation of a new version causes any committed or squashed versions to be removed from the TSC. Moreover, it requires (i) that all committed and squashed versions be identified, (ii) that all squashed versions be removed from the TSC, and (iii) that all committed versions be removed from the TSC with a writeback as well for most recent committed version. The state bits of the TSC line provide the support for this solution. The commit bit and the squash bit identify the committed and squashed versions. The token bit identifies the most recent version (only one version can have this bit set). Note, a speculative version with the token holds the token if it is squashed until a subsequent creation of a new version recovers the token; at which time, the token is transferred to the most recent existing version.

The VCL is able to disambiguate a store in a straightforward manner (as for a load) since all relevant versions are in the active window of execution due to the invariant. A memory data dependence violation for later loads that have been performed before an intervening store can be found using the load and store bits for each version in a manner similar to the ARB. For the CTSC, the VCL inspects versions in what amounts to a forward scan from the accessed version to the tail. For the DTSC, the VCL performs the same process, using the pipelined snooping bus to see the other versions.

5.2.4 Events

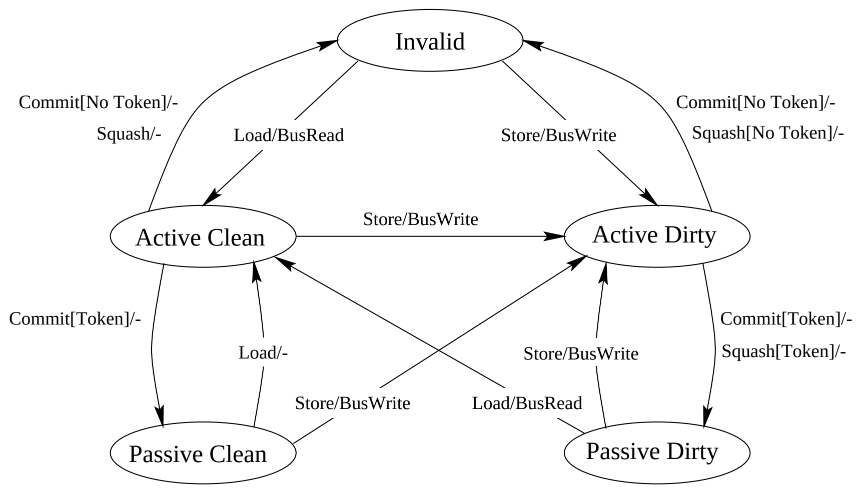
A PU access to the CTSC or DTSC may correspond to a load, store, commit, or squash PU event. A load or store event corresponds to the instruction oriented mode of access described in section 3.2 and involves versions for all PUs that match on the address tag of the load or store. In contrast, a commit or squash event corresponds to the task oriented mode of access described in section 3.2 and involves versions for a particular PU regardless of address. In either case, an involved version exists in shared row that is accessed by all PUs for the CTSC and in a private row that is accessed by only one PU for the DTSC. In addition to PU events, a PU access for the DTSC unlike the CTSC, may generate a related bus event, either a bus read, a bus write, or a bus writeback. A bus read event occurs for a load or store PU event that causes a TSC miss. A bus write event occurs for a store PU event that may cause versions to become inconsistent. A bus writeback event occurs if a dirty versions must be written back from the TSC.

5.2.5 States

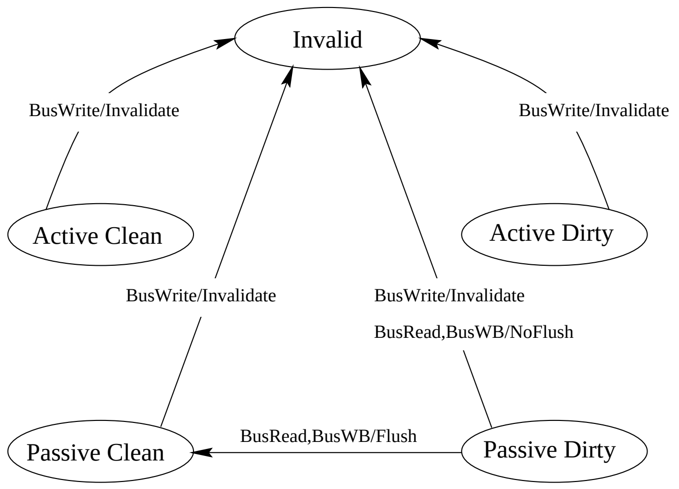
As before, we group all physical states (many) along with the relevant transitions into categories which represent logical states (few). Thus, the operation of the TSC is given in terms of these logical states rather than the actual physical states – transitions are from categories of states to categories of states instead of from individual states to individual states, just as for the ARB. A version may fall in one of five categories (heretofore referred to as states): Invalid, ActiveClean, ActiveDirty, PassiveClean, or PassiveDirty. Below is a description which gives the meaning of the states which a version may occupy.

An Invalid version is one for which the tags, state, and data of the line has no context. A version in this state is not considered a part of the speculative or architectural storage, as it is not involved in the operation of the TSC until it is accessed (and made valid). An ActiveClean version is one for which a task in the active window has performed only loads (unmodified version). An ActiveDirty version is one for which a task in the active window has performed stores and possibly loads (modified version). Moreover, a version in either of these states is considered part of the speculative storage, as it is involved in the operation for disambiguation amongst other versions. It is important to realize that these states of the TSC are precisely the same as those of the ARB.

Nevertheless, the TSC introduces additional states in order to maintain coexistent speculative and architectural versions. A PassiveClean version is one for which a task previously in the active window had performed only loads (unmodified version); this line is a clean copy of a line that was either committed or squashed, corresponding to a delayed commit or delayed squash of the version. A PassiveDirty version is one for which a task previously in the active window performed stores and possibly loads (modified version); this line is a dirty copy of a line that corresponds to a delayed commit or delayed squash as in PassiveClean, but unlike in PassiveClean, that may need to be written back to the next level of the data memory. A version in either of these states is considered part of the architectural storage.



(a) Version Tag = PU



(b) Version Tag <> PU

Figure 7: State transitions for TSC.

5.2.6 Transitions

The transitions between version states are caused by the events and by the directives of the VCL as depicted in Figure 7 (self transitions are not shown for conciseness). It is worth pointing out that the states and transitions of the ARB as depicted in Figure 4 are a subset of those of the TSC.

Consider the top state transition diagram of Figure 7(a). The arcs between states are given in the form of [PU event]/[bus event] (where a – means no corresponding event). The transitions in this half of the diagram indicate the effects on a version for accesses performed by its PU. A load or store by a PU always transits a version to ActiveClean or ActiveDirty indicating it is contained in the active window of execution. Likewise, a commit or squash by a PU only effects a version in ActiveClean or ActiveDirty. An ActiveClean version goes to the PassiveClean with the token and to Invalid without the token on commit. An ActiveClean version always goes to Invalid on squash. An ActiveDirty version goes to the PassiveDirty state with the token (to hold the token for recovery) or to the Invalid state on a squash or commit.

Consider the bottom state transition diagram of Figure 7(b). The arcs between states are given in the form of [bus event]/[VCL response] (where a – means no corresponding event). The transitions in this half of the diagram indicate the effects on a version for accesses performed by other PUs. In contrast to the ARB, the VCL of the TSC plays a pivotal role in directing state transitions for other PU events in the CTSC or rather bus events in the DTSC. In addition to the invalidate directive in the ARB, the VCL may provide the flush, acquire, and supply directives in the TSC. The invalidate command invalidate a version in the TSC. The flush command indicates that a particular committed version must be written back to the next level of the data memory. The acquire command reconciles the token for the most recent version. The supply command indicates the data of a version must provided to some other PU (and/or private TSC).

Regardless of the version state and event, an invalidate directive causes a version to go to Invalid. A load performed by some other PU causes a PassiveDirty version to go to PassiveClean with a flush or to Invalid without a flush. A store performed by some other PU causes a PassiveDirty version to go to Invalid (as already said) with a flush for a version with the token and with a flush for other versions. On writeback, a PassiveDirty version goes to PassiveClean with a flush or to Invalid without a flush. Note that the other VCL directives, acquire and supply, are not shown in the diagram since no state transition occurs in connection with them. Nevertheless, on each other PU event, the token must be reconciled and the data must be supplied as directed by the VCL.

5.2.7 Replacement and Writeback

A load or store cause a miss if no valid corresponding version exists. In the CTSC, a load or store miss is handled by a shared TSC bank. In the DTSC, a load or store miss is handled by a private TSC. The data to be supplied for a miss comes from some other version or from the next level of the data memory. The line enters the TSC in the ActiveClean state. While handling the miss is straightforward, selecting a line to replace and the possible writeback may be involved.

The options to choose a version to replace (in order of preference among possible versions) are Invalid, PassiveClean, PassiveDirty, and ActiveClean or ActiveDirty versions. If a line is Invalid, it can be replaced with no writeback. If a line is PassiveClean, it can be replaced with no writeback. If a line is PassiveDirty, it can be replaced, but if it has the token, there must be a writeback before it can be replaced. If a line is Active, it can be replaced, but under restricted conditions.

When the PU that caused the miss is not the head, it may be stalled until version that is neither ActiveClean nor ActiveDirty becomes available or until it becomes the head (whichever occurs first). When the PU that caused the miss is the head, it may writeback versions of lines associated with it or it may squash other PUs to invalidate other versions of lines. (This approach guarantees forward progress, since a PU may always replace lines as the head; and as the head, a PU at some point yields to the next PU to be the head.)

6 Experimental Evaluation

We have performed an experimental evaluation of opposite ends of the spectrum of data memory alternatives we have considered in the previous section – the CARB and the DTSC. Our objective in this evaluation is to show the effects of hit latency on overall performance and to compare equivalent (in terms of overall storage) designs of the CARB and DTSC.

6.1 Methodology

All of the results in this paper have been collected on a simulator that faithfully models a multiscalar processor. The simulator accepts annotated big endian MIPS instruction set binaries produced by the multiscalar compiler, a modified version of gcc. In order to provide results which reflect reality with as much accuracy as possible, the simulator performs all of the operations of a multiscalar processor and executes all of the program code, except system calls, on a cycle-by-cycle basis. (The system calls are handled by trapping to the operating system of the simulation host.)

6.2 Configurations

The multiscalar processor used in the experiments is a 4 PU configuration in which each PU is based on the RUU [11] and has been configured with 2-way out-of-order issue characteristics. A PU executes instructions on its own collection of pipelined functional units (2 simple integer FU, 1 complex integer FU, 1 floating point FU, 1 branch FU, and 1 memory FU) according to its class. The unidirectional point-to-point ring connecting the register files [2] of the PUs imposes a 1 cycle communication latency between units and matches the ring width to the issue width of the PU. Each PU has its own instruction cache with 32k of 2-way set-associative storage in 44 byte blocks. An access hit returns 4 words in a hit time of 1 cycle with an addition penalty of 10+3 cycles, plus any bus contention, on a miss.

The global sequencer maintains a 1024 entry 2-way set associative cache of task descriptors. The control flow predictor of the global sequencer uses a dynamic path based scheme which selects from up to 4 task targets per prediction and keeps 7 path histories XOR-folded into a 15-bit path register [6]. 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 is a 2-bit counter and a 32-bit address. The control flow predictor includes a 64 entry return address stack.

For the CARB, we have used 8 banks of interleaved (on the data cache line address) shared storage connected to the PUs by a crossbar. Each bank is composed of a fully-associative ARB with 32 rows and of a data cache with 4k or 8k of directed-mapped storage in 16-byte lines (32k or 64k total). Both loads and stores are non-blocking

Benchmark	CARB		DTSC	
	32KB	64KB	4x8KB	4x16KB
compress	0.114	0.096	0.148	0.134
espresso	0.026	0.011	0.026	0.012
gcc	0.025	0.016	0.059	0.047
sc	0.039	0.036	0.058	0.046
xlisp	0.028	0.025	0.024	0.015

Table 1: Miss Ratios for CARB and DTSC.

Benchmark	4x8KB	4x16KB
compress	0.315	0.310
espresso	0.183	0.170
gcc	0.238	0.229
sc	0.191	0.177
xlisp	0.303	0.295

Table 2: Snooping Bus Utilization for DTSC

with 4 MSHRs [7] per bank. Disambiguation is performed at the byte-level. An access has a hit time of 1, 2, 3, or 4 cycles, with an additional penalty of 10 cycles for a miss supplied by the next level of the data memory (plus any bus contention). All access to the next level of the data memory are handled by a 4-word pipelined bus.

For the DTSC, we have used private storage at each PU, connected to one another and the next level of the data memory by a 4-word pipelined snooping bus where a bus transaction requires 3 or 4 PU cycles. Each PU has its own private TSC with 8k or 16k of 2-way set-associative storage in 16 byte lines (32k or 64k total). Both loads and stores are non-blocking with 8 MSHRs per TSC. Disambiguation is performed at the byte-level. An access has a hit time of 1 cycle, with an additional penalty of 10 cycles for a miss supplied by the next level of the data memory (plus any bus contention).

6.3 Benchmarks

We used the following programs as benchmarks from the SPECint92 suite with inputs given in parentheses: compress (in), espresso (ti.in), gcc (integrate.i), sc (loada1), and xlisp (7 queens).

6.4 Experiments

In Figures 8, 9, 10, 11, and 12 we present the instructions per cycle (IPC) for a multiscalar processor configured with a CARB and a DTSC. (The configurations keep total TSC storage and total ARB/data cache storage roughly equivalent, since the amount of ARB storage is rather modest compared to its data cache.) For the CARB, we fix the total ARB size at 256 rows (from all banks) and vary the total data cache size between 32k and 64k of storage (from all banks) with hit latencies ranging from 1 through 4 cycles. For the DTSC, we vary the total TSC size between 32k and 64k and fix the hit latency at 1 cycle. In Table 1 we present the miss rates for the CARB and the DTSC configurations. (For the DTSC, an access is only counted if it is supplied data by the next level of the data memory; TSC-to-TSC transfers are not counted as misses.) In Table 2 we present the bus utilization of the DTSC configurations. From these preliminary experiments, we make three observations: (i) the hit latency of data

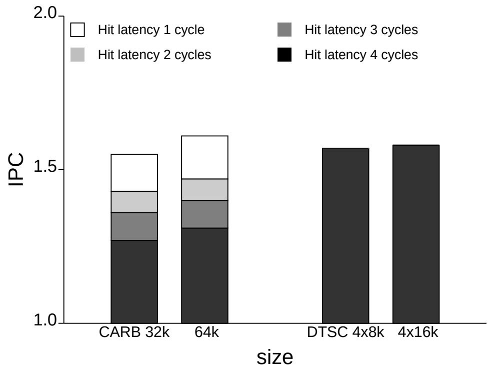


Figure 8: IPCs for Compress.

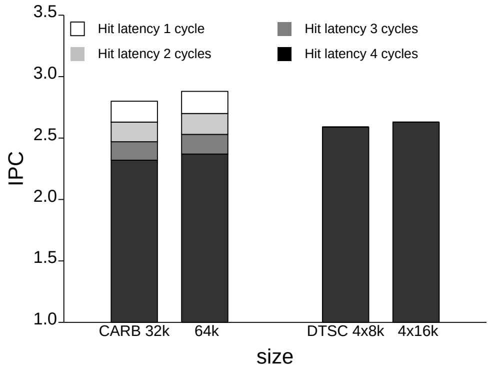


Figure 9: IPCs for Espresso.

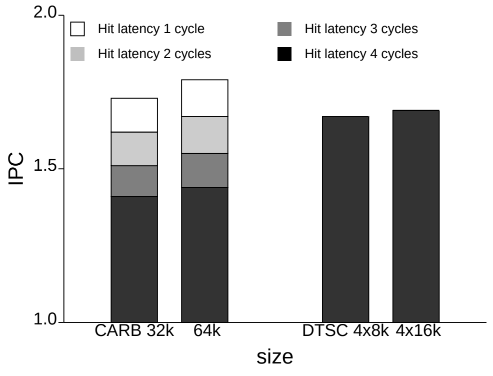


Figure 10: IPCs for Gcc.

memory significantly impacts performance, (ii) the CARB and the DTSC trade-off hit latency for hit rate and vice-versa to achieve performance, and (iii) for the same total TSC and data cache storage (not counting the ARB storage which is transient), the DTSC performs better than the CARB with a hit latency of 3 or more cycles. The IPC numbers of the CARB with hit latency of 1 cycle, as shown in the Figures 8, 9, 10, 11, and 12, give a bound on the performance achieved by multiscalar configurations designed to improve hit latency. The graphs in these figures show that even for a 64k CARB performance improves in the range of 21% to 40% when decreasing the hit latency

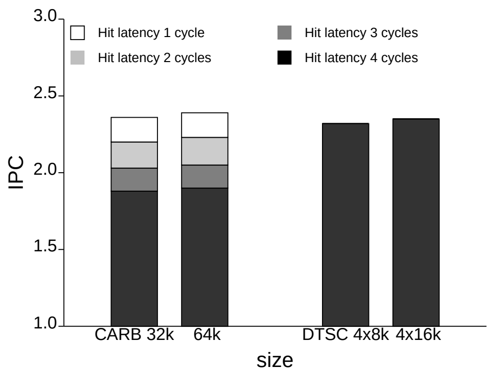


Figure 11: IPCs for Sc.

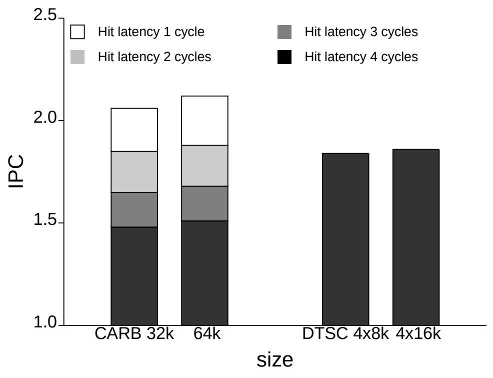


Figure 12: IPCs for Xlisp.

from 4 cycles to 1 cycle. This improvement indicates that techniques (like the DARB and the DTSC) to improve hit latency are an important factor in increasing overall performance (even for a latency tolerant processor like a multiscalar processor).

Comparing the same total amounts of storage, the distribution of storage for the DTSC produces higher miss rates than for the CARB (xlisp is an exception). In spite of this handicap, the IPC achieved by the DTSC for xlisp is better than that achieved by the CARB with hit latency of 3 or more cycles. In the case of gcc, compress, and sc, the IPCs indicate that the DTSC outperforms the CARB with hit latency of 2 or more cycles. Though, for xlisp, both schemes perform nearly as well, the trend is reversed for espresso. The increase in the miss rates may be attributed to two factors. First, the distribution of storage in the DTSC causes reference spreading [8] across multiple TSCs leading to an increase in misses. Though multiple accesses in close proximity go to only one ARB and/or data cache in the CARB, these accesses may be spread across different TSCs of the DTSC destroying the locality of the accesses. Moreover, due to temporal locality, the same data may be replicated in each TSC of the DTSC, essentially reducing its size. Second, the fine-grain sharing of data between multiscalar tasks causes cache lines to move from one TSC to another. Such fine-grain communication may increases the number of misses as well. The evidence for such communication is the fairly high bus utilizations of the DTSC ranging from 17% to 31%.

7 Discussion

We discuss aspects related to precise interrupts in a uniprocessor system and sequential consistency in a multiprocessor system for a data memory design in general and briefly describe how these aspects can be provided for the memory designs we have considered, the ARB and the TSC.

7.1 Precise Interrupts

We classify interrupts into two classes: program interrupts (i.e., traps or exceptions) and external interrupts [10]. In a multiscalar processor, an exception may be received by speculative as well as non-speculative tasks. An exception received by the non-speculative task at the head can be made precise by dealing with it in the same fashion as a superscalar processor. However, a commit must be simulated to flush the architectural state buffered in data memory before the exception is serviced. If a speculative task receives an exception, we may stall the execution of the task at the faulting instruction until either the task becomes the head task or is squashed. This approach ensures that data memory is not corrupted with data from instructions beyond the faulting instruction in the task. If the task is squashed, the exception is spurious and need not be handled. If the task becomes non-speculative, the exception can be handled as already described. To handle external interrupts, all the speculative tasks may be squashed and the head task stopped in a precise manner at the instruction that is associated with the external interrupt. The program can be restarted at the excepting instruction since data memory holds the architectural state at the faulting instruction.

7.2 Sequential Consistency

The most common memory model (often implicitly) assumed by designers of parallel programs is sequential consistency. We describe a working solution to provide sequential consistency for a multiprocessor composed of multiscalar nodes. Other more aggressive solutions are possible, but we do not address them here. The data memory alternatives described in this paper merge stores from a PU to the same memory location and do no track the order in which stores to different locations have been performed. This loss of order could lead to violations of sequential consistency. To avoid such violations, each load or store must be globally committed [1] in program order to provide a total order for memory accesses made by a parallel program. This condition implies that no load or store can be retired from the reorder buffer of a PU until the associated task becomes a non-speculative task at the head and until all previous instructions have globally committed. Moreover, a store cannot be performed to data memory until it is ready to be globally committed. We employ the strategy used in the R10000 [13, 4] to allow the speculative execution of loads and detect a violation of sequential consistency, in which case all instructions from the offending load instruction must be squashed [1]. In addition, a store is allowed to proceed to the ARB/TSC once it is ready to be retired from the PU reorder buffer even though the task might be speculative. Once the task becomes non-speculative, we revisit loads and stores in program order and commit stores to the architectural state, bypassing the ARB/TSC. Note that the speculatively executed loads and stores may overlap their communication overheads with the execution of other instructions.

[1] On an invalidation received by a load instruction in the non-speculative task, a squash signal must be sent to the ARB/TSC to squash all relevant instructions, including those of the head PU.

8 Summary

We addressed the problem of performing ambiguous loads and stores in a vast active window of instructions built using using multiple program counters. In the context of a data memory alternatives for multiscalar processors, we explored the key issues of allocation/deallocation, access, ordering, and transfer with regard to speculative and architectural storage. We partitioned the design space on the basis of composition, whether the storage for speculative and architectural versions are *separate* or *aggregate*, and on the basis of organization, whether the storage for speculative and architectural versions is *shared* or *private* between the various PUs. We presented two approaches to perform disambiguation of loads and stores, the address resolution buffer (ARB) and the time-sequence cache (TSC), both of which reduce the extent to which elements are searched in this process. We explored a broad spectrum of solutions in the form of centralized and distributed schemes for the ARB and the TSC. Moreover, we showed experimental evidence from a simulation of a multiscalar processor with a centralized address resolution buffer (CARB) and a distributed time-sequence cache (DTSC) that hit latency is an important factor (even for a latency tolerant processor like a multiscalar processor) in determining performance and that distributed schemes trade-off hit rate for hit latency to achieve performance. In our investigation, we have attempted to unify the four quadrants of the design space into a common framework by pointing out that designs differ only in details such as the number of states and the version control logic.

Acknowledgements

This work was supported in part by NSF Grants CCR-9303030 and MIP-9505853, by U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346, by a research grant from Intel Corp., and by an equipment donation from Sun Microsystems. 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] Sarita V. Adve and Mark D. Hill. Implementing sequential consistency in cache-based systems. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages 47–50, 1990.

[2] Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. The anatomy of the register file in a multiscalar processor. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 181–190, 1994.

[3] 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.

[4] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 255–364, 1991.

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

[7] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87, 1981.

[8] 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.

[9] David A. Patterson and John L. Hennessy. *Computer Architecture A Quantitative Approach*, chapter 8, pages 635–755. Morgan Kaufmann Publishers, 1996.

[10] James E. Smith and Andrew R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 36–44, June 17–19, 1985.

[11] Gurindar S. Sohi. Instruction issue logic for high performance, interruptable, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[12] 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.

[13] K.C. Yeager. MIPS R10000 superscalar microprocessor. *Micro*, April 1996.