Fast Checkpoint/Recovery to Support Kilo-Instruction Speculation and Hardware Fault Tolerance

Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill, David A. Wood
Computer Sciences and Electrical Engineering Departments
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706

Abstract

The increased relative cost of accessing memory is encouraging processor designers to explore deeper uniprocessor speculation (e.g., with branch and value prediction) and consider multiprocessor speculation (e.g., on coherence message types and values). While some mechanisms have been proposed to support deep speculation using speculative multithreading, current mechanisms for conventional processors are not as good.

To support kilo-instruction speculation with conventional processors, this paper proposes *Multiversion Memory (MVM)*, a processor/memory interface that allows processors to create multiple versions of memory and recover to previous versions when necessary. In this paper, we develop an efficient implementation of MVM that uses a level one cache to keep recent speculative blocks (like a future file for memory), uses version buffers to keep old versions of blocks for which speculation is pending (like a memory history buffer), and leaves the level two cache (and beyond) unchanged (like a memory architectural file).

Concurrently, requirements for highly-available computers and manufacturing trends to deep-sub-micron design encourage techniques to mask transient faults (e.g., with error correcting codes and execution retry). Most current designs consider speculation and fault tolerance independently. Nevertheless, a second result of this paper is that MVM can provide support for both needs, perhaps making the use of hardware fault-tolerance more widespread.

Simple cost models with parameters from commercial workloads show that our implementation of MVM allows kilo-instruction speculation and fault tolerance that can recover faster (e.g., less than 273 vs. 362 cycles), uses recovery storage that is smaller (e.g., 5,356 bytes vs. 10,000 bytes), and has lower common-case overhead than other recently proposed schemes.

1 Introduction

Speculative execution is an important technique for improving computer system performance. This technique begins with a *prediction* of what work is likely to be needed soon. Then the work is performed *speculatively* so that it can be *committed* if the prediction is deemed correct or *aborted* otherwise. Most processors today, for example, perform branch predictions [35] and then execute instructions speculatively following those predictions. Furthermore, many future processors will make a wider use of prediction, in part to tolerate the increased relative time to access memory (i.e., the *memory wall* [32]). These speculations may be based on predicting values [21], various aspects of multiprocessor coherence message types or values [8, 18, 24], or new ideas not yet invented. Data value prediction is an example of a speculation technique with a large potential for performance gain, since it can hide the long latency of accessing memory, assuming the system can provide efficient mechanisms for speculative execution. As the opportunity cost of waiting for data to arrive or a condition to resolve increases, the potential benefits of speculative execution increase.

Along with good predictors, the twin challenges of implementing speculative execution are (i) keeping overhead low in the (hopefully) common case when speculations commit (i.e., the prediction is verified to have been correct), and (ii) minimizing the delay caused when a speculation aborts (i.e., the prediction is determined to have been incorrect). These costs depend on whether one recovers from aborts by restoring to a *checkpoint* of pertinent state made at or before the time of the prediction, squashing tentatively-performed speculative operations, using a log to *rollback* speculatively overwritten state, or some combination of these mechanisms. The MIPS R10000 [42], for example, checkpoints register maps and only tentatively performs memory stores.

One approach to deep speculation is *speculative multithreading* [4, 12, 15, 25, 37, 38]. With speculative multithreading, contiguous sequences from the dynamic instruction stream of a program's execution—called *speculative threads*—are distributed (often with compiler support) to processing elements within a processor. Speculative multithreading, however, is only one approach to deep speculation. In this paper, we focus on supporting deep speculation techniques that *do not rely* on speculative multithreading, such as those based upon value prediction [21] and multiprocessor coherence [8, 18, 24]. A specific example is false sharing speculation, where a processor speculatively uses data in its cache that another processor invalidated.

The challenge of supporting deep speculative execution grows rapidly with (a) the latency between when a prediction is made and when it is verified and (b) whether multiple predictions are in simultaneous use. The R10000 mechanisms, for example, depend on structures that must be associatively searched and updated on many processor cycles. Since the size of these structures is proportional to the number of active instructions, it is not viable to extend them to support a multiprocessor prediction that cannot be verified for 250 cycles (say 1000 instruction opportunities). Schemes such as Ranganathan et al.'s Speculative Retirement [29] and Gniady et al.'s SC++ [14] seek to extend the depth of speculation beyond that of the R10000 to narrow the performance gap between sequential consistency and weaker memory models, but these schemes are still not efficient for kilo-instruction speculation. This and other related work is discussed more fully in Section 2.

To efficiently support kilo-instruction speculation, this paper proposes a processor/memory interface called *multiversion memory (MVM)*[1] and develops one implementation of this interface. Multiversion memory allows a processor to create multiple versions of memory, commit versions that are no longer needed, and recover to previous versions if necessary. While MVM provides the clean abstraction shown in Figure 1 (for simplicity, we show a uniprocessor system), the challenge lies in its efficient implementation.

In this paper, we develop *one efficient implementation* of multiversion memory that we call MVM1. MVM1 works with a standard, speculative, out-of-order processor core that is augmented to occasionally checkpoint its non-memory state (e.g., program counter and registers) and to tag loads and stores with a version number. MVM1 is *not* tied to speculative multithreading. MVM1 consists of an augmented level-one (L1) cache

1. Multiversion memory gets its name from the superficially similar software technique of using multiple versions for database concurrency control [26].

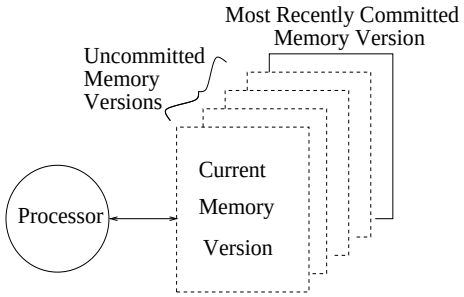

FIGURE 1. Abstraction of the Uniprocessor MVM Processor/Memory Interface

with a special write-back buffer called the *version buffer (VB).* L1 and VB blocks are also tagged with version numbers. The rest of the memory hierarchy is standard: level-two (L2) cache, optional L3 or more caches, coherence protocol, interconnect, and memory. On a prediction, a processor declares a new version and checkpoints its non-memory state. Thereafter, the processor core can proceed rapidly, without concern for speculation, including loading from and storing to the memory hierarchy. A store that is about to overwrite an L1 cache block tagged with a previous version number first triggers a writeback of the old version to the VB. A VB entry is not written to the L2 cache until its version commits. A processor with MVM1 can support kilo-instruction speculation with multiple simultaneous versions (e.g., 8-16). Versions commit in constant time and abort in constant time plus the implicit cost of some additional L1 misses. Sections 3 and 4 present MVM1 design issues without and with coherence, respectively.

Astute readers may notice some conceptual similarities between the MVM interface and the techniques that support backward error recovery for hardware fault tolerance. *Backward error recovery (BER)* techniques periodically checkpoint the state of the system and, if a fault is detected, recover the system to a saved checkpoint from before the occurrence of the fault [10, 28]. Traditional BER techniques tend to be more heavy-weight than the MVM1 implementation presented here (e.g., by requiring checkpoints to be flushed to memory or disk), and these overheads have limited their use beyond highly-reliable systems. Section 5 discusses modest MVM1 changes to enable the *unified support* of deep speculation and fault tolerance.

Finally, Section 6 provides simple cost models with parameters from commercial workloads that compare *mechanisms* for kilo-instruction speculation without restricting speculation policy or fault model. Results show that MVM1 recovers from kilo-instruction speculation in under 273 cycles, which is faster than either Speculative Retirement [29] (362 cycles) or SC++ [14] (500 cycles). MVM1 also uses less recovery storage (MVM1: 5,356 bytes, SR: 10,000 bytes, SC++: 12,000 bytes), and it has lower common-case overhead.

2 Related Work

The present effort to support deep recovery for speculation or fault tolerance builds on several existing threads of work. The primary related thread is the effort to extend the speculation capabilities of conventional (i.e., not speculatively multithreaded) out-of-order speculative processors, like the MIPS R10000 [42], so as to narrow the performance gap between systems that support sequential consistency (SC) and those

TABLE 1. Related Work in Speculation Support

Technique	Speculation Limit	Comment
MIPS R10000's Speculative Out-of-Order [42**]**	Instruction window or address queue size	Limited to 10-100 entries to permit associative searches on many processor cycles
Ranganathan et al.'s Speculative Retirement [29**]**	Store buffer and history buffer	Limited to 10-100 entries to permit associative searches on many processor cycles
Gniady et al.'s SC++ [14**]**	Recovery cost is linear in the number of speculative instructions	Non-perfect speculation is not viable when recovery cost gets too high.
MVM1 Implementation (this paper)	Recovery cost involves "misses" to cache blocks speculatively stored	Non-perfect speculation is not viable if recovery cost is too high.

that support weaker memory consistency models. Recall that the R10000 implements SC while still allowing many operations to speculatively proceed out of program order. Speculation depth is limited to the minimum of 32 instructions (due to the instruction window) or 16 memory operations (due to the address queue). While these structures will likely increase, neither structure can increase dramatically, since both must be associatively searched and updated on many processor cycles. Ranganathan et al.'s Speculative Retirement scheme [29] relieves pressure on the instruction window by allowing instructions to speculatively retire from the instruction window into a new history buffer that maintains enough state to permit instructions to be unrolled on a coherence violation. Speculation depth in this scheme is primarily limited by store buffer and history buffer size. Like the instruction window, the history buffer cannot be too large since it is also associatively searched to detect coherence violations. Gniady et al.'s SC++ scheme [14] permits even deeper speculation by (a) letting speculative stores complete into the level-one cache, (b) maintaining a large non-associative history buffer of all instructions, and (c) detecting coherence violations that trigger rollbacks with a new associative block lookup table that flags blocks accessed by any load or store in the history buffer. The depth of speculation is primarily limited by the cost of recovery, and this cost is linear in the number of speculative instructions.

MVM1 differs from this thread of related work by supporting even deeper speculation with the help of several features. First, MVM1 adds no associative structures that must be manipulated each cycle, since most MVM1 logic is behind the L1 cache. Second, MVM1 structure sizes are not linear with speculation depth, since processor state is recovered via checkpointing rather than by rolling back a history buffer. Third, recovery cost is much less than linear in speculation depth, because it is proportional to the blocks stored per version. Table 1 compares MVM1 with the R10000, Speculative Retirement, and SC++.

A second relative thread of related work is the effort to support a particular type of speculation, *speculative multithreading*, and not speculation in general [4, 12, 15, 25, 37, 38]. With speculative multithreading, contiguous sequences from the dynamic instruction stream of a program's execution—called *speculative*

threads—are distributed (often with compiler support) to processing elements (PEs) within a processor.[2] For example, a compiler could indicate that loop iteration 1 should go to one PE, loop iteration 2 to the second PE, etc. Speculative threads execute in parallel if the instructions are actually independent, and some mechanism is needed to detect and enforce the appearance of sequential thread execution when the threads are not independent. The first proposed mechanism is Multiscalar's Address Resolution Buffer (ARB) [37]. The ARB uses a centralized implementation that puts significant associative logic on the critical path of even load and store hits. DMT also uses a centralized solution that depends on having enough associative load and store buffers for all speculative threads [4]. Subsequent designs [12, 15, 25, 38] used a fast cache with each sub-processor backed by logic similar to snooping coherence to detect and enforce dependences.[3] These caches may be called *level-zero* caches, because they miss more often than a standard level-one cache due to (a) data sharing among sub-processors (e.g., a datum written by one sub-processor and read by the next) and (b) data replication consuming sub-processor cache capacity (e.g., a datum read by several sub-processors).

Multiversion Memory differs from this thread of related work primarily because MVM looks to support deep speculation and fault tolerance for processors and multiprocessors rather than speculative multithreading, in particular. Furthermore, MVM1 avoids either adding associative searches to load and store hits (like the ARB and DMT) or increasing the primary cache miss rate (like the other schemes). MVM, however, does not currently support speculation where versions execute in parallel in a manner similar to how speculative threads execute in parallel.

There is a vast amount of prior research in checkpoint/recovery schemes for fault tolerance (refer to Elnohazy et al. [10] for a survey and to Pradhan [28] for additional background). These schemes tend to be heavy-weight and conservative, in that they seek to tolerate a wider range of faults and they are less concerned with performance than reliability. Some schemes resemble our MVM1 implementation in that they use the caches to hold uncommitted state and use the shared memory to hold architectural state [16, 41]. In particular, Wu et al. [41] label cache blocks with checkpoint IDs in a manner similar to MVM1 version tags. MVM1 does not present a novel or superior fault tolerance mechanism; rather, it achieves some fault tolerance at a low cost by unifying it with the issue of deep multiprocessor speculation.

3 A Non-Coherent Uniprocessor Multiversion Memory

This section presents the MVM1 implementation of the multiversion memory interface, in the context of speculation, for a uniprocessor without coherence issues. This simplifies presentation by deferring coherence issues until Section 4 and fault tolerance issues until Section 5. This non-coherent design is only interesting as a stepping stone to the design of Section 4, because even most uniprocessors have coherent DMA.

2. DMT differs in that it places threads on a processor that supports simultaneous multithreading (SMT).

3. One of these designs [12] is coincidentally named multiversion caching, but the versioning is designed to support the Multiscalar paradigm and not speculation, in general.

3.1 Big Picture

The purpose of uniprocessor MVM1 is to allow processor speculations that do not resolve for 1000 instructions, which is much larger than a viable instruction window, and to allow several concurrent predictions and speculations (e.g., 8-16). The processor may speculate for any reason. An example uniprocessor speculation that does not resolve for 100s of cycles is a value prediction on a datum in main memory.

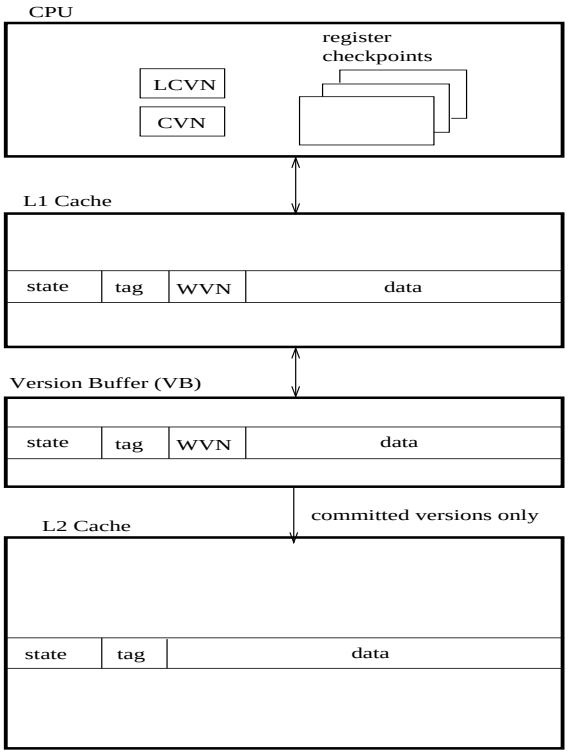

FIGURE 2. Uniprocessor MVM1 System

Figure 2 illustrates a system with uniprocessor MVM1. The processor maintains a *current version number (CVN)* to tag loads and stores, a *last committed version number (LCVN)*, and *register checkpoints* for uncommitted versions. Register checkpoints include architectural non-memory state, such as the registers, processor status, and PC. Blocks in the L1 cache and VB maintain a *write version number (WVN)*. The L2 cache and beyond are standard. The design will maintain the following invariants:

- *Uniprocessor correctness:* A store will always write its value into the current version of a block containing its address, while a load will always return the value from the most recent version of its block.
- *L1 as memory future file:*[4] Each valid block in the L1 cache is the most recently written version of the block.

4. The terms *future file*, *history buffer*, and *architectural file* were used by Smith and Pleszkun [36] when discussing registers. A future file contains the most recently updated (speculative) state, a history buffer contains old values clobbered by speculative writes, and the architecture file contains the safe (non-speculative) state. MVM1 uses similar notions, but it applies them to memory instead of registers.

- *VB as memory history buffer:* The VB holds all speculative blocks that are not the most recently written.
- *L2 and beyond as memory architectural file:* L2, L3, memory, etc., only hold committed state.

Consider an example using stores to a single two-word block at address 100. We use **bold** and shading to highlight changes. Assume initially that the processor's CVN=2 and that the L1,VB, and L2 state for block 100 are:

Structure	Address	WVN	Data	Comments
L1	100	2	{2,4}	block already dirty with respect to L2
VB	NONE			
L2	100	n/a	{0,0}	

The processor declares a new version by **setting CVN=3 and creating a version 3 register checkpoint.** This action is fast because no L1, VB, or L2 changes occur:

Structure	Address	WVN	Data	Comments
L1	100	2	{2,4}	no change
VB	NONE			no change
L2	100	n/a	{0,0}	no change

The processor performs a store of the value 12 to address 100: **ST 12, (100).** This action forces **the old version of block 100 to enter the VB** (from where it can be recalled after a misspeculation or fault):

Structure	Address	WVN	Data	Comments
L1	100	**3**	{**12**,4}	**update WVN & word 100**
VB	**100**	**2**	**{2,4}**	**write old version to VB**
L2	100	n/a	{0,0}	no change

The processor performs a store of the value 24 to address 104: **ST 24, (104).** This action proceeds like a store hit, illustrating how the L1 coalesces speculative updates:

Structure	Address	WVN	Data	Comments
L1	100	3	{12,**24**}	**just update word 104**
VB	100	2	{2,4}	no change
L2	100	n/a	{0,0}	no change

To commit version 2, the processor **sets LCVN=2 and discards the version 2 register checkpoint. The VB block's WVN is changed from 2 to null.** Commits are fast because no data movement is required:

Structure	Address	WVN	Data	Comments
L1	100	3	{12,24}	no change
VB	100	**null**	{2,4}	**non-speculative version**
L2	100	n/a	{0,0}	no change

The VB may now writeback the block to the L2, but this can occur any time later:

Structure	Address	WVN	Data	Comments
L1	100	3	{12,24}	no change
VB	NONE			**committed version gone**
L2	100	n/a	**{2,4}**	**update L2**

3.2 MVM1 Specification

This section examines MVM1 in more detail.

3.2.1 MVM1 Components

Processor. MVM1 uses a standard speculative, out-of-order, processor core with two additions. First, the processor must be able to checkpoint "register" state (e.g., PC and registers) when it declares a new version.[5] Second, it must maintain and use two version numbers. The *current version number (CVN)* gives the processor's current version, and it is used to tag stores to the L1 cache. The *last committed version number (LCVN*) gives the last version committed (i.e., guaranteed not to have a misspeculation), and it is the version to which the processor recovers after detecting a mis-speculation.

L1 Cache. The L1 cache serves as a memory future file. Each L1 block is tagged with a *write version number (WVN)*. While a block is not committed, the WVN denotes the version that wrote it. The WVN is *null* for committed blocks. MVM1 ensures that each valid block in the L1 cache is the most recently written version of the block.

Version Buffer. The VB serves as a memory history buffer. The VB holds versions of blocks, with corresponding WVNs, that were forced out of the L1 cache for one of two reasons. First, the L1 copies dirty blocks to the VB when stores create new L1 versions (as in the example from the last section). Second, the VB serves as a write-back buffer for normal dirty L1 replacements. As usual, clean blocks may be silently deleted. VB blocks are written to the L2 only if they become committed. Thus, the VB contains all uncommitted blocks that are not the most recently written.

L2 Cache. The L2 serves as the memory architectural file. It is a standard cache that holds committed state and has no version information. Our implementation assumes that the L2 cache maintains inclusion with the L1 cache by maintaining a bit per L2 block indicating whether the block may be in the L1 or VB, but inclusion is not necessary for MVM1.

5. Processor cores may also use one recovery method for predictions that are rapidly resolved (e.g., branch prediction) and then use multiversion memory for longer-term speculation in a manner inspired by the hierarchical speculation approach of Ranganathan et al. [29].

3.2.2 MVM1 Operation

Uniprocessor MVM1 operation proceeds as follows. The processor creates versions, performs loads, stores, and replacements during versions, commits versions, and aborts versions. Figure 3 illustrates each of these operations, and we discuss them below.

Creating a new version. A processor creates a new version by checkpointing non-memory state and incrementing its CVN.[6]

Execution within a version. During a given version, the processor executes without regard for MVM1 except to tag loads and stores with the CVN.

Committing a version. A processor commits a version by incrementing its LCVN and discarding the now unneeded architectural checkpoint. MVM1 commits version *i* in the L1 and VB by setting WVN=*null* for all blocks that had WVN=*i*.

Aborting a version. If a mis-speculation occurs in version *i*, the processor will revert to the checkpoint at the beginning of version *i*, and MVM1 will invalidate all L1 and VB blocks whose WVN ≥ *i*.

3.3 Implementation Issues

L1 Cache. The L1 cache design is conventional, with three important exceptions: (1) a store hit may trigger a writeback of the old block, (2) a commit of version *i* must find blocks with WVN=*i* and then set WVN=*null*, and (3) an abort of version *i* must invalidate blocks with WVN ≥ *i*. Case (1) can be detected by comparing the processor's CVN and the stored block's WVN in parallel with a standard tag comparison. A store to the cache thus reads the cache tags (but not data) before writing it, but this is also the case for normal stores, since they require a tag lookup.

Version commits and aborts can be made to operate globally on the L1 cache in constant time with two changes. First, we store version numbers decoded as 1-hot bit vectors. This representation requires *k* bits to support *k* active versions, which is not a problem for the small *k* we envision (e.g., 8-16). Second, we augment the L1 cache with a flash clear on each version bit column, similar to the mechanism used in caches that support *flash invalidation* [20].

Version Buffer Design. The VB design is simpler than the L1 cache design, and it is smaller. Later we will show that supporting 64-256 entries is generous. Given that entries are only accessed on L1 misses, a fully-associative design with a several-cycle access time would be adequate.

One VB design challenge is that the VB can contain multiple versions of the same block. In general, this could complicate block lookup circuits because multiple matches could occur. One way to avoid this problem is to bank the VB with the invariant that the blocks in a bank come from the same version, and each

6. Version number wraparound errors can be avoided by stalling if creating a new version would make CVN=LCVN.

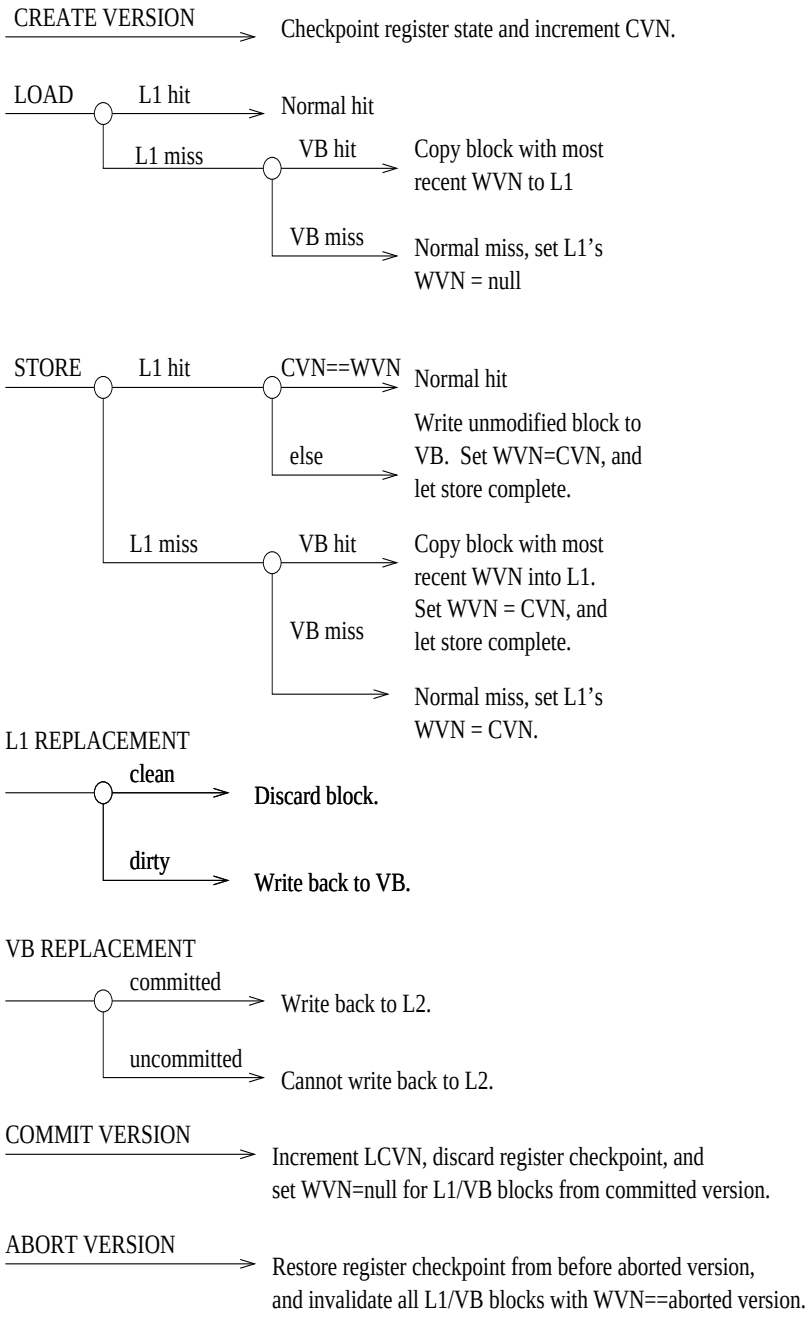

FIGURE 3. MVM1 Operation

bank could indicate its WVN. This helps because each version has at most one copy of a block. A version with a large number of blocks would span multiple banks.

An L1 miss in this design, for example, would query all banks for a block and then either select the most recent version with a priority encoder or declare a VB miss. A new bank would be allocated (if available or

stall otherwise) for a new version or when a version fills a bank. The bank(s) for a committed version could be written to the L2 cache.

On balance, this VB design has some complexity, but the design is not large (e.g., 16 16-block banks) and it can operate in a few cycles since it is behind the L1 cache.

Version Creation Policy. A naive version creation policy is to create a new version on each speculation. This policy could exhaust MVM1 resources by creating too many versions during a burst of speculations, thus limiting the effectiveness of coalescing stores from the same version in the L1 cache and VB. A better policy might be to create a new version on a speculation only if no version has been created in the last *i* instructions. Future work will determine how to set *i* or whether other policies are better.

4 Coherent Multiprocessor Multiversion Memory

This section augments the non-coherent uniprocessor MVM1 of the previous section so that it can operate with coherence and flourish in a cache-coherent shared memory multiprocessor. We assume a standard cache-coherent multiprocessor where each processor is augmented with MVM1. This new design is also appropriate in a uniprocessor that supports coherent DMA.

There are two primary differences between the MVM1 of a non-coherent uniprocessor and that of a coherent multiprocessor. First, multiprocessor MVM1 must deal with external coherence requests. To do this, it must be able to search the L1 and VB for the most recently committed version of a block and it must sometimes trigger version aborts. Second, multiprocessor MVM1 can support additional "multiprocessor" methods of speculation. These techniques include speculating on values cached at other processors, on values left in local cache blocks, and on lock acquisitions or critical sections. In addition to uniprocessor invariants, multiprocessor MVM1 implements a memory consistency model, such as *sequential consistency.*

4.1 Multiprocessor MVM1 Specification

MVM1 COMPONENTS. Multiprocessor MVM1 components are the same as for uniprocessor MVM1, except that each L1 and VB block includes a *read version set (RVS)* in addition to the WVN. The RVS of a block indicates which uncommitted versions have read (this version of) the block. Since an external coherence request for a possibly speculative block can cause an abort, as we will see later, maintaining the RVSs allows MVM1 to selectively abort speculation on external requests rather than blindly aborting.

MVM1 OPERATION. Multiprocessor MVM1 operation is the same as for uniprocessor MVM1, with the addition of maintaining the RVSs and handling external coherence requests.

Maintaining RVSs. The RVS is maintained with the following actions. A load during CVN=*i* adds *i* to a block's RVS. The RVS is copied with a block if the block moves between the L1 and VB. The RVS is initialized to *null* when a new write-version of a block is created or when the block is loaded from the L2. Finally, *i* is removed from all RVSs when version *i* commits. An uncommitted clean block, which was not possible in uniprocessor MVM1, must be written back from the L1 to the VB.

Handling external coherence requests. External coherence requests cause the following *standard* actions in an MSI protocol (exclusive **M**odified, read-only **S**hared, and **I**nvalid):[7]

- **M-->I:** another processor seeks an M copy, so send the block to the other processor and invalidate,
- **M-->S:** another processor seeks an S copy, so send the block to the other processor and memory and downgrade to S, or
- **S-->I:** another processor seeks an M copy from another source, so invalidate the block.

Most external coherence requests will be handled in the standard way by the L2 cache because the block's inclusion bit is not set. The remaining requests search the L1 and VB two ways. First, they search for versions of the block written in a committed version:

- If found, MVM1 applies the standard actions by obtaining data from the most-recently committed version and downgrading the state of all committed versions.
- If not found, the L2 cache performs the standard actions.

Second, external coherence requests search for versions of the block that have been *read or written* in still speculative versions:

- If found, all speculative versions are aborted.[8]
- If not found, no action.

The RVS allows MVM1 to know when a block being invalidated by an external coherence request has been read by a speculative version. MVM1 must trigger a recovery in this case to support most memory consistency models [13]. If RVSs were not maintained, multiprocessor MVM1 would have to trigger recoveries on all external invalidates, resulting in serious performance loss.

Implementation of new VB mechanisms follows from uniprocessor MVM1 implementation issues. Maintaining RVSs, for example, is similar to maintaining the unary encoded WVNs. Also, finding the most recently committed version in the VB is similar to finding the most recent version in the VB.

4.2 High-Level Issues

Multiprocessor Correctness. To argue that multiprocessor MVM1 is correct, we argue that it can be used to implement *sequential consistency (SC)* [19]. It then follows that it can be used to implement more relaxed models, since SC is a correct implementation of more relaxed models.

SC requires that a multiprocessor appears to the programmer as if the memory references of each processor (in program order) are interleaved to form a total order. We sometimes think of processor memory references entering the total order one at a time as they commit. With multiprocessor MVM1, all processor memory

7. All MOESI states can be handled at a cost of enumerating more cases.
8. A more sophisticated implementation could recover to the speculative version that first accessed the block.

references from a version enter the total order atomically as the version commits. Nevertheless, this "coarser" interleaving is a valid interleaving for SC.

Furthermore, the values obtained by coherence requests are correct, because coherence requests only obtain the most recently committed values, as they would in a standard system with coherence, and a processor never commits instructions that read speculative values that are later invalidated by an external coherence request, because the coherence request would trigger a recovery.

Livelock. An external coherence request that seeks to obtain a block that has been speculatively read will invalidate the block and trigger a recovery. The processor will then often issue a coherence request for the block. If the block is then in speculative use at another processor, this will cause an invalidate, trigger a recovery, and may cause the pattern to repeat and create a livelock. It would appear that the system could avoid livelock by not handling the coherence request until the block was non-speculative. Applying this solution in general, however, can lead to deadlock.[9]

There are simple solutions that risk livelock, detect when the processor is not making forward progress (because it is re-executing the same instruction), and complete that instruction. One way is to execute non-speculatively after livelock detection. Another possibility is to create a one-instruction version and defer coherence requests until the version commits. Deadlock will not occur because there is no "cross coupling" of dependences. We are investigating if livelock occurs often enough to warrant a more clever solution.

5 Unifying the Support for Speculation and Hardware Fault Tolerance

Seemingly unrelated to speculation is hardware fault tolerance. Hardware fault tolerance techniques commonly use either *forward error recovery (FER)* or *backward error recovery (BER)* [28]. FER techniques tolerate a fault while continuing to execute forward (e.g., using *error-correcting codes (ECC) or triple modular redundancy (TMR)).* Austin's DIVA design is a recent example of FER [5]. BER techniques restore the pre-fault state and re-try, similar to mechanisms for handling mis-speculation. Methods for restoring state include checkpointing and rolling back with logs. Implementation complexity depends on the class of faults to be tolerated and the latency from fault occurrence to detection. To date, however, powerful fault tolerance techniques have been deployed mostly in systems willing to trade performance for reliability [7, 17, 34].

Computer customers, such as providers of Internet services, are becoming increasingly interested in obtaining more robust systems, provided that they do not cost much more than traditional high-performance systems. Fortunately, MVM—initially included to support deep speculation—can also be used to provide this robustness using BER to improve hardware fault tolerance. Furthermore, the modest cost of extending `MVM1` for BER can make it attractive for traditional computer systems that have eschewed heavy-weight fault-tolerance mechanisms. The rest of this section discusses key issues.

9. Assume, for example, that processor 1 needs block B before it can commit block A, while processor 2 needs block A before it can commit block B.

Fault Model. Designing fault tolerance begins with a clear *fault model* that states which faults will be tolerated and which are beyond the scope of proposed mechanisms. MVM1 designs of Sections 3 and 4 can be made to tolerate faults within a single processor, provided the faults can be detected reasonably soon after they occur using reasonable additional hardware and that they disappear on re-execution (i.e., are transient). Examples of faults that can be tolerated include incorrect operation of processor datapath, functional units, and control logic. Not included (at this time) are faults in architectural state (e.g., registers, cache, and memory) or faults in interactions between processors (e.g., memory interconnect). Faults in architectural state may be addressed with error correcting codes, while the end of this section explores inter-processor issues.

Fault Detection Mechanisms. Faults of interest must be detected with reasonable latency (e.g., less than 100 cycles). Recent mechanisms useful for our fault models use redundant SMT threads [30, 31]. Other mechanisms include duplication and comparison [17,33] or diagnostic and coding techniques [27].

Version Creation Policy. To support speculation only, we assumed a new version would be created on a speculation only if no version has been created in the last *i* instructions, for some *i* to be determined. To also support fault tolerance, we must retain at least one version from more than *c* cycles ago, where *c* is the maximum latency between when a fault occurs and when it is detected and invokes a recovery. If speculation is frequent, no versions need to be created especially for fault-tolerance. If speculation is rare, creating versions with any period greater than *c* ensures that only three versions are needed to support fault tolerance.

Output Commit Problem. Another issue that arises in fault tolerance is that we cannot allow operations from uncommitted versions to interact with the outside world—disks, networks, and other I/O devices—since we might want to recover. This is the *output commit* problem that exists for backward error recovery schemes, in general [11, 23]. We propose to handle this issue with the standard solution of keeping uncacheable operations at processor nodes until their versions commit. This solution will work well for emerging I/O approaches, such as VIA [9] and InfiniBand [1], that first set up I/O descriptors in memory and then trigger I/O with a single uncached "doorbell." It could be slow, however, for conventional I/O interfaces, although the latency of I/O operations is likely to dominate this overhead.

Future Extensions to Global Recovery. Multiprocessor MVM enables multiprocessor speculation, but it only handles the same faults as uniprocessor MVM, namely, faults within the processor that can be tolerated with a recovery of a single processor. Ultimately, we would like to support a more general multiprocessor fault model that, for example, enabled a recovery from transient interconnection network errors, such as corrupted messages, lost messages due to buffer overflows, and temporary losses of synchronization between a sender and receiver.

We are investigating how to support global recovery by having processors create versions at the same logical times. Coordinating checkpoints avoids the problem of *cascading rollbacks* where inconsistent checkpoints can force rollbacks arbitrarily far back [28]. Using logical rather than physical time for coordinated check-

points mitigates the delay for creating checkpoints. In an SMP, for example, the number of coherence requests processed is a viable source of logical time.

6 Performance Evaluation

MVM1 was designed to support a wide variety of speculation and fault models. As such, the evaluation of MVM1 *will focus on the efficiency of its mechanisms rather than on its performance for any particular speculation policies or fault models*.

6.1 Methodology and Benchmarks

We develop simple cost models with input parameters obtained using the Simics full system simulator [22] to simulate the SPARC v9 architecture running Solaris 7. We simulate a 16 processor system, but we focus on uniprocessor issues. The L1 cache is 64 kB and 2-way set-associative. We evaluated MVM1 with three commercial applications and one scientific application.

- **Database decision support system (DSS):** We selected a representative query from the TPC-H benchmark [39], a recent successor to TPC-D, and we executed it on a 100 MB database using IBM's DB2. v6.1 database management system.
- **Web Server:** We used the Apache 1.3.9 web server [2] driven by SURGE, the Scalable URL Request Generator [6].
- **Web search engine:** We used an evaluation copy of the Altavista Search Engine V2.3A for Solaris [3].
- **Scientific application:** We selected *barnes* from the SPLASH-2 suite [40], using the 1K body input set.

6.2 Results

In this section, we compare the recovery latencies, storage costs, and common case overhead of MVM1 versus previous schemes for general speculation (but not speculative multithreading, in particular). We develop cost models, and the inputs to these cost models are derived from program profiling data that was gathered while running the benchmarks. Table 2 provides the profiling data. For each benchmark, and for various version lengths, it lists the mean number of stores and the mean number of distinct blocks written per version.

TABLE 2. Program Profile Data

version length	TPC-H, query 11		Apache/SURGE		Altavista		Barnes	
(in instructions)	stores	store blocks	stores	store blocks	stores	store blocks	stores	store blocks
50	5.2	2.6	6.3	3.0	6.1	2.3	5.1	2.5
100	9.4	4.1	12.0	4.9	11.7	3.4	9.9	4.0
150	13.6	5.5	17.7	6.8	17.3	4.2	14.6	5.0
1000	76.2	22.9	115.4	30.0	112.1	13.2	95.7	8.1

6.2.1 Recovery Latencies

We now compare the recovery latencies for speculation that takes 1000 instructions to resolve. We assume that the MVM1 system uses 10 versions of 100 instructions each. We also assume that data blocks are 64 bytes and that addresses are 4 bytes (32 bits).[10]

Speculative Retirement. Ranganathan et al.'s Speculative Retirement scheme [29] recovers from a mis-speculation by rolling back the register writes for each instruction that wrote to a register and was then logged in the history buffer. The rollback latency is equal to:

mean number of instructions that write a register per 1000 × *latency to rollback history buffer entry*

The first term is on the order of the number of instructions, but, most significantly, it does not include store or branch instructions. Subtracting out the mean number of stores in the worst-case benchmark (76 for TPC-H, as shown in Table 2) and assuming that branches are 20% of all instructions, we get a total of 1000 - (76 + 200) = 724 instructions in the history buffer. Assuming we can restore 2 history buffer entries per cycle, the rollback latency is 362 cycles.

SC++. Gniady et al.'s SC++ scheme [14] recovers by rolling back the effects of every speculative instruction. The rollback latency is equal to:

1000 instructions × *latency to rollback history queue entry*

The maximum latency to rollback a history queue entry is equal to the latency to write a store into the L1 cache. Assuming we can rollback 2 instructions per cycle, we have a latency of 500 cycles.

MVM1. MVM1 recovers by restoring to the checkpoint state that precedes the mis-speculated instruction and replaying any unnecessarily undone instructions. The recovery latency of MVM1 consists of three factors. First is the cost of recovering the register state and invalidating the speculative blocks from the L1 and the VB. Second is the time to replay the lost non-speculative work that was done between when the checkpoint was taken and when the abort occurred. The third factor is the implicit cost of MVM1 replay that is due to the L1 cache misses that will occur for accesses to blocks that were squeezed out of the L1 cache when they were clobbered by (useless) speculative stores.

Assuming naïvely that the aborted processor will stall until it has re-issued and completed the same stream of loads and stores that were aborted, the recovery cost is conservatively equal to:

register recovery and invalidate latency +
time to replay lost non-speculative work done between checkpoint and abort +
(mean number of store blocks per 1000 instructions × *latency to refill store block)*

The register recovery and invalidate latency is short, say 8 cycles, since the register recovery is determined by the number of registers (32) divided by the number of register file write ports (assume 4), and the invali-

10. We assume 32-bit addresses since we simulate a 32-bit machine, but there is no restriction against 64-bit addresses.

date can be implemented as a flash invalidation. The time to replay lost non-speculative work is, on average, on the order of half the number of instructions in the version. At 2 instructions per cycle, this comes to 25 cycles, but this is pessimistic for policies we would expect to use in practice. Policies would likely trigger versions upon speculation rather than at fixed intervals, as assumed here, making this cost negligible. For Apache/SURGE, the mean number of store blocks per 1000 is only 30. We assume the latency to refill a store block is roughly equal to an L2 hit (although it could be a faster VB hit), and we choose a value of 8 cycles. Thus, we appear to have a latency of 8 + 25 + 240 = 273 cycles. However, the actual latency is considerably less than that, since the processor can easily pipeline the VB accesses and potentially overlap useful work. Moreover, a possible optimization for hiding refill latency would be to prefetch aborted store blocks from the oldest version that was aborted.

Summary. The recovery latencies of Speculative Retirement and SC++ are 362 and 500 instructions, respectively. The recovery latency for MVM1 is 33 required cycles plus 240 potentially overlapped cycles. MVM1's low penalty for recovery can be viewed as either permitting higher mis-speculation rates for the same performance, thus enabling more aggressive speculation techniques, or providing better performance for a given mis-speculation rate.

6.2.2 Storage Structure Costs

We now compare the storage structure costs for speculation that takes 1000 cycles to resolve, and we make the same assumptions as in Section 6.2.1. We first present the cost, in bytes, of each scheme. Then we discuss the relative costs per byte, because, while storage cost is partly a function of the sheer number of bytes needed to buffer speculative state, storage cost also depends on the complexity of the hardware necessary to search the buffer.

Speculative Retirement. An entry is logged in the history buffer for every speculative instruction that writes to a register. Storage cost is equal to:

maximum number of instructions that could write a register per 1000 × *size of entry in history buffer*

The first term is equal to 1000 instructions, even though the mean number of instructions that write a register is less than all 1000. The size of a history buffer entry is equal to the sum of the sizes of the program counter (4 bytes), the previous value of the register (4 bytes), and the register map information (2 bytes). Thus, this scheme has a storage cost of 1000 × 10 bytes = 10,000 bytes.

Beyond the sheer number of storage bytes required, storage for Speculative Retirement is costly because it requires that the entire history buffer be content addressable. Building increasingly large content addressable memories (CAMs) is expensive and increases the time to access the CAMs.

SC++. An entry is logged in the speculative history queue for every speculative instruction. The storage cost is equal to:

1000 instructions × *size of entry in speculative history queue*

The size of a speculative history queue entry is equal to the size of an address (4 bytes) plus the size of a word of data (8 bytes). Thus, we get a storage cost of 12,000 bytes. SC++ also allocates storage to a block lookup table which is a list of all of the block addresses that are accessed by loads and stores in the history queue. Its size is much smaller than the size of the history queue.

The storage cost per byte of the history queue is less than that of Speculative Retirement, because it does not need to be content addressable. The block lookup table, however, needs an associative lookup.

MVM1. MVM1 saves copies of the register state for every version, and it stores an entry in the VB for every store block in a version. The storage cost of MVM1 (for 10 versions of 100 speculative instructions each) is equal to:

$$10 \times \textit{mean number of store blocks per 100 instructions} \times \textit{safety factor} \times \textit{size of VB entry} + 10 \times \textit{number of registers} \times \textit{register size}$$

The mean number of store blocks per 100 instructions for Apache/SURGE is 5. The safety factor is included to accommodate variability in the mean number of stores, and we choose a safety factor of 1.5. The size of a VB entry is equal to the size of a tag (4 bytes) plus the size of a block of data (64 bytes). We assume that we have 32 registers at 8 bytes each. Thus, MVM1 has a storage cost of 5,356 bytes.

MVM1 requires associative search of the VB but, since the MVM1 design is banked by version, it can have relatively small CAMs. Also, MVM1 can tolerate relatively slow CAMs, since they are beyond the L1 cache.

Summary. Speculative Retirement uses 10,000 bytes, of which 4,000 bytes (40%) need to be fast CAM. SC++ uses 12,000 bytes, but it only requires a small fast CAM. MVM1 uses a total of 5,356 bytes, which is divided up among 10 banks. Each bank needs a 40 byte CAM, but the CAM does not have to be fast, since it is beyond the L1 cache.

6.2.3 Common Case Overhead

The most nebulous cost to quantify—but a cost that is critical to performance—is the overhead that is incurred in the common case, while the processor is operating in the absence of mis-speculations and faults. Accurate calculation requires detailed implementation of all three schemes. Instead, we estimate these costs based on the overheads for load and store hits, coherence requests for non-speculative blocks, L2 replacements, and committing speculative work once the prediction is verified.

Speculative Retirement. An associative lookup of the history buffer is incurred for every store hit, coherence request, and L2 cache replacement. For each of these events, the system must ensure that there are no speculative loads in the history buffer which could be invalidated.

SC++. A history queue lookup is incurred for every coherence request and L2 replacement, but a costly associative search of the history queue is avoided through the use of the block lookup table. However, the

TABLE 3. Comparison of Speculation and Fault Tolerance

	uniprocessor speculation	**multiprocessor speculation**	**fault tolerance**
prediction examples	branch direction, data value	false sharing, lock acquisition	fault-free execution
probability of correct prediction	med-high	med-high	very high
checkpoint state	registers, core state	registers, core state, cache state	registers, core state, cache state
recovery destination	before mispredict	before mispredict	before fault occurrence (not detection)
instructions until checkpoint can be committed	10s - 100	100s	100s - 1000s

block lookup table still must be queried for every coherence request and L2 replacement, and this lookup is on the critical path.

MVM1. The cost of a writeback from the L1 to the VB is incurred whenever a store hit displaces an old version of the same block. Coherence requests and L2 replacements require an MVM1 lookup, but only for blocks that the L2 indicates could possibly be in the L1 or VB. Moreover, the L1 tag match and the VB lookup are both fast. Creating new versions and committing speculative work is done in constant time.

7 Conclusions

The increased relative cost of accessing memory is encouraging processor designers to explore deeper uniprocessor speculation (e.g., with branch and value prediction) and consider multiprocessor speculation (e.g., on coherence message types and values). Concurrently, manufacturing trends toward deep-sub-micron design and customer requirements for highly-available computers are encouraging techniques to mask, at least, transient faults (e.g., with error correcting codes and execution retry). Most current designs consider speculation and fault tolerance independently, but the similarities between the support interfaces required by speculation and fault tolerance, summarized in Table 3, suggest a common interface.

In this paper, we propose a unified processor/memory interface, called Multiversion Memory (MVM), to support checkpoint and recovery for both speculation and fault tolerance. We develop an efficient MVM implementation, called MVM1, that uses a level one cache to keep recent speculative blocks (like a future file), version buffers to keep old versions of blocks for which speculation is pending (like a history buffer), and leaves the level two cache unchanged (like an architectural file).

Simple cost models with parameters for 16 processors and commercial workloads show that using MVM1 allows kilo-instruction speculation and fault tolerance, with little overhead to create new versions and a recovery cost that is smaller than previous mechanisms.

We have implemented an MVM1 memory system simulator, which we use in conjunction with the Simics full system simulator [22], and we plan on pursuing research with MVM. MVM enables many types of kilo-instruction speculation techniques that have not had a sufficient checkpoint/recovery mechanism, and it also enables speculation techniques that have not yet even been explored. For example, we are not aware of any multiprocessor speculation technique that uses global checkpointing to allow speculative data to be observed by other processors.

8 References

[1] http://www.futureio.org/home.html.

[2] http://www.apache.org/httpd.html.

[3] http://doc.altavista.com/business_solutions/bus_solutions.html.

[4] Haitham Akkary and Michael A. Driscoll. A Dynamic Multithreading Processor. In *31st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 31)*, pages 226–236, November 1998.

[5] Todd M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *32nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 32)*, pages 196–207, November 1999.

[6] Paul Barford and Mark Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the 1998 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 151–160, June 1998.

[7] P. Bernstein. Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing. *IEEE Computer*, 21(2), February 1988.

[8] E. Ender Bilir, Ross M. Dickson, Ying Hu, Manoj Plakal, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Multicast Snooping: A New Coherence Method Using a Multicast Address Network. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, Atlanta, Georgia, May 1999.

[9] Dave Dunning et al. The Virtual Interface Architecture. *IEEE Micro*, 18(2), March/April 1998.

[10] E.N. Elnohazy, D.B. Johnson, and Y.M. Wang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU-CS-96-181, Department of Computer Science, Carnegie Mellon University, September 1996.

[11] E.N. Elnohazy and W. Zwaenepoel. Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. *IEEE Transactions on Computers*, 41(5):526–531, May 1992.

[12] Manoj Franklin. Multi-Version Caches for Multiscalar Processors. In *Proceedings of 1st International Conference on High Performance Computing*, 1995.

[13] 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 (Vol. I Architecture)*, pages I–355–364, August 1991.

[14] Chris Gniady, Babak Falsafi, and T.N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 162–171, May 1999.

[15] Sridhar Gopal, T.N. Vijaykumar, James E. Smith, and Gurindar S. Sohi. Speculative Versioning Cache. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture*, pages 195–205, February 1998.

[16] D.B. Hunt and P.N. Marinos. A General Purpose Cache-Aided Rollback Error Recovery (CARER) Technique. In *Proceedings of 17th Annual Symposium on Fault-Tolerant Computing*, pages 170–175, 1987.

[17] D. Johnson. The Intel 432: A VLSI Architecture for Fault-Tolerant Computing. *IEEE Computer*, pages 40–48, August 1984.

[18] Stefanos Kaxiras and James R. Goodman. Improving CC-NUMA Performance Using Instruction-Based Prediction. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, pages 161–170, January 1999.

[19] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[20] David D. Lee and Randy H. Katz. Using Cache Mechanisms to Exploit Nonrefreshing DRAM's for On-Chip Memories. *IEEE Journal of Solid-State Circuits*, 26(4):657–66, April 1991.

[21] Mikko H. Lipasti and John Paul Shen. Exceeding the Dataflow Limit via Value Prediction. In *29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 29)*, December 1996.

[22] Peter S. Magnusson et al. SimICS/sun4m: A Virtual Workstation. In *Proceedings of Usenix Annual Technical Conference*, June 1998.

[23] J. Roger Mitchell and Vijay K. Garg. Group Message Logging Without Orphans for Fast Output Commit. Technical Report TR-PDS-1997-003, Parallel & Distributed Systems group, Dept. of Electrical and Computer Engineering, Univ. of Texas at Austin,

March 1997.

[24] Shubhendu S. Mukherjee and Mark D. Hill. Using Prediction to Accelerate Coherence Protocols. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[25] Jeffrey Oplinger, David Heine, Shih-Wei Liao, Basem A. Nayfeh, Monica S. Lam, and Kunle Olukotun. Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor. Technical Report CSL-TR-97-715, Stanford University, May 1997.

[26] Christos Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, Rockville, Maryland, 1986.

[27] Dhiraj K. Pradhan. *Fault-Tolerant Computing: Theory and Techniques*, volume I. Prentice-Hall, Inc., 1986.

[28] Dhiraj K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice-Hall, Inc., 1996.

[29] Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *Proceedings of the Ninth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 199–210, June 1997.

[30] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 25–36, June 2000.

[31] Eric Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *IEEE 29th International Symposium on Fault-Tolerant Computing*, pages 84–91, June 1999.

[32] Ashley Saulsbury, Fong Pong, and Andreas Nowatzyk. Missing the Memory Wall: The Case for Processor/Memory Integration. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 90–101, May 1996.

[33] O. Serlin. Fault-Tolerant Systems in Commercial Applications. *IEEE Computer*, pages 19–30, August 1984.

[34] Timothy J. Slegel et al. IBM's S/390 G5 Microprocessor Design. *IEEE Micro*, pages 12–23, March/April 1999.

[35] J. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 135–148, May 1981.

[36] James E. Smith and Andrew R. Pleszkun. Implementing Precise Interrupts in Pipelined Processors. *IEEE Transactions on Computers*, C-37(5):562–573, May 1988.

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

[38] J. Gregory Steffan and Todd C. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture*, February 1998.

[39] Transaction Processing Performance Council. TPC Benchmark H (Decision Support), Standard Specification, Revision 1.1.0, June 1999.

[40] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Shingh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 22–24, 1995.

[41] K. Wu, W. K. Fuchs, and J. H. Patel. Error Recovery in Shared Memory Multiprocessors Using Private Caches. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):231–240, April 1990.

[42] Kenneth C. Yeager. The MIPS R10000 Superscalar Microprocessor. In *IEEE Micro*, pages 28–40, April 1996.