Calvin: Deterministic or Not? Free Will to Choose

Derek R. Hower, Polina Dudnik, Mark D. Hill, and David A. Wood

Computer Sciences Department
University of Wisconsin-Madison
1210 W Dayton St
Madison, WI 53706
{drh5,pdudnik,markhill,david}@cs.wisc.edu

Abstract

Most shared memory systems maximize performance by unpredictably resolving memory races. Unpredictable memory races can lead to nondeterminism in parallel programs, which can suffer from hard-to-reproduce hiesenbugs.

We introduce Calvin, a shared memory model capable of executing in a conventional nondeterministic mode when performance is paramount and a deterministic mode when execution repeatability is important. Unlike prior hardware proposals for deterministic execution, Calvin exploits the flexibility of a memory consistency model weaker than sequential consistency. Specifically, Calvin logically orders memory operations into strata that are compatible with the Total Store Order (TSO). Calvin is also designed with the needs of future power-aware processors in mind, and does not require any speculation support.

We develop a Calvin-MIST implementation that uses an unordered coalescing write cache, multiple-write coherence protocol, and delayed (timebomb) invalidations while maintaining TSO compatibility. Results show that Calvin-MIST can execute workloads in conventional mode at speeds comparable to a conventional system (providing compatibility) or execute deterministically for a modest average slowdown of less than 20% (when determinism is valued).

1. Introduction

Nondeterminism in multithreaded applications arises from memory races that current implementations does not control, especially for shared memory multiprocessor systems such as multicore processors. This nondeterminism can lead to problems, such as hard-to-find bugs that cost billions of dollars per year [38].

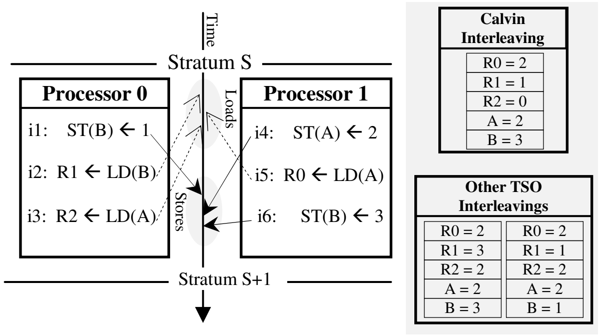
Recently, researchers have proposed various hardware [11,43] and software [5,11,32] solutions to address multithreaded nondeterminism. They have shown that addressing the problem has the potential to (1) increase software reliability by enhancing software test coverage before release [43], (2) increase system reliability through replication based fault tolerance [9], (3) aid in multithreaded software engineering [42], and (4) enhance security by providing a tool to analyze an attack [13]. Many of these prior proposals either rely on the ability to replay a previously recorded execution [14,20,27,28,31,41,42], incur a performance overhead that is likely too high for always-on usage [5], require complex speculative hardware [11], or only guarantee determinism in well behaved programs [32].

In response to these shortcomings, we propose Calvin, a multiprocessor system model that can guarantee determinism for multithreaded applications at an acceptably low overhead (e.g., 20%). The Calvin model is fully compatible with the *Total Store Order (TSO)* memory model [18,40], making it backward compatible with the majority of commercially relevant architectures, including x86, SPARC, PowerPC, and ARM. TSO defines a total memory order that is consistent with each processor's program order, except that stores may be delayed provided a processor's loads see its own stores immediately (e.g., an implementation can use FIFO store buffers, even without speculation).

While determinism shows great potential for developing new multithreaded software, some applications may not benefit from system-enforced determinism, so those applications should not have to pay a determinism performance penalty. For example, some language and run-time systems provide deterministic execution semantics on nondeterministic hardware [2,8,16] and existing multithreaded software may already be robust to nondeterministic effects. These systems receive little or no benefit in exchange for any overheads of system-enforced determinism.

To allow both deterministic and nondeterministic execution, a Calvin system can execute in one of three modes with different determinism guarantees.

- In *Conventional (C) mode,* a Calvin system does not make specific guarantees about execution order and behaves like a conventional TSO system.
- In *Bounded Deterministic (BD) mode*, a Calvin system guarantees that an execution will be repeatable when run on *the same* Calvin hardware implementation and given the same input.
- In *Unbounded Determinism (UD) mode,* a Calvin system guarantees that an execution will be repeatable when run on *any* Calvin hardware implementation and given the same input.



Calvin Interleaving
R0 = 2
R1 = 1
R2 = 0
A = 2
B = 3

Other TSO Interleavings	
R0 = 2	R0 = 2
R1 = 3	R1 = 1
R2 = 2	R2 = 2
A = 2	A = 2
B = 3	B = 1

Figure 1 –*Calvin execution deterministically enforces a single valid TSO interleaving (top right) from among the several possible alternatives. Within a stratum S, all processors logically order all loads first and then all stores in a fixed order (e.g., processor P0's stores before P1's). To conform to TSO, each load by Pi gets its value from a store by Pi before it in program order (if any) or from the value at the end of stratum S-1. For example, instruction i2 gets its value from i1, while i3 gets a value from stratum S-1. Strata are sequentially ordered.*

As we will show in Section 2.2, the three modes of a Calvin system offer a user-adjustable knob that can trade off reduced performance for stronger determinism guarantees. Importantly, we also show that a user not wanting determinism does not have to incur a large performance penalty in a Calvin system (i.e., Conventional mode has comparable performance to a non-Calvin baseline system). Depending on application requirements, users can choose BD mode when determinism is desired across different systems or UD mode when determinism on the same system will suffice. When the weaker guarantee of BD is sufficient, performance may improve.

Hardware enforced determinism is valuable only if it can be achieved with good performance at acceptable power across many systems, including those that use simple cores with little or no speculation [21,39]. To this end, we explore the extreme position of implementing Calvin with a simple, in-order non-speculative core (and without the speculation support required by previous deterministic systems [11]). Future work may show that adding speculation makes performance-power sense for some systems.

Calvin works by having all processors map memory operations into a series of global *strata,* (see Figure 1). Strata end when a stratum termination function holds for all processors. The stratum termination function differs for each of the three Calvin modes. Conventional mode minimizes Calvin's performance overhead by ending strata nearly simultaneously (e.g., by counting cycles). BD mode considers deterministic micro-architectural events (e.g., store buffer full) as well as architectural events (e.g., store count). UD mode ends strata based on architectural events only.

To this end we develop the Calvin-MIST implementation with some key micro-architectural features:

- It replaces a standard FIFO store buffer with a simpler-to-make-larger *unordered coalescing write cache*, while still maintaining TSO.
- It implements a *multiple-writer coherence protocol*, again without compromising TSO.
- The protocol adds a *timebomb (T) state* to the conventional MSI states, hence the name "MIST," to plant delayed invalidations that cause blocks to self-destruct when the current stratum ends.

We evaluate Calvin-MIST with the Parsec [6] and Mantevo [1] workloads running on x86 Linux 2.6.26. We simulate a 8-processor multicore with Bochs [25] and GEMS [24] and compare against a conventional nondeterministic system implementing an MOESI protocol. Results ask and answer two questions:

Question 1: Can Calvin-MIST avoid harm? Yes, Calvin-MIST executes nondeterministic programs at speeds comparable to a conventional system, thereby maintaining functional/performance compatibility.

Question 2: Can Calvin-MIST do some good? Yes, Calvin-MIST executes deterministic programs at a performance overhead less than 20%, thereby providing a benefit when determinism is valued.

Moreover, if record-replay is desired, Calvin's deterministic execution can eliminate the need for memory race recording at reasonable overhead, because only one memory race outcome is possible.

In our view, contributions of this paper include:

- **Demonstrate a Non-Speculative Hardware Implementation** that shows determinism can be provided at an acceptable performance even without the power and complexity of speculation.
- **Leverage Total Store Order (TSO) Hardware** which is compatible with ARM, SPARC, PowerPC, and x86 systems and provides more freedom for optimization than sequential consistency, assumed in previous hardware determinism systems [11,20,27,30,31,33,34,42].

Below, we present the Calvin execution model (Section 2), describe the Calvin-MIST implementation (Section 3), give evaluation methods (Section 4), provide experimental results (Section 5), contemplate future work (Section 6),discuss related work (Section 7), conclude (Section 8), and formalize (Appendix A).

2. Calvin Model

Calvin partitions an execution into strata whose termination condition determines the execution mode. To follow TSO terminology, we use loads/stores to refer to the reads/writes of x86 instructions.

2.1. Strata

The Calvin execution model partitions the dynamic loads and stores of a multiprocessor execution into global strata. Operationally, each processor begins a stratum, executes dynamic loads and stores until a stratum termination condition holds, synchronizes with other processors to ensure deterministic store order and repeats for the next stratum. An interrupt gets deferred until the next stratum boundary, much like how an interrupt during a complex instruction awaits an instruction boundary. The system logically keeps strata in sequence, so that all processors appear to complete loads and stores for stratum S before they appear to execute loads and stores for stratum S+1.

Within each stratum, execution proceeds as:

1. Each processor appears to execute its instructions, including loads and stores, in program order, but defers the global visibility of stores so that they appear after all loads (e.g., with a store buffer).
2. Loads return the address' value at the *beginning of the stratum*, unless the same processor has performed a store to the same address within the stratum (i.e., store buffer bypassing).
3. Finally, Calvin specifies that the stores of different processors be ordered in a predictable order. After all loads are logically complete, processor P0's stores get ordered, then processor P1's stores, etc. Priorities should be rotated during subsequent strata to ensure fairness and avoid deadlock.

Strata rules have several consequences. First, stratum execution is legal under TSO, ensuring backward compatibility. See a proof sketch in Appendix A. Second, stratum rules permit exactly one TSO execution, ensuring determinism within each stratum. Third, loads and stores from different processors do not communicate within a stratum. In particular, each load gets a value either from a previous store by its own processor or the value at the end of the last stratum. This consequence will allow our implementation to use *unordered store buffers* and a *multiple-writer coherence protocol*.

The stratum memory ordering invariants hold for all three Calvin execution modes. The next subsection discusses how adjusting stratum termination determines whether the complete execution exhibits bounded determinism, unbounded determinism, or nondeterminism (i.e., conventional).

2.2. Stratum Termination Condition Determines Execution Mode

A Calvin processor reaches the end of a stratum when a stratum termination condition holds for that particular processor, while the stratum globally completes when all processors have arrived at the stratum boundary. Thus, stratum termination is logically a barrier but does not have to be implemented as one.

The stratum termination condition determines whether the system operates in conventional, bounded deterministic or unbounded deterministic mode:

Conventional (C). A Calvin system executes in conventional mode if the stratum termination function depends on nondeterministic criteria. For example, a stratum termination function based on cycle count produces stratum boundaries at nondeterministic execution points, but can maximize performance by minimizing the load imbalance of when processors end strata.

Bounded Deterministic (BD). A stratum termination condition that uses both architected and non-architected but predictable state can provide a bounded deterministic execution. For example, a stratum could end either after a certain number of instructions have completed or when a store buffer fills up. This mode may reduce the cost of building a Calvin system compared to a more robust form of determinism (discussed next) by, for example, permitting a smaller store buffer.

Unbounded Deterministic (UD). An unbounded deterministic execution results if the stratum termination condition depends only on architected state, e.g., instruction count. A UD execution is deterministic across *all* implementations of the Calvin architecture.

2.3. Atomic Operations

Atomic read-modify-write operations require special treatment in the Calvin model, just as they do in the underlying TSO model. Atomic operations in a TSO system obey the following rules: (a) execute all previous load and stores, (b) perform the load and store of the atomic operation, and (c) then execute any subsequent loads and stores. Operations of other processors may interleave with (a) and (c), but not (b).

Calvin handles atomic operations by (1) ensuring that at most one atomic executes per stratum and (2) logically placing atomics at the end of a stratum. Calvin inserts an implicit condition into all stratum termination conditions to end a stratum immediately after an atomic, achieving condition (1) above. Second, Calvin executes a processor's atomic as if it were the processor's last store of the stratum (including the read part of a RMW). This ensures that all previous loads and stores are ordered before the atomic (TSO rule part a).

While Calvin's atomic rules correctly implement TSO rules, they have an important consequence. Processors can communicate within a stratum via atomics, thereby violating Calvin rule 1. For example, if processor P0 stored 0, while processors P1 and P2 performed atomic increments on the same address, the address's final value would be 2. Thus, both atomic increments observe a value updated in their own stratum.

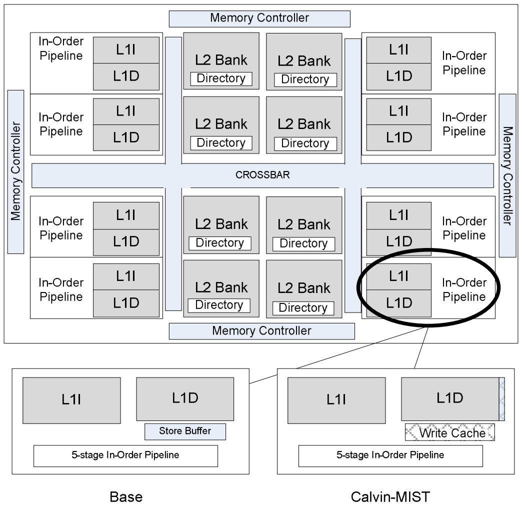


Figure 2 - *Base system with Calvin additions highlighted: the write cache, a single timebomb bit per L1D cache block, and a dedicated barrier line*

2.4. External Inputs

All potentially deterministic systems can only be deterministic in response to deterministic input. This is straightforward for programs that operate on fixed input data that is available before execution begins.

Calvin also remains deterministic in the presence of internally generated and/or asynchronous inputs. Internally generated inputs are predictably scheduled a predefined number of strata after a causal action (e.g., after initiating a DMA read). Asynchronous inputs are made repeatable by recording the contents and logical time of the input (e.g., an interrupt vector number and the dynamic instruction count when the interrupt was raised), as done by record-replay systems [42].

3. Calvin-MIST: A First Implementation

Calvin-MIST, our initial implementation of the Calvin model, targets the multicore system illustrated in Figure 2. Calvin-MIST replaces the conventional ordered FIFO store buffer found in conventional TSO systems with a set-associative, non-FIFO, unordered coalescing write cache (Section 3.1). Our design also implements the MIST multiple-writer coherence protocol (Section 3.3), which supports multiple concurrent writers and a timebomb (T) state that causes blocks to self destruct at the end of strata.

Calvin-MIST executes each stratum in two phases, illustrated in Figure 3. In phase one, each processor locally executes its instructions in program order. Stores write their address and data into the write cache. Loads check the write cache, bypassing their data if present, and access the cache hierarchy otherwise. Processors synchronize at the end of phase one using a dedicated fast hardware barrier.

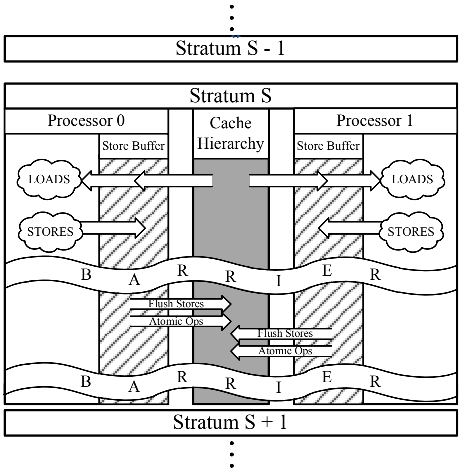


Figure 3 - *Calvin-MIST operation*

In phase two, the processors flush their write caches in parallel to the cache hierarchy. Updates to exclusive blocks occur locally, incurring no additional communication beyond a conventional writeback coherence protocol. For blocks with multiple writers, the MIST coherence protocol ensures that updates occur in a deterministic order. Atomic operations also execute entirely in phase two, ensuring that atomic reads receive the correct value (Section 2.3). Phase two ends with a second fast barrier.

3.1. Write Cache

Calvin-MIST replaces a traditional store buffer with a structure we call the write cache. Unlike a store buffer, the write cache does not have to maintain program order of stores and can therefore be implemented as a set-associative cache. Like a store buffer, a processor puts all stores into the write cache and subsequent loads bypass from it. Unlike a store buffer, stores in the write cache can be flushed to the L1 in any order since the MIST coherence protocol ensures that memory operations appear in the correct Calvin order regardless of when they are written back. Furthermore, it allows the write cache to coalesce stores.

The write cache keeps all stores private until the stratum's second phase by buffering update values. During phase two, stores move from the write cache to update the local cache and (much less often) coordinate with the directory in the case of a conflict. After flushing the write cache, the processor synchronizes at the second barrier and is ready to begin the next stratum.

Because the write cache is responsible for ensuring that writes remain private in a stratum, Calvin-MIST must handle write cache overflow carefully. In bounded deterministic or conventional modes, it is sufficient to simply end the stratum when an overflow is about to occur since Calvin does not make any guarantees about the actual stratum size in those modes.

In unbounded deterministic mode only, the stratum termination cannot depend on the write cache capacity (which may differ between implementations) and so we use a simple logging technique to logically extend the write cache size. When a store does not fit in the write cache, it is written to a software log in the virtual address space of an application, similarly to how values are remembered in some transactional memory systems [3,29,35]. Additionally, a flag in the corresponding write cache set indicates that an overflow has occurred. On any subsequent miss to that set, a log walk determines if the address is present and, if found, the log entry is treated like a normal entry in the write cache. Access to the log is performed out of band from the standard MIST protocol (Section 3.3) to ensure that reads/writes complete immediately.

3.2. Stratum Termination Function

In Calvin-MIST, the execution mode determines when a processor stops executing instructions and coordinates to terminate a stratum. Let a `STRATUM_LIMIT` register hold a maximum count.

- *Conventional (C) mode* terminates a stratum (a) when the number of *cycles* elapsed in the stratum equals `STRATUM_LIMIT`, (b) a serializing instruction executes (e.g., atomics, I/O), or (c) a processor resource saturates (e.g., the write cache).
- *Bounded Deterministic (BD) mode* terminates a stratum (a') when the number of *instructions* elapsed in the stratum equals `STRATUM_LIMIT`, (b) a serializing instruction executes, or (c) a processor resource saturates (e.g., the write cache).
- *Unbounded Determinism(UD) mode* terminates a stratum (a') when the number of *instructions* elapsed in the stratum equals `STRATUM_LIMIT` or (b) a serializing instruction executes.

C mode minimizes processor idle time, but is not deterministic. BD is deterministic on the same hardware only, as it includes micro-architectural events. UD includes architectural events only.

3.2.1. Stratum Limit Prediction.

As the results in Section 5 will show, different workloads perform best with very different values of `STRATUM_LIMIT`. Workload (phases) with fine-grain synchronization prefer small values to decrease inter-thread communication latency while those with more coarse grain interaction prefer large values to better amortize stratum termination overheads.

To avoid setting `STRATUM_LIMIT` *a priori*, Calvin-MIST uses a standard two-bit predictor to vary `STRATUM_LIMIT` in powers of two between two extremes (e.g, 64-4096 instructions). The predictor decrements when a stratum ends with one or more processors executing an atomic. Strata that end with no atomics increment the predictor. When the predictor saturates high (low), `STRATUM_LIMIT` is doubled (halved) within the extremes. C and BD modes also decrement the predictor for resource exhaustion.

Determining whether to increment/decrement the predictor can be done by piggy-backing a single bit logical-OR reduction on the stratum ending barrier, similar to the wired-OR signal that snooping systems use to determine ownership. The predictor is replicated at each processor and kept in sync by updating only at the end of a stratum.

3.3. MIST Coherence Protocol

Calvin-MIST implements a novel directory coherence protocol, called MIST, to enforce the stratum ordering constraints of the Calvin model. The coherence protocol must ensure two things:(1) that all cache misses return data from the end of the previous stratum and (2) that stores by different processors to the same cache block within the same stratum are ordered deterministically. Furthermore, for performance the protocol should ensure that (3) cache blocks with a single writer should perform comparably to a conventional writeback coherence protocol. To achieve these goals, MIST has several features that distinguish it from more traditional protocols:

Multiple Concurrent Writers. The MIST protocol supports multiple concurrent writers, since multiple threads can store to the same address during a stratum. To ensure deterministic execution, the MIST directory tracks concurrent writers and ensures that their updates are performed in a deterministic order.

Timebomb State. The timebomb state allows readers to coexist with writers in the same stratum. Rather than invalidate blocks when another processor signals intent to write during phase one, the timebomb state allows a processor to retain read permission (to the value from the end of the previous stratum). At the end of the current stratum, the block self-destructs and becomes invalid. The timebomb state eliminates the need to send explicit invalidate messages.

To support both multiple concurrent writers and the timebomb mechanism, the Calvin-MIST protocol interacts with an on-chip 16-cycle hardware barrier [4,10] that communicates stratum boundaries out of band from normal coherence request. To ensure correctness, all outstanding coherence requests must complete before a processor asserts the barrier. Also, as a consequence of allowing multiple writers, MIST implements two-phase stores. Stores are placed in the write cache during phase one and only update the cache hierarchy in phase two. Loads execute entirely during phase one, ensuring that they never see the effect of another processor's store during the same stratum.

Table 1 – *L2 Directory States in MIST.*

State	Meaning	Global Invariant	Valid at
I	Not Present/Invalid	0 readers, 0 writers	Memory
S	One or more readers	1 or more readers, 0 writers	L2 Cache
M	Only one writer	0 or more readers, 1 writer	Processor
MM	No readers/writers	0 readers, 0 writers	L2 Cache
MS	Multiple writers	0 or more readers, 1 or more writers	L2 Cache

3.3.1. Directory States

The directory in Calvin-MIST is split into banks at the last level of cache (L2). It has a bit vector to keep track of either concurrent readers or concurrent writers.

Table 1 lists the MIST directory's five stable states. The MM, M, S, and I states are similar to those in a conventional MSI protocol. A block in the MS state indicates multiple concurrent writers and plays a key role in enforcing the Calvin stratum ordering rules. At the end of a stratum's phase one, the bit vector for a block in the MS state indicates all processors that intend to write the block. The directory uses this information during phase two to determine the order in which those stores complete (Section 2.1's rule 3).

Directory block replacements in MIST are complicated because in doing so the directory forgets which processors are concurrent writers (if any). We add a single replacement bit to each directory *bank* that is set when any block is replaced and is cleared at the end of phase two. When an incoming request misses in the directory while the replacement bit is set, the directory conservatively assumes that it has already seen and replaced that block in the current stratum and initiates a *WhoIsWriter* query. All processors check their write cache for the block and reply either affirmative or negative. Because the query and the DRAM fetch for the missed block occur in parallel, there is almost no latency penalty. Our observations of Calvin-MIST in action indicate that the querying for writers occurs rarely and so is not a concern for performance.

3.3.2. L1 Cache States

MIST is designed for write-back L1 caches in order to minimize communication with the directory. L1 caches in MIST operate on five stable states and one timebomb state, as shown in Table 2. The M, S, and I states are like those in a conventional protocol while the remaining are specific to MIST. Below we will describe each of the remaining stable and timebomb states and how they help MIST enforce the deterministic memory order demanded by the Calvin model.

The Mw state differs from the M state in that it represents a block written in the current stratum, as opposed to one written in a previous stratum. Like M, the Mw state indicates that there are no other writers, allowing the write cache to update the L1 cache (in phase two) without communicating with the directory. The distinction between M and Mw also allows the protocol to correctly detect whether or not a conflicting coherence request indicates multiple writers in the same stratum. Blocks in Mw transition to M in phase 2.

Table 2 – *L1 Cache MIST states*

State	Meaning	Global Invariant
I	Not Present/Invalid	0 or more readers, 0 or more writers
S	Read Permission, no other writers in the system	1 or more readers, 0 writers
M	Write permission, didn't write in current stratum	0 readers, 1 writer
Ts	Read permission until the end of the stratum	1 or more readers, 1 or more writers
Mw	Write permission, wrote in current stratum	0 readers, 1 writer
MMw	Write permission until the end of the stratum	0 or more readers, 2 or more writers

The timebomb state, Ts, corresponds to temporary read permission for a block in the presence of one or more other writers. Data in the Ts state may be read until the end of the stratum, at which point the timebomb self-destructs and the block returns to the I state. The timebomb allows MIST to efficiently handle situations where a processor is reading a block that will be overwritten by another processor's store at the end of the stratum. Without a time-delayed invalidation mechanism, readers in this situation would have to be explicitly invalidated by the directory during phase two. Blocks in Ts are anonymous because the directory bit vector is reused to track both reader and writers; while at least one processor is writing the block the directory cannot keep track of the readers.

Finally, blocks in the MMw state represent data being written by the local processor and at least one other. Stores for blocks in the MMw state will be written back to the directory in phase two so that the store can be correctly ordered. After a store request completes in phase two, a block in MMw transitions to I.

3.3.3. MIST Complexity

Here we compare MIST to a conventional MESI protocol designed for the same base system and an MOESI protocol designed for a multi-chip CMP in an attempt to gauge the complexity of our new protocol. Table 3 shows the number of stable, transient, and total states for each protocol (from Wisconsin GEMS [24]).

Results show that MIST's state count is comparable to MESI and MOESI. Thus, while MIST may seem more complex, in part, because it is unfamiliar, it has comparable complexity.

Table 3 – *The number of states in MIST compared to conventional MESI and MOESI protocols*

	MIST	MESI	MOESI
Stable @ L1	6	4	7
Transient @ L1	12	6	8
Stable @ L2	5	3	13
Transient @ L2	17	14	46
Total	40	27	54

3.4. Example to "Put It All Together"

Figure 4 illustrates Calvin-MIST in action (time goes down) for Processor P0 (left), directory (center), and Processor P1 (right) manipulating one location whose address is omitted.

Stratum S illustrates P1 acquiring write permission in phase one, and then completing the store locally in phase two. A GetM request by P1 (1) acquires write permission and causes P0 to transition into the Ts state. P1 transitions to Mw because it is the only writer. At the end of phase 1, P1 issues a store (2) which transitions the block into the M state. At the end of phase 2, the block in Ts timebomb state at P0 explodes.

Stratum S+1 shows the common case where P1 already has write permission to the block in phase one, and completes the store without communicating with others. Processor P1 can make its intent to write (3) known and write the data (4) without communicating with others.

Stratum S+2 shows how MIST resolves conflicting stores. GetM requests (5) and (6) acquire write permission for processors P0 and P1, and both end up in state MMw. When phase 1 completes, both processors write their data back to the directory (7), (8). At the directory P1 is ordered after P0, so P0's writeback applied (9), while the writeback from P1 is nacked (10). P1 retries the writeback (11), which is then accepted by the directory (12).

3.5. Calvin Hardware Overhead

Compared to a conventional multiprocessor system using in-order pipelines, Calvin-MIST adds only a small number of additional hardware structures. First, the store buffer in a conventional system is replaced with the write cache in Calvin-MIST. Because of Calvin's buffering requirements, the write cache will likely be sized slightly larger than a store buffer in a similar conventional system, but the write cache itself is a simpler structure because it doesn't have to order stores. If unbounded determinism is desired, Calvin-MIST additionally adds a log head and tail pointer to keep track of write cache overflows.

Calvin-MIST also adds a single bit to every L1 cache line to represent the timebomb state. A Calvin-MIST cache must also have the ability to flash clear this bit on the end of a stratum [17]. At each di rectory bank, a single replacement bit is also

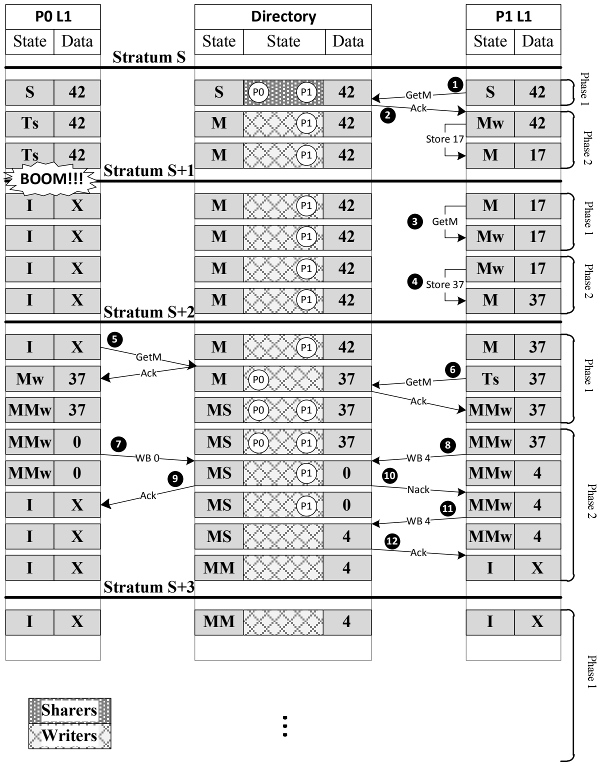


Figure 4 – *Calvin-MIST in action for a block*

introduced so that the directory can know that it may be missing information about outstanding writers (Section 3.3.1).

Calvin-MIST adds a dedicated hardware barrier so that stratum boundaries can be communicated quickly [4,10]. For the predictor, a two-bit counter is added to each core and a global wired-OR line is used to communicate the prediction at the end of each stratum.

3.6. Extensions

While we have described Calvin-MIST in terms of a specific in-order multicore system, the mechanisms could be extended to work with alternative base architectures. In particular, Calvin-MIST can work with out-of-order cores by dealing only with committed store values. In this situation, values in the write cache would hold non-speculative state only.

4. Evaluation Methods

We have implemented Calvin-MIST in an execution driven full system simulator based on Bochs[25] and a modified version of Wisconsin GEMS [24]. We model pipelined in-order x86 processors running 64-bit Linux version 2.6.26. For comparison, we use a base system shown in Figure 2 of Section 3 modeled after the parameters in Table 4.

Table 4 - *System parameters*

	Base	Calvin-MIST
Cores	8, 2.0 GHz in-order pipelined	
Write Cache	N/A	64 entry, 8 way
L1 Cache	Private, Split L1 I&D, 32K 8-way, 1 cycle	
Coherence Protocol	Conventional MOESI	Multiple Writer MIST
Barrier	N/A	16-cycle latency
L2 Cache	Shared, 8MB, 16-way, 8 banks, 12 cycles	
Directory	Distributed at the L2 banks	

We ensure that interrupts appear deterministically across runs of the same program in our simulated system by (a) restricting interrupt injection to stratum boundaries and (b) by ensuring that interrupts occur after a well-defined amount of logical time has passed. For example, when an inter-processor interrupt (IPI) is sent from one processor to another, we ensure that the interrupt will be received in the stratum after a set number of instructions have completed. Similarly, we ensure that input instructions always receive the same value by starting the system from a checkpointed state and by ensuring that our device models are deterministic.

To help verify that Calvin-MIST does indeed enforce a deterministic execution, we used the Racey microbenchmark that is exceedingly sensitive to the order of unsynchronized data accesses [19]. The Racey program produces a signature that has a high probability of changing under different race outcomes. We have observed hundreds of runs of Racey on Calvin-MIST produce the same signature, even when introducing frequent random network delays, lending strong evidence (though not proof) that our implementation is correct.

We evaluate Calvin-MIST using the Parsec 2.0 [6] and HPC Mantevo [1] workload suites. Some workloads from Parsec and Mantevo are not included in the results due to a combination of compilation issues and simulator constraints. For all Parsec workloads, we use the simsmall input set.

5. Evaluation Results

These results ask and answer two questions and then perform some sensitivity analysis.

Question 1: Can Calvin-MIST avoid harm? Yes, Calvin-MIST executes nondeterministic programs at speeds slightly worse than a conventional system, thereby maintaining performance compatibility.

Question 2: Can Calvin-MIST do some good? Yes, Calvin-MIST executes deterministic programs at a performance overhead less than 20%, thereby providing a benefit when deterministic is valued.

5.1. Bottom Line: Calvin Performance

In Figure 5 we compare the performance of Calvin-MIST to our baseline system and find that on average Calvin-MIST performs with a modest degradation (8%) to the baseline in conventional mode and sees around a 20% slowdown for both deterministic modes.

Calvin-MIST facilitates adoption by providing functional and performance compatibility with nondeterministic workloads. There are many reasons why Calvin-MIST could perform comparably to the baseline system even with the overhead of a two-phase execution. For one, Calvin-MIST reduces the impact of false sharing by allowing multiple simultaneous writers and by delaying reader invalidation. Delayed invalidation has previously been shown to reduce the negative impact of false sharing [12] and improve the performance of critical sections [36]. Second, other results (not shown) indicate that several of the Parsec workloads benefit from the coalescing effect of the write cache. Third, the simple strata size predictor used by Calvin-MIST dynamically detects application synchronization and communication patterns, limiting load imbalance within a stratum.

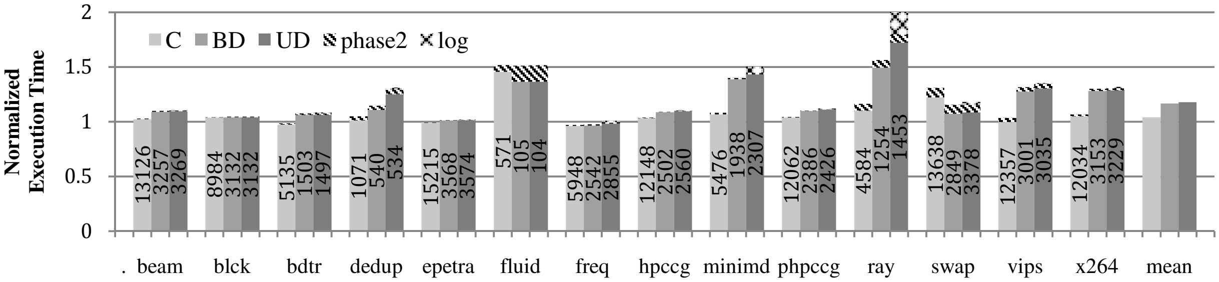


Figure 5 – *Calvin-MIST performance using stratum limit prediction. We show the execution time normalized to our baseline for C, BD, and UD modes. For each data point, we show the average stratum limit over the run, in number of cycles for C and number of instructions for BD and UD, that the predictor chose. Also, the stack segments of each bar show how much time is spent in phase one (shaded), phase two (black), and accessing the overflow log (light grey, nearly negligible).*

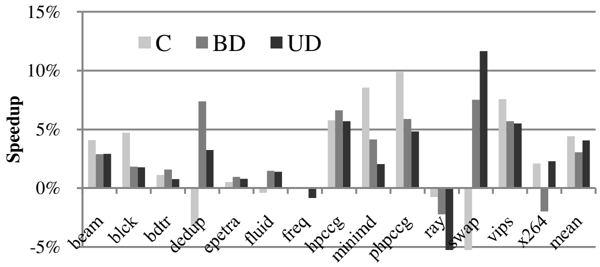


Figure 6 – *Prediction Effectiveness, as compared to statically chosen stratum limits. Higher indicates better effectiveness.*

Some workloads perform slightly worse in conventional mode Calvin-MIST. There are at least three causes of this slowdown. First, the conservative 16-cycle barrier we modeled in Calvin-MIST has a noticeable impact when small stratum limits are used, such as in Fluidanimate. Results, not shown, with a 4-cycle barrier largely mitigates the slowdown. Second, even though the conventional stratum termination function tries to mitigate the impact of load imbalance, a processor cannot enter the barrier until all outstanding instructions have completed. Thus a cache miss on one processor just before phase one is scheduled to end can cause all processors to stall until it completes. Finally, inter-thread communication through shared memory is delayed when running in Calvin because threads cannot communicate within a stratum. Workloads that exhibit frequent fine-grained locking, such as Fluidanimate, are affected by this communication delay.

The deterministic modes are somewhat slower than conventional mode because in the deterministic modes the speed of each stratum as a whole is limited by the slowest running processor. Thus, if one processor is frequently missing to main memory it will slow down the entire system. Calvin-MIST experiences an average (geometric mean) slowdown of around 20% over the baseline in both deterministic modes.

5.2. Execution Breakdown

Figure 5 also shows the breakdown of each execution into time spent in normal execution (phase one), flushing the write cache (phase two), and, in the case of unbounded deterministic mode, time spent overflowing the write cache to the software log.

As expected, most time is spent in phase one. The effect of flushing the write cache is small because for data race free programs, the only store conflicts that occur are due to false sharing in cache blocks. Thus, most stores in phase two are L1 cache hits.

To gauge the performance impact of unbounded determinism support, we calculated the effect of overflowing the write cache by charging an L1 miss (17 cycles) for each read/write to the log. We find that for most workloads, the impact of log access is negligible.

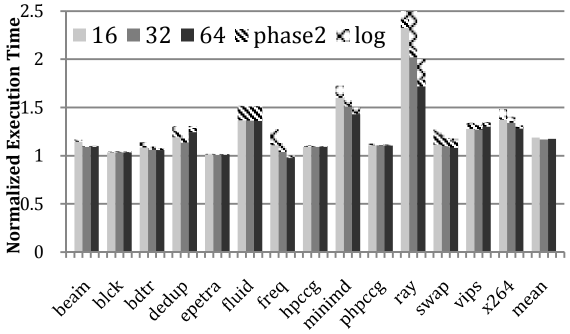


Figure 7 – *Write cache sensitivity analysis for UD mode. Results are normalized to a conventional MESI protocol.*

5.3. Prediction Effectiveness

We tested the accuracy of the Calvin-MIST stratum limit predictor by comparing the execution time using the predictor to a run that uses a best-case static stratum limit. We tested static stratum limits between 64-2048 instructions for deterministic mode and 100-20,000 cycles for conventional mode, and then selected the size that resulted in the best performance.

Figure 6 shows the speedup of the system using a predictor over one using static stratum limits. The predictor performs better in all but four workloads, most likely because the predictor is able to capture phase behavior over the course of a run. Workloads that perform better with static stratum limits may exhibit patterns not captured by the predictor, such as communication through a flag without the use of atomics.

5.4. Write Cache Sensitivity Analysis

Next we varied the write cache size among 16, 32, and 64 64-bytes entries (1, 2, and 4 KB). Figure 7 shows the results of this analysis, and indicates that the write cache does not need to be large for good performance in our workloads. We also varied the associativity between 4 and 8 ways (not shown) and found that associativity has a negligible effect on performance.

Our results show that systems with unbounded determinism support are more sensitive to write cache size than systems configured for bounded determinism due to log accesses. This is illustrated by the execution breakdown in Figure 7, in which looking at each bar without the final stack for log accesses closely approximates results for bounded determinism.

5.5. Frequency of Writeback Messages

Calvin-MIST generates extra writeback messages whenever two or more processors write the same cache block in the same stratum, since the directory must

Table 5 - *HPCCG 1024 instructions/stratum, bounded determinism*

CPU	0	1	2	3	4	5	6	7
Insn cnt (M)	235	235	235	235	235	235	235	235
Total WB (K)	517	548	547	549	553	549	514	549
Extra WB	235	328	374	294	426	357	206	292
Extra Nacks	1	6	3	3	6	4	0	17

resolve the multiple writes in the correct order. Fortunately, as Table 5 shows for a representative benchmark--the HPCCG benchmark running in bounded determinism mode with 1024 instruction strata--these extra writebacks occur rarely, *three orders of magnitude less often than regular writebacks*. And since extra writebacks are rare, the directory almost never get nacks them (as may be necessary to ensure correct write ordering).

These results are typical because most well-written programs are data-race-free, and thus will not store to a shared variable outside a critical section. Because of how Calvin handles atomic operations, critical sections are entered by at most one thread per stratum. Thus, extra writebacks will generally only occur due to false sharing, which is also relatively rare in well-constructed software.

6. Future Work

As described so far, Calvin is applied at the system level. With modification, Calvin can also be applied in isolation to different domains running on the same machine, similar to how Capo virtualizes deterministic replay [28]. For example, it could execute one virtual machine deterministically and another conventionally in a consolidated workload environment. Future work will address the difficulties that could arise in a multiple-domain environment, such as making the hypervisor invisible to the execution domain.

Future work may also address scalability concerns of the Calvin-MIST implementation, particularly focusing on the barrier bottleneck. It is important to note that although Calvin-MIST uses a barrier, it is not strictly necessary to meet Calvin requirements.

Finally, we may investigate methods to improve the performance of Calvin-MIST's conventional mode. For example, it may not be necessary to wait at a barrier in conventional mode and some protocol states could be optimized.

7. Related Work

The work most similar to Calvin is the CoreDet compiler infrastructure by Bergen, et al. [5]. CoreDet and Calvin both share the same insight that the TSO memory model can be exploited to provide determinism, and both execute programs as a series of multiphase strata. CoreDet is implemented entirely in software, though, whereas Calvin is a hardware memory model. Thus, the tradeoffs between the two are similar to other proposed mechanisms that can be implemented in either software or hardware, such as transactional memory systems [26]. The CoreDet runtime overhead varies between 1-11x whereas Calvin can execute with less than 0.5x and 20% on average overhead for all workloads.

Deterministic Shared Memory Multiprocessing (DMP) [11] deterministically serializes execution quanta from each processor so that only one ordering is possible. They use Bulk's transactional memory that broadcasts signatures to achieve parallelism among quanta by speculatively executing then rolling back if a conflict occurs. While DMP posts results similar to Calvin, they exclude privileged instructions from their evaluation, which we found to be a significant impact on performance.

Kendo [32] proposes a software-only solution for achieving weak determinism, in which a program is repeatable only if it is data-race-free. Kendo uses a custom library for locks that ensures locks are always acquired in the same order, and experiences a 16% performance overhead. Calvin provides strong determinism at a similar performance cost but requires hardware support.

Other work in determinism has focused on a two-phase record/replay approach [13,20,27,30,31,34,42]. These proposals supply hardware support for recording inputs and memory race outcomes to a log that is used later to replay an execution verbatim. Calvin does not require a recording phase, and instead guarantees that only one outcome exists given a program and inputs.

Strata [30] is a proposal for deterministic record/replay, which, as the name suggests, bears similarities to Calvin's stratified execution. Strata is designed for a system with sequential consistency, whereas Calvin can take advantage of the implementation optimizations afforded by TSO.

Programming languages exist [7,15] that guarantee deterministic execution by limiting how actors communicate. A Calvin system can run a program deterministically regardless of the language or communication pattern.

The UltraSPARC IV [22] contained a unit called the write cache that served as a coalescing buffer in the memory system. The UltraSPARCIV's write cache was between the L1 and L2, though, and would only place blocks into the cache once they were globally ordered. Calvin's write cache, on the other hand, sits between the processor and the L1 and inserts blocks before they are ordered in the memory system.

The Wisconsin Wind Tunnel [37] was a discrete event simulator that used a concept resembling Calvin's strata called a quantum. Two threads could not communicate in a quantum, which allowed for performance optimizations. However, WWT could only simulate a sequentially consistent execution and, because it was fundamentally cycle accurate simulator, did so considerably slower than Calvin.

The timebomb state in Calvin-MIST resembles the concept of tear-off blocks proposed by Lebeck and Wood [23]. However, blocks in the timebomb state are invalidated deterministically whereas tear-off blocks are not. Two groups have also previously made the observation that delaying an invalidation for a small amount of time can actually improve performance by reducing the effect of false sharing [12] and by leading to better lock behavior under high contention [36]. Unlike Calvin, neither delays the invalidation by a deterministic amount of time.

8. Conclusions

We propose Calvin, a system that can execute in one of three modes: conventional nondeterministic, bounded deterministic, and unbounded deterministic. Depending on application requirements, Calvin implementations can switch among modes by adjusting the stratum termination condition. We show that Calvin running in conventional mode has minimal overhead compared to the baseline and may outperform the baseline. Calvin systems execute deterministically with low performance overhead and no speculation.

9. Acknowledgements

We thank Dan Gibson, Mike Swift, the Wisconsin Multifacet group, the anonymous reviewers, and the Wisconsin Computer Architecture Affiliates for their comments and/or proofreading. Finally we thank the Wisconsin Condor project and the UW CSL for their assistance.

This work is supported in part by the National Science Foundation (CNS-0551401, CNS-0720565 and CNS-0916725), Sandia/DOE (#MSN123960/ DOE890426), and University of Wisconsin (Kellett award to Hill). The views expressed herein are not necessarily those of the NSF, Sandia or DOE. Hill and Wood have a significant financial interest in Microsoft. Dudnik is now at Google.

10. References

[1] Mantevo Project. https://software.sandia.gov/mantevo/.

[2] Allen, M.D., Sridharan, S., and Sohi, G.S. Serialization sets: a dynamic dependence-based parallel execution model. *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ACM (2009), 85-96.

[3] Ananian, C.S. et al. Unbounded Transactional Memory. *The 11th International Symposium on High-Performance Computer Architecutre (HPCA)*, (2005).

[4] Beckmann, C.J. and Polychronopoulos, C.D. Fast barrier synchronization hardware. *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, IEEE Computer Society Press (1990), 180-189.

[5] Bergan, T. et al. CoreDet: a compiler and runtime system for deterministic multithreaded execution. *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ACM (2010), 53-64.

[6] Bienia, C. et al. The PARSEC benchmark suite: characterization and architectural implications. *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ACM (2008), 72-81.

[7] Bocchino Jr, R.L. et al. A type and effect system for deterministic parallel Java. *ACM SIGPLAN Notices 44*, 10 (2009), 97–116.

[8] Bocchino, R.L. et al. Parallel Programming Must Be Deterministic by Default. *HotPar-1: First USENIX Workshop on Hot Topics in Parallelism*, (2009).

[9] Bressoud, T.C. and Schneider, F.B. Hypervisor-based Fault Tolerance. *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating Systems Principles*, (1995).

[10] Culler, D.E. and Singh, J. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., 1999.

[11] Devietti, J. et al. DMP: Determinisitc Shared Memory Multiprocessing. *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, (2009), 85--96.

[12] Dubois, M. et al. Delayed consistency and its effects on the miss rate of parallel programs. *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, ACM (1991), 197-206.

[13] Dunlap, G.W. et al. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, (2002), 211-224.

[14] Dunlap, G.W. et al. Execution Replay on Multiprocessor Virtual Machines. *International Conference on Virtual Execution Environments (VEE)*, (2008).

[15] Edwards, S.A., Vasudevan, N., and Tardieu, O. Programming Shared Memory Multiprocessors with Deterministic Message-Passing Concurrency: Compiling SHIM to Pthreads. *Proc. of the Conference on Design, Automation, and Test in Europe*, (2008), 1498-1503.

[16] Frigo, M. Multithreaded programming in Cilk. *Proceedings of the 2007 international workshop on Parallel symbolic computation*, ACM (2007), 13-14.

[17] Hammond, L. et al. The Stanford Hydra CMP. *IEEEMICRO 20*, 2 (2000), 71-84.

[18] Hangal, S. et al. TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model. .

[19] Hill, M.D. and Xu, M. *Racey: A Stress Test for Deterministic Execution*. .

[20] Hower, D.R. and Hill, M.D. Rerun: Exploiting Episodes for Lightweight Race Recording. *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, (2008), 265-276.

[21] Kongetira, P., Aingaran, K., and Olukotun, K. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEEMICRO 25*, 2 (2005), 21-29.

[22] Krewell, K. UltraSPARC IV Mirrors Predecessor. *MICROREPORT*, (2003), 1-3.

[23] Lebeck, A.R. and Wood, D.A. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. *Proceedings of the 22nd annual international symposium on Computer architecture*, (1995), 48-59.

[24] Martin, M.M.K. et al. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, (2005), 92-99.

[25] Mihocka, D. and Swartsman, S. Virtualization without direct execution - designing a portable VM. *The 1st Workshop on Architectural and Microarchitecrual Support for Binary Translation*, (2008).

[26] Moir, M. *Hybrid Transactional Memory*. 2006.

[27] Montesinos, P., Ceze, L., and Torrellas, J. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. .

[28] Montesinos, P. et al. Capo: A Software-Hardware Interface for Practical Determinisitic Multiprocessor Replay. *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, (2009), 73--84.

[29] Moore, K.E. et al. LogTM: Log-Based Transactional Memory. *Twelfth IEEE Symposium on High-Performance Computer Architecture*, (2006), 258-269.

[30] Narayanasamy, S., Pereira, C., and Calder, B. Recording Shared Memory Dependencies Using Strata. *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, (2006), 229-240.

[31] Narayanasamy, S., Pokam, G., and Calder, B. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. *Proceedings of the 32nd annual international symposium on Computer Architecture*, (2005), 284-295.

[32] Olszewski, M., Ansel, J., and Amarasinghe, S. Kendo: Efficient Deterministic Multithreading in Software. *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, (2009).

[33] Prvulovic, M. CORD: Cost-effective (and nearly overhead-free) Order Recording and Data race detection. .

[34] Prvulovic, M. and Torrellas, J. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. *Proceedings of the 30th Annual International Symposium on Computer Architecture*, (2003), 110-121.

[35] Rajwar, R., Herlihy, M., and Lai, K. Virtualizing Transactional Memory. *Proceedings of the 32nd annual international symposium on Computer Architecture*, (2005).

[36] Rajwar, R., Kägi, A., and Goodman, J.R. Improving the Throughput of Synchronization by Insertion of Delays. *Proc. of the 6th International Symposium on High-Performance Computer Architecture (HPCA)*, (2000), 168-179.

[37] Reinhardt, S.K. et al. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, (1993), 48-60.

[38] RTI. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. 2002.

[39] Seiler, L. et al. Larrabee. *ACM Transactions on Graphics 27*, 3 (2008), 1.

[40] Weaver, D.L. and Germond, T., eds. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.

[41] Xu, M., Bodik, R., and Hill, M.D. A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording. 49-60.

[42] Xu, M., Bodik, R., and Hill, M.D. A “Flight Data Recorder” for Enabling Full-system Multiprocessor Deterministic Replay. *Proceedings of the 30th annual international symposium on Computer architecture*, (2003), 122-133.

[43] Yu, J. and Narayanasamy, S. A case for an interleaving constrained shared-memory multi-processor. *SIGARCH Comput. Archit. News 37*, 3 (2009), 325-336.

Appendix A Proof of Calvin-TSO Compatibility

For ease of presentation, we discuss only loads and stores and ignore fairness.

TSO. Weaver and Germond formally define the TSO memory model in their Appendix D [18,40] using the following notation: La and Sa represent a load and a store, respectively, to address a. Orders <p and <m define program and global memory order, respectively. For TSO:

(1) Each of P processors inserts its loads and stores into global memory order <m preserving program order <p between two loads, two stores, and a load then a store (but not necessarily a store then a load).

The value returned by each load La is given by:

(2) Value(La) = Value (Max$_{<m}$ { S | Sa < m La or Sa <p La })

Intuitively, this dense equation means that load La gets its value from the last store that has updated coherent memory “Sa <m La” unless there is a later store that the load will bypass from the processor’s store buffer “Sa <p La”.

Calvin. Calvin logically constructs a global memory order <m by partitioning the loads and stores in program order <p into strata using the following rule:

(a) For each stratum ***S***, all memory operations in stratum ***S*** are ordered in <m, such that they are after all the memory operations of stratum ***S*** -1 and before all the memory operations of stratum ***S*** +1.

Moreover, Calvin orders memory operations within each stratum ***S*** as follows:

(b) Each processor i inserts its loads into global memory order <m preserving program order <p and ordered after all loads from processor i-1 and before all loads from processor i+1,

(c) Processor 1 inserts its stores into global memory order <m ordered after all loads from processor P.

(d) Each processor i inserts its stores into global memory order <m preserving program order <p and ordered after all stores from processor i-1 and before all stores from processor i+1.

Thus, Calvin constructs a global memory order <m compatible with TSO Rule (1). Since Calvin also implements store buffer bypassing, it implements TSO Rule (2). Therefore, Calvin is compatible with TSO.