In *IEEE Int'l Symposium on Performance Analysis of Systems and Software (ISPASS)*, Austin, TX, March 2005.

Accelerating Multiprocessor Simulation with a Memory Timestamp Record

Kenneth C. Barr, Heidi Pan, Michael Zhang, and Krste Asanović

MIT Computer Science and Artificial Intelligence Laboratory*
The Stata Center, 32 Vassar Street, Cambridge, MA 02139
{kbarr, xoxo, rzhang, krste}@csail.mit.edu

Abstract

We introduce a fast and accurate technique for initializing the directory and cache state of a multiprocessor system based on a novel software structure called the memory timestamp record *(MTR). The MTR is a versatile, compressed snapshot of memory reference patterns which can be rapidly updated during fast-forwarded simulation, or stored as part of a checkpoint. We evaluate MTR using a full-system simulation of a directory-based cache-coherent multiprocessor running a range of multithreaded workloads. Both MTR and a multiprocessor version of functional fast-forwarding (FFW) make similar performance estimates, usually within 15% of our detailed model. In addition to other benefits, we show that MTR has up to a 1.45× speedup over FFW, and a 7.7× speedup over our detailed baseline.*

1 Introduction

Computer architects rely heavily on simulators to evaluate, refine, and validate new designs before implementation. However, simulation time continues to increase as microarchitectures become more complex and multicore designs become more common. The intricate timing-dependent behavior of these machines cannot be accurately captured with trace-driven simulation, thus more expensive execution-driven simulation is typically used.

To reduce simulation time, overall behavior can be estimated using short samples taken from a complete application run. Many published architecture studies have obliviously chosen a single sample, either taken from the beginning or after some fixed number of instructions into the run. But applications generally contain multiple phases of execution with varying properties and much better characterization is possible by using multiple sample points spread throughout a run [5, 11, 21, 27]. Two alternative sampling implementations are commonly used with execution-driven simulators: (1) *Fast-forwarding* uses a fast functional simulator to update the architectural state (registers and memory) in between sampling points, then switches to a slower detailed simulator to accurately model the microarchitecture during the measurement samples. (2) *Checkpointing* initializes the detailed simulator at each sampling point using a saved copy of the architectural state. Checkpoints can be created using functional simulation or by interrupting execution of the application on a real machine.

The performance of modern microprocessors is greatly dependent on large quantities of microarchitectural state, such as branch predictors and caches, which must be initialized correctly at each sample point to avoid large systematic errors. Two common approaches to initialize this state are: 1) *detailed warming*, where the detailed model is run for a period before measurements are taken to warm up caches and predictors, and 2) *functional warming* where important microarchitectural state is updated while fast-forwarding and then used to initialize the detailed simulator [27]. Microarchitectural state is less amenable to checkpointing, as it is often the microarchitecture that is varied across experiments. Some industry development groups report detailed warming runs require up to two weeks when starting from stored architectural checkpoints. Modern directory-based, cache-coherent multiprocessors have an even larger quantity of microarchitectural state, including the directory and multiple large caches. The long histories of these structures makes detailed warming impractical, and even functional warming has significant overhead when maintaining coherent caches.

In this paper, we present a fast and accurate technique for initializing the directory and cache state of a multiprocessor system based on a novel software structure called the *memory timestamp record* (MTR). The MTR is a versatile, compressed snapshot of memory reference patterns which can be rapidly updated during fast-forwarding, or stored as part of a checkpoint. During fast-forwarding, instead of maintaining the directory and cache state for functional warming, the MTR simply records the time of every processor's

*This work was partly funded by the DARPA HPCS/IBM PERCS project, NSF CAREER Award CCR-0093354, and an Intel Fellowship.

last access to every memory block. This bookkeeping adds little overhead to functional simulation, yet the MTR can quickly and accurately reconstruct cache and directory state largely independent of size, organization, or protocol. We show that MTR achieves an average speedup of 1.19-1.45 over conventional fast-forwarding with functional warming (FFW) for a single cache configuration. Additional speedup is possible when multiple different cache organizations are reconstructed at each sample point. We show we can simulate around several different configurations using MTR in the same time as one run with FFW. Finally, both MTR updates and MTR cache reconstruction are highly parallelizable, thus easily supporting parallel-hosted simulation.

2 MTR Design

This section describes the conceptual structure of the MTR and how it accomplishes the dual goals of fast update and versatile reconstruction.

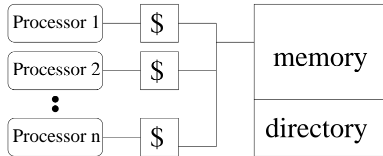

Figure 1. A simple SMP model.

To illustrate the operation of MTR, we use the simple symmetric multiprocessor model shown in Figure 1 as our simulation target. Every processor has a local L1 cache, each having the same cache parameters (e.g. size, associativity) and using an LRU replacement policy. The memory uses a centralized full-map bit-vector directory and the MSI write-back invalidation protocol to support sequential consistency. The directory is always notified when dirty blocks are written back, whereas clean blocks may be silently evicted without informing the directory. A single block size is used for both caches and memory. Alternative cache organizations, coherence, and eviction policies are explored in Section 3.

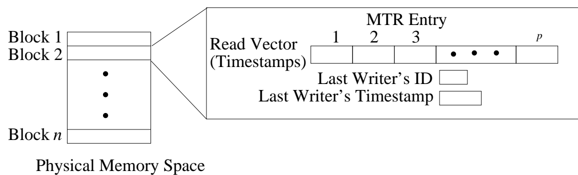

Figure 2. A Memory Timestamp Record.

2.1 MTR Structure

During fast-forwarding, MTR reduces the amount of work per memory access to a simple and fast table update, deferring cache state reconstruction until the transition into detailed mode. The key observation is that directory and cache state can be reconstructed if, for each memory block, we know about each processor's latest accesses and the relative order of these accesses across processors. During MTR fast-forwarding, each simulated processor reads and writes a shared "magic memory" for instant resolution of loads and stores. We also capture the most recent memory accesses using the MTR structure shown in Figure 2. The MTR has an entry for every memory block. Each block's entry contains an array of read timestamps, one per processor, indicating the last time each processor read the block. Additional fields record the ID of the last processor to modify the block and the timestamp of the write. Each timestamp reflects the cycle in which the memory request was issued.

2.2 Fast-Forwarding Phase

During fast-forwarding, whenever any processor issues a read or write request, the MTR is updated using the algorithm shown in Figure 3. A read request for block $address$ issued by processor cpu will record the current timestamp in the corresponding read timestamp. A subsequent read of the same block by the same processor will overwrite the previous read's timestamp. A write request updates the MTR entry's writer ID and timestamp. We also execute UPDATE during detailed simulation to keep the MTR consistent with directory and cache state at all times, avoiding a costly synchronization after every detailed period. The lightweight MTR update has little effect on detailed simulation speed.

```
Update(time, address, isStore, cpu) {
   MTR[address].readers[cpu] = time
   if(isStore) {
      MTR[address].writer = cpu
      MTR[address].writetime = time
   }
}
```

Figure 3. MTR updates during fast-forwarding.

2.3 Cache Reconstruction

At each detailed sampling point, the cache and directory state must be quickly reconstructed. Cache reconstruction is split into two phases. First, we filter the latest memory accesses recorded in the MTR to determine the subset that are still cached based on cache size and associativity. Second, we examine cross-processor interactions to determine which of the cached blocks should still be valid and/or dirty, according to the cache protocol.

To determine the cached subset, we observe that for k-way set-associative caches, an LRU policy dictates that only the last k accessed blocks remain cached in each set. To compare memory accesses that map to the same cache

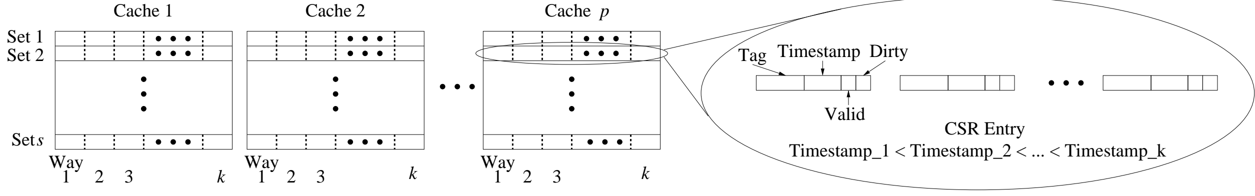

Figure 4. A Cache Set Record (CSR) entry.

set, we reorganize the information in the MTR into a separate structure called the *cache set record* (CSR), shown in Figure 4. The CSR contains an entry for each set in every cache, holding a timestamp-sorted list of the k most recent memory accesses to that set. Figure 5 describes how to fill the CSR by sorting all the memory accesses mapping to the same cache set. To avoid inserting both a read and write timestamp for the same block into the CSR entry, the routine checks if a processor was the last writer. Note that in COALESCECACHEBLOCKS, we insert all memory accesses into the CSR, whether they are valid or not. Although some of these cached blocks may be invalidated later by the cache protocol, they were valid when brought into the cache and potentially caused evictions. As described below, we only examine memory blocks that have been accessed, so the runtime of this procedure is $O(\text{touched lines} \times \text{NUMCPUS} \times k)$.

In the second phase, we step through the CSR to determine the valid and dirty bits of these cached blocks using the algorithm FIXUPCACHES shown in Figure 6. The state of a cached block is dependent on other processors' accesses to the same block, so we need to refer back to the corresponding MTR entry to determine when it was last read and written. A cached block can only be valid if it was cached upon or after the last write. A modified cached block only remains dirty if no other processors have read the data since the modification, otherwise it would have been downgraded to shared status by the subsequent read request. To check for such downgrades, we use the simple ISCLEANSHARED test shown in Figure 6. This results is a worst case runtime for FIXUPCACHES of $O(\text{single cache size} \times \text{NUMCPUS}^2)$.

```
CoalesceCacheblocks(CSR, MTR) {
  for each entry, i, in MTR {
    set = i.address >> SETSHIFT
    for(p = 1 to NUMCPUS) {
      if(i.readtime[p] is valid and p != writer)
        Insert(CSR[set][p], i.tag, i.readtime[p])
    }
    if(i.writetime is valid)
      Insert(CSR[set][writer], i.tag,
             MAX(i.readtime[writer], i.writetime))
} }
```

Figure 5. Building the CSR from the MTR.

```
IsCleanShared(MTRentry i) {
  if(i.writetime is valid) {
     for(p = 1 to NUMCPUS) {
        if(i.writer != p and readtime[p] > i.writetime)
           return true
     }
     return false
  }
  else
     return true
}

FixupCaches(CSR, MTR) {
  for each cached block b in CSR {
    lastwriter = MTR[b.address].writer
    lastwritetime = MTR[b.address].writetime
    if(b.timestamp < lastwritetime) {
       b.valid = false      /* invalidated read */
       b.dirty = false
    }
    else {
       b.valid = true
       if(IsCleanShared(MTR[b.address])
          b.dirty = false   /* downgraded write
                               or cached read   */
       else
          b.dirty = true    /* cached write     */
}}}
```

Figure 6. Reconstruct cache valid and dirty bits by examining cross-processor interactions.

2.4 Directory Reconstruction

MTR directory reconstruction is similar to cache reconstruction, as shown in Figure 7. If a block is read but never written, or read by another processor after the last write, the block is shared. The sharers consist of the last writer (if any) and all subsequent readers. Although some of these sharers may have already evicted their clean copy of the block, they remain in the sharing vector under the silent drop policy, whereas the directory is always notified of dirty writebacks. Thus, we must verify that a dirty copy is still in the cache before marking it modified, otherwise it is invalid. All unrequested blocks are invalid. This procedure's worst case runtime is $O(\text{touched lines} \times \text{NUMCPUS}^2)$.

The reconstruction time would be extremely high if we had to reconstruct the entire directory and cache state during every transition from fast-forwarding to detailed mode. Luckily, only a small subset of the memory locations and cache entries are accessed in each fast-forwarding period, so we only have to apply the reconstruction algorithms to this

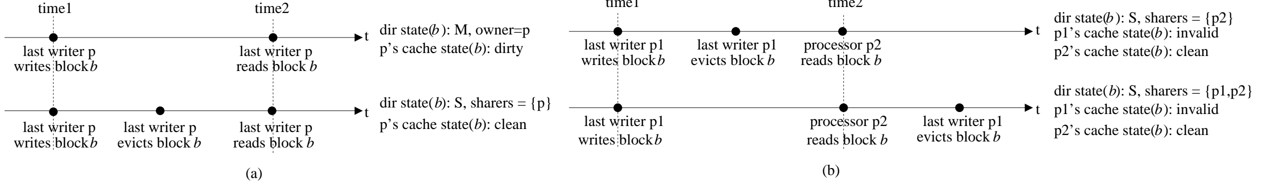

Figure 8. Ambiguities from evictions. The top and bottom lines in each of (a) and (b) show two alternative sequences of events that result in the MTR holding the same two timestamped accesses at time 1 and time 2. These ambiguities require additional effort to resolve, though a quick heuristic is usually adequate.

```
CreateDirectoryFromMTR(directory, MTR) {
  for each entry, i, in MTR {
    address = addressOf(i)
    directory[address].state = Invalid
    if(isCleanShared(i)) {
       directory[address].state = Shared
       for(p = 1 to NUMCPUS) {
         if(i.readtime[p] >= i.writetime)
            directory[address].addSharer(p)
    } }
    else if(i.writetime is valid)
      if(isValidInCSR(address, i.writer)) {
        directory[address].state = Modified
        directory[address].owner = i.writer
} } }
```

Figure 7. Reconstruct directory state in a system with silent evictions.

subset. We add two levels of valid bits to the MTR, which are updated during fast-forwarding to track which memory regions and which blocks within these regions have been touched. We can then quickly skip untouched MTR entries during reconstruction. In addition, since we only examine the most recent memory accesses, the CSR will only reflect the *changed* portion of the cache state and needs to be merged with the cache state from the end of the last detailed sample (which has become stale over the course of fast simulation). During the merge, we must also update the directory to reflect involuntary eviction of stale cache entries.

2.5 Handling Ambiguous Cases

There are scenarios where the MTR timestamp information is insufficient to distinguish between multiple different directory and cache states. For example, consider Figure 8(a), where a processor writes then reads a block without other processors' interference. Given the write and read timestamps, one can infer either of the following scenarios: (1) the read request results in a cache hit and the data remains modified, or (2) the data is evicted and written back in between, so the read request brings the data back in a clean shared state. Another ambiguous example is depicted in Figure 8(b). From MTR reconstruction, we know that a line is first written by processor $p1$ then read by processor $p2$, and that the line has been evicted from $p1$'s cache. If $p1$ evicts the block before $p2$ reads it, $p1$'s copy would have been dirty, so $p1$ would have to write back the data. On the other hand, if $p1$ evicts after $p2$'s read, $p1$'s block would have already been downgraded to a clean shared status, so it can be silently dropped. The directory would include $p1$ in the sharing vector in the first case, but not in the second.

These ambiguities arise because MTR only sees loads and stores, not evictions. There are two approaches to resolving these ambiguities. The faster and less accurate approach is to always reconstruct the directory and cache state to reflect the more probable scenario. In the first example described above, we mark the block as M, assuming good locality and no communication misses such that the data is not evicted. In the second example, we leave $p1$ in the sharing vector, assuming that the block remains in $p1$'s cache long enough to be downgraded and written back before being evicted.

The more accurate approach distinguishes the ambiguous scenarios by pinpointing eviction times, i.e. determining when an owner's copy, installed at $time1$, is evicted before a later access, at $time2$, to the same memory block. A block b is evicted from the set of a k-way associative cache when the processor accesses k new blocks mapping to the same set after it last accesses b. If we re-examine the MTR and find at least k such accesses between $time1$ and $time2$, b was definitely evicted in that period.

When the MTR entry contains fewer than k other accesses within the time frame, we cannot reach a conclusion about the eviction time — there might have been more memory accesses within that time frame that are not recorded by the MTR. For example, a block cached before $time2$ that causes block b to be evicted may have been accessed again after $time2$, so the previous access would have been overwritten in the MTR. This special case of unresolvable ambiguity is the tradeoff for being able to maintain so little state in the MTR. However, as we will show in Section 4, the unresolved ambiguities are rare, and have little effect on overall accuracy.

Figure 9 depicts another potentially ambiguous case caused by unknown eviction times. Assume the caches in

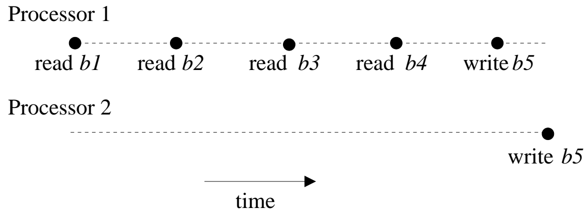

Figure 9. Scenario illustrating why write accesses are recorded in the MTR as read-modify-write. Assume a 4-way associative cache, and blocks $b1 - b5$ map to the same set. We need to retain the time of P1's write to $b5$ to reconstruct the eviction of $b1$.

the figure are 4-way associative, and that all the memory accesses depicted map to the same set. Processor 1's write to block $b5$ causes $b1$ to be evicted, but the write request is overwritten in the MTR by Processor 2's write to the same block. However, unlike the ambiguities described previously, this type of ambiguity is easily resolved. During the initial MTR update, the read vector is also updated upon every write request as if the request were a read-modify-write. In this specific example, the read-modify-write update allows the read vector to retain knowledge of P1's access to $b5$ for reconstructing the eviction. Note that marking writes as read-modify-writes preserves the correctness of the MTR algorithm, because the final directory and cache state for read-modify-writes is indistinguishable from the final state for writes. If no other processor accesses the same block afterwards, the write timestamp indicates that the block is modified. If another processor writes to the same block, the previous owner's copy becomes invalid, no matter if it is a clean-shared or modified copy. Similarly, if another processor reads the same block, the previous owner's copy becomes clean-shared.

3 Extensions of MTR

In this section, we describe how MTR can be used to reconstruct many different cache configurations and coherence protocols. We also discuss the use of MTR within a checkpointing simulator, and we outline how both MTR updates and reconstruction can be parallelized.

3.1 Alternative Cache Configurations

The MTR structure is independent of the simulated cache parameters, with each MTR entry corresponding to a memory block. Thus, MTR can support multiprocessor counterparts of earlier uniprocessor multiconfiguration cache simulation techniques [22, 9], allowing a single fast-forwarding run or a stored MTR checkpoint to capture the behavior of multiple different multiprocessor cache configurations. The state of any sized cache with any associativity can be reconstructed by time sorting touched memory blocks into the appropriate cache sets; cache block sizes that are a multiple of the MTR block size can be easily constructed by merging timestamps from all constituent MTR blocks. In addition, the MTR structure can reconstruct most common forms of time-based replacement policy (e.g., LRU or cache-decay counters).

Multi-level cache hierarchies can be supported by reconstructing the largest cache with MTR and using detailed warming or a further reconstruction for the smaller caches. To reconstruct a non-inclusive cache hierarchy, we first reconstruct the inner caches then omit these cached lines while reconstructing the CSR for the outer caches.

3.2 Alternative Coherence Protocols

MTR can support a variety of cache coherence protocols, such as MESI, MOESI, update protocols, and systems with imprecise directory representations. Here, we illustrate how MTR supports the MESI protocol, which enhances the MSI protocol with an additional exclusive (E) state. A read request for an unshared memory block is cached in the exclusive state as an optimization for read-modify-writes, while a read request for a shared memory block is cached in the shared state as in the MSI protocol. Under MESI, MTR reconstruction must determine whether a clean line is in the exclusive or shared state. A memory block held by multiple caches must be in the shared state. However, if there is only one cache with a valid and clean copy of a particular memory block, it can be in either exclusive or shared state. The ambiguity is similar to that depicted in Figure 8(b). If the directory is notified of P1's dirty writeback, it would grant P2 an exclusive copy of the data, but if P1's copy is not evicted until after P2's read request, P2 would have received a shared copy instead. This ambiguity is relatively infrequent since it only occurs when there is exactly one read sharer, and can usually be resolved using the same technique described earlier for determining eviction times under the MSI protocol.

In addition, MTR works with snoopy protocols, which are simpler to support than their directory-based counterparts since they only require cache reconstruction.

3.3 MTR and Checkpointing

Checkpointing removes the overhead of repeated functional fast-forwarding when evaluating multiple microarchitectural configurations, although it is more difficult to implement and requires large disk files to hold the memory contents at each checkpoint. MTR snapshots can be added to architectural checkpoints to reduce the amount of detailed warming required. Because MTR is

microarchitecture-independent, many different configurations can be initialized from the single snapshot.

The MTR representation is highly compressible. In the temporal dimension, long timestamps can be compressed using small deltas from a local base timestamp. In the spatial dimension, the multilevel valid-bit scheme discussed in Section 2.4 eliminates the need to store addresses along with data: the address can be reconstructed based on offset within the snapshot and the empty page bit vector.

3.4 Parallelization

MTR is well suited to parallel-hosted simulation in which each simulated CPU runs in its own thread. A slightly modified MTR in which each processor has its own write timestamp field would permit fast simulation without synchronization for every memory operation. The only concurrency constraint is that application-level atomic instructions must be implemented with atomic accesses to the simulated shared memory to ensure legal execution orderings. During fast-forwarding, a processor would make fast updates as before. The last writer would be determined at reconstruction time by comparing all CPUs' write timestamps. During parallel reconstruction, the shared MTR is read-only and the writeable CSR is divided into per-CPU sections that can be written without the need for locks. A barrier is needed between the coalesce and fixup stages.

4 Evaluation

To evaluate the MTR, we implemented a flexible and detailed cache-coherent distributed shared memory system model that includes primary caches, main memory with variable latency, and interconnection networks. We drive the memory system with Bochs, a popular x86-based full-system SMP-capable emulator [13]. The full-system nature of Bochs (i.e. it boots 4-way SMP Linux 2.4.24) allows us to test MTR with realistic workloads that require OS support. Furthermore, the execution-driven nature of the simulator allows our detailed memory system to affect the interleaving of threads, something difficult to achieve in trace-driven simulation.

The overall simulator structure is shown in Figure 10. The main loop of the simulator moves round-robin between the CPU models and the devices, incrementing a cycle count at the end of each loop. Those devices which need attention assert an interrupt line and are handled by the operating system on the simulated machine. During fast-forwarding mode, the Bochs CPUs access a shared "magic memory" for instant resolution of loads and stores. During detailed simulation, the memory backend performs detailed timing simulation such as cache miss/refill and network routing operations. MTR performs cache and directory reconstruction

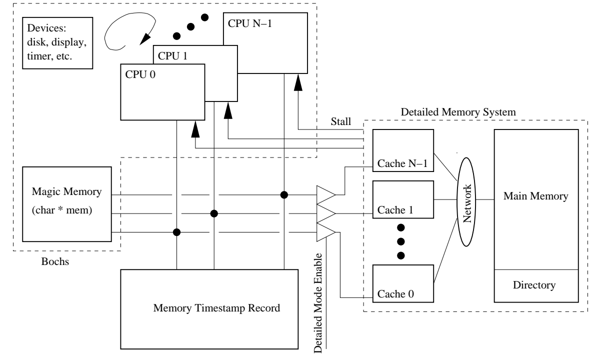

Figure 10. Our detailed memory model can stall a processor's execution based on timing models

for fast to detailed transitions. During detailed to fast transitions, the processors halt execution until all of the outstanding memory requests are serviced. To reduce the chance of coinciding with a periodic behavior in the benchmark, we randomly sample the detailed portions rather than rely on periodic sampling across the duration of the benchmark.

We used a simple timing model for this paper, to allow a greater number of runs to be completed. We believe the conclusions regarding accuracy of our memory system model are relatively unaffected by the exact timings chosen, although conclusions about a given real configuration would obviously require a more detailed system model. We use an in-order processor model that assumes each non-memory instruction takes one cycle to decode and execute. We combine this with a cache of L2-like size and organization, but L1-like timing to approximate the IPC of a modern out-of-order superscalar processor. A low DRAM latency was chosen to shorten simulation time while still causing significant timing-dependent instruction interleaving. Instructions that access the memory system are subject to the latencies of the model (shown in Table 1).

The multithreaded workloads used to evaluate MTR include Fortran/OpenMP NAS Parallel Benchmarks, server-style benchmarks which spend more time in the OS, and one dynamically scheduled AI benchmark written in Cilk[14]. Refer to Table 2 for a description of the benchmarks. For simulation automation, benchmarks were invoked in a run-level without superfluous processes/daemons to ensure that non-essential processes do not interfere with the benchmark. Each benchmark's inputs were chosen to allow detailed runs to complete within 12 hours on a 2.2GHz Pentium 4.

4.1 Baseline Functional Warming (FFW)

To provide a baseline, we added a fast functional warming (FFW) mode to our simulator, which is a straight-

Benchmark	Description	Instructions (Millions)	Mem Refs (Millions)	Ambiguous Addresses	Touched blocks / mem refs (%)
BT	NPB: block-tridiagonal CFD application, class S	780	400	95632	0.23
CG	NPB: conjugate gradient kernel, class S	792	390	111818	0.42
EP	NPB: embarassingly parallel kernel, class W	4306	2076	93786	1.40
FT	NPB: 3X 1D fast fourier transform, class S (-O0)	5264	2964	106356	0.44
LU	NPB: lower-upper decomposition with SSOR CFD application, class S	368	172	105995	0.24
MG	NPB: multigrid kernel, class W	2621	1549	292698	1.78
SP	NPB: scalar pentagonal CFD application, class S	361	164	101855	0.23
dbench	executes Samba-like syscalls, 3 clients, 10000 requests (gcc 2.96)	2692	867	180328	1.07
apache	apache benchmark 'ab' worker threading model, 2000 requests, 3 at a time (gcc 2.96)	2782	947	143332	0.27
ck	Cilk checkers (parallel alpha-beta search) 4 processors, black plies 6, white plies 5 (Cilk 5.3.2, gcc 2.96)	2617	396	102389	0.12

Table 2. Benchmark Description. All benchmarks are compiled with ifort-v8 -g -O2 -openmp unless noted. Instructions and Memory Reference columns reflect full detailed runs of the benchmark. The table also includes ambiguity and touched block stats for 1:100 MTR runs.

Number of Processors	4
Cache hit latency	1
Cache organization	4-way, 256KB
DRAM access latency	20
Cache miss buffer length	16
Network latency	1-cycle to neighbor
Network Topology	2D-mesh

Table 1. Simulation Parameters

forward SMP extension of earlier uniprocessor functional warming work [27]. FFW fast-forwards the simulation by updating the cache and directory state in each simulated cycle, but these FFW updates are significantly more expensive than MTR updates. For each memory request during fast-forwarding, FFW must first calculate a set index, then search all ways of the local cache to perform a tag check. If the request results in a hit, FFW must update the local LRU information; otherwise, multiple non-local caches and the directory must be updated to reflect all of the downgrades or invalidations caused by the cache refill. Whereas each MTR update runs in constant time, FFW updates scale with the number of caches and ways, and can vary due to miss rates and sharing patterns. FFW is also more difficult to parallelize than MTR, requiring some form of mutual exclusion to implement parallel cache and directory state updates correctly. On the other hand, FFW does not require any form of reconstruction during the transition between fast to detailed mode, because the directory and cache state is kept up-to-date during fast-forwarding.

4.2 Full System Simulation Variations

Fast-forwarded simulation yields vastly different thread interleavings than detailed simulation, since processors do not stall on memory instructions during fast-forwarding. It is well known that even small changes in SMP system timing often introduce large variations in simulation results [2]. It is therefore meaningless to compare a single run of detailed simulation and a single run of fast-forwarded simulation, because the different results may simply reflect the variation introduced by different, but still representative, thread interleavings rather than differences in simulation accuracy. We present the fast-forwarding results in the context of reasonable timing variations induced by altering system parameters.

We introduce two sources of variability in our system. First, we enable Bochs's slowdown timer component, which keeps the emulator in sync with real time on the host, and causes emulated devices to be handled at nondeterministic rates, varying the OS scheduling of threads. Second, we model changing processor workloads by choosing a different processor to run 25% slower than its peers every 10,000 instructions. Repeated simulation runs with these timing variations capture various representative thread interleavings and coherence race conditions. We use 100,000-cycle measurement samples, which should be long enough to span the duration of practical coherence races, so samples (of which we take hundreds per run) should observe races in proportion to their occurrence in full runs.

4.3 Accuracy Comparison

This section compares the cache miss rate and coherence message count (under the MSI coherence protocol) reported by detailed simulation, FFW, and MTR. Figure 11 shows these metrics for only one cache due to space limitations, but the results are similar for the other caches. We use *FFW(a:b)* and *MTR(a:b)* to denote the results of FFW and MTR where a:b is the ratio of instructions ex-

Figure 11. Accuracy comparison of FFW and MTR to detailed simulation. Each bar represents the median of eight individual runs of the benchmark. Max and min are noted with thin lines.

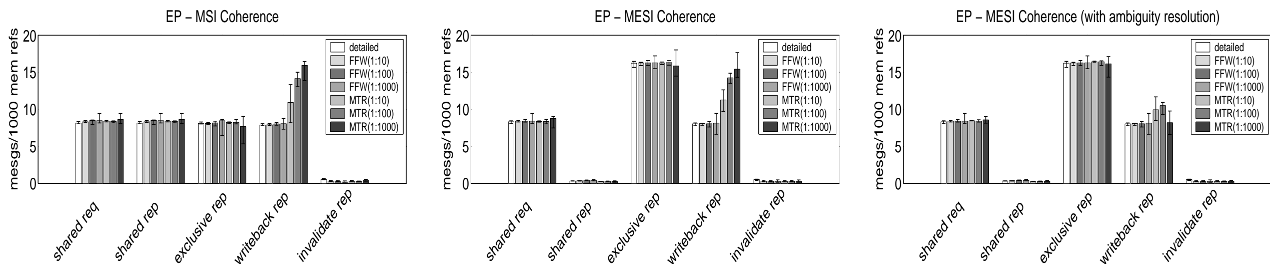

Figure 12. Comparison of MSI and MESI protocols. The rightmost figure shows the reduction in writeback messages caused by resolving M/S ambiguities as described in Section 2.5

ecuted in the detailed period to those executed in the fast period. The length of the detailed phase is fixed at 100,000 instructions. There are seven bars for each reported metric, each representing a simulation configuration: fully detailed run, FFW(1:10), FFW(1:100), FFW(1:1000), MTR(1:10), MTR(1:100), and MTR(1:1000). After trial runs with a wide range of parameters, we have found this set of detailed-to-fast instruction ratios to work well. It should be noted that the parameter selection is suited only for our implementation of the detailed model and benchmark set.

To show how the system variations affect simulation measurements, we report each result as median of eight separate runs (denoted by the solid bars), and the range of values observed (as denoted by the thin lines). We do not observe, and there is no reason to expect, a normal distribution of miss rates or message counts across runs. We also note again that the detailed runs can only capture *some* of the legal thread interleavings, which may differ significantly from those captured by their corresponding fast-forwarding runs. Nevertheless, in an attempt to quantify error incurred by our techniques, we note the largest excursion from the detailed run's median, normalized to the detailed median. Most accelerated benchmarks, with a detailed to fast ratio of 1:10, have an error in miss rate of less than 15%. LU, SP, and Apache exceed this error. For LU and SP, our shortest benchmarks, the problem is likely due to an inadequate number of detailed samples. Both MTR and FFW exhibit similar deviations from the detailed simulations – an effect most prominent in Apache. The technique we use to introduce variability by selectively adding processor stalls has a much larger relative impact on the fast-forwarding schemes than on the fully detailed run, where processor CPI is already much higher due to memory system stalls. We believe this explains the bias and generally greater variance observed with the fast-forwarding schemes versus the detailed model.

Increasing the ratio to 1:1000 leads to an unacceptable fast forwarding error in most cases. We achieve a good balance of error and speed when we set the detailed to fast ratio to 1:100. At this rate, six of the ten benchmarks have error within 25%, with the best performers being EP, MG, and CG with error below 12%. These benchmarks have relatively fewer invalidation and downgrade requests, indicating simpler sharing behaviors that are less likely to be perturbed by the effects of fast-forwarding.

Despite using different fast-forwarding strategies, both FFW and MTR report similar results for all metrics. The slight discrepancies can usually be attributed to the MTR's assumption that all ambiguous blocks should be marked "modified." Table 2 lists the number of ambiguously reconstructed addresses. The number of ambiguities does not always correlate directly with the accuracy of MTR fast-forwarding. First, although the number of ambiguities may be high, the resulting error may be small if we always pick the correct way to reconstruct. Second, the ambiguously reconstructed blocks may not be needed again, which is likely in applications with low temporal locality. Third, the error caused by incorrect reconstruction may be negligible compared to the simulation's variability, so it does not affect the overall accuracy of the fast forwarding simulation. To reduce error further, we can adopt the more sophisticated approach of establishing eviction times to help resolve ambiguities at the cost of slower reconstruction as described below.

4.4 Case Study with Fast-Forwarding

The ultimate goal of fast-forwarding is to allow designers to make quick and reliable architectural decisions even in the presence of large cross-run variations. This section presents a simple example to show results obtained from fast-forwarded simulations can provide as much insight as those from detailed runs. We have run the EP benchmark under both the MESI and the MSI protocols, and the collected coherence message metrics are shown in Figure 12. As expected (unless many read-modify-writes are present), shared requests under MESI and MSI are similar. However, the shared *replies* are quite different, since part of the shared reply messages in MSI become exclusive reply messages in the MESI protocol. The increase in shared replies is prominent in both fast-forwarding and detailed simulation data, thus allowing one to draw the same conclusion about the effects of the MESI versus MSI with much shorter simulations.

Despite indicating a much larger absolute number of writeback replies than detailed or FFW runs, MTR runs present similar writeback replies under both protocols, which allows one to draw the correct conclusion about the effect of MESI on writeback replies. This MTR discrepancy in writeback replies is due to our assumption that certain ambiguous blocks should be marked M rather than S. When we enhance MTR reconstruction to resolve this category of ambiguities, MTR results converge to their detailed and FFW counterparts, because fewer spurious writeback messages get generated by incorrectly marked M blocks (rightmost graph in Figure 12).

4.5 Speedup

Figure 13 compares the running times of our two fast-forwarding schemes. Each group of three bars represents the benchmark's execution time normalized to the slowest run for that benchmark. The three bars represent detailed-to-fast ratios of 1:10, 1:100, and 1:1000 respectively. When only 1% or less instructions are run in detailed mode, over 95% of the simulation time is spent in fast mode. The small fast-to-detailed transition times confirm that the re-

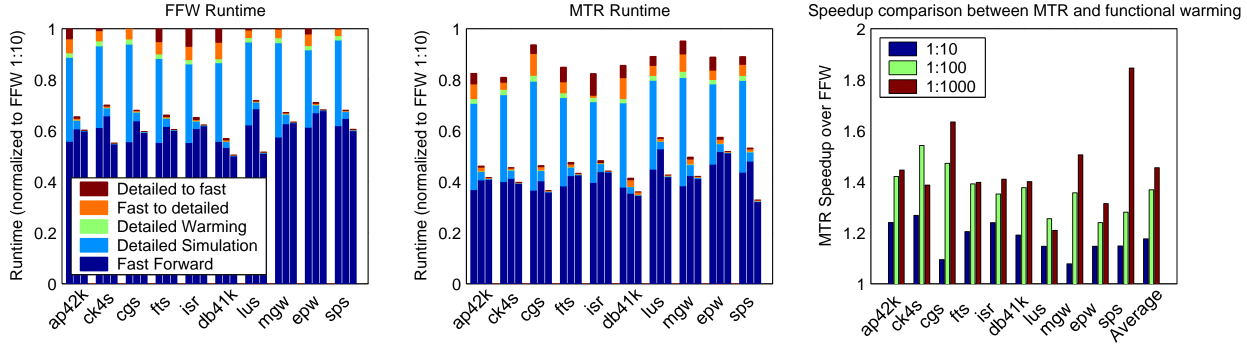

Figure 13. Normalized running time of FFW and MTR and the relative speedup of MTR over FFW. Each group of three bars represents the run time with detailed-to-fast ratios of 1:10, 1:100, and 1:1000. Each bar separately shows time spent in the three phases of simulation including transitions.

$$C = \frac{(T_{fast,FFW} + T_{slow,FFW}) - (T_{fast,MTR} + T_{slow,MTR})}{T_{slow,MTR}} + 1$$

Figure 14. In this equation, C represents number of configurations, $T_{speed,scheme}$ is an absolute time, and T_{slow} includes all reconstruction time.

construction time of the MTR scheme does not outweigh the speedup it can provide during fast mode. This is largely due to the fact that while our programs can make billions of memory references, the MTR compresses repeated references to the same block. Table 2 shows the total number of memory references during all of execution and the ratio of "touched blocks" to total references. The number of touched blocks which must be considered during reconstruction is a small fraction of the total requests – usually less than half a percent.

MTR edges out FFW by up to 1.45× speedup on average, as shown in Figure 13. While MTR is always faster than FFW, the relative improvement due to sampling ratio is effected by variation, number of touched blocks, and particular sharing scenarios. Although MTR achieves a respectable speedup in the serial execution of a single configuration, what is more exciting is the additional speedup from multi-configuration simulation and parallelization. Using the equation in Figure 14 and assuming 1% detailed execution time, we estimate that MTR could simulate five different configurations simultaneously and still have comparable running time to FFW running one configuration. The numerator represents the amount of time saved by using MTR instead of FFW. This time can be spent reconstructing and executing additional detailed simulations – each requiring $T_{slow,MTR}$.

For completeness, we note that the MTR is 7.7 times faster than our detailed model when using the relatively accurate 1:100 sampling ratio. FFW is 5.5 times faster at this ratio. Of course, much higher speedups will be achieved when using a more complex detailed model, for example, with a detailed DRAM model in place of our fixed-latency memory, or an out-of-order superscalar processor model instead of our in-order single-issue processor model.

5 Related Work

In *statistical sampling*, as used in our FFW and MTR schemes, the simulator selectively executes representative portions of the full benchmark. Much of the accelerated simulator literature falls under this category, only differing in which portions of the code are selected and how to fast-forward to those specific points in the program [12, 19, 27]. Several techniques have been proposed to warmup cache state prior to measurement in order to obtain more accurate simulation results [1, 4, 5, 10, 11, 27]. The MTR adds multiprocessor and directory support to these techniques.

Memory Reference Reuse Latency (MRRL) can be used to bound the amount of "detailed warming" prior to a sample to achieve a desired accuracy [8]. In a uniprocessor model, MRRL can remove 90% of the warmup costs, but as structures are larger and duplicated in a multiprocessor, the remaining warmup time can still be significant. Also, the reference patterns in a full-system multiprocessor simulation change non-deterministically with changes in the microarchitecture, making it more difficult to apply MRRL.

In addition to bounding the amount of detailed simulation needed to achieve small error rates with the desired confidence, the SMARTS framework [27] recently proposed functional warming, which simulates large structures (such as caches and branch predictors) in detail during fast-forwarding mode. By introducing a slight over-

head during fast-forwarding, functional warming shortens the detailed warming phase while providing accurate measurements. TurboSMARTS amortizes the lengthy warming phase by operating from stored snapshots which support various uniprocessor microarchitectures [25].

In SMT simulation, instruction throughput during a detailed sample interval can be used to guide fast-forwarding to the next interval, or even to perform a purely analytical simulation[24]. Such a technique can be augmented with an MTR to accelerate its fast-forwarding periods. Our simulator has the benefit of having an operating system decide realistically which threads may be executing concurrently. This allows us to ignore combinations which do not occur in practice, which might represent a substantial fraction of all possible phase combinations. SMARTS has recently been extended [7] to support estimation of program runtime on multiprocessors by taking detailed samples from only those processors comprising the "critical path." When parallel programs have no clear critical path or when runtime is an insufficient metric, the MTR should allow more comprehensive studies.

When the host and target are of similar architectures, *direct execution* is a natural choice [6, 16, 26] for fast simulation. However, in a multiprocessor simulation the memory operations, not the instruction emulation, constitute the largest obstacle to high speed simulation of a parallel processor [15]. Thus, direct execution can be complemented with MTR updates during fast-forwarding to further speed up simulation.

Parallelizing the simulator is another logical speedup approach [3, 16, 20, 28]. Section 3 outlined how the MTR is well-suited for parallel hosts. An extension to stack analysis can be used to support single-pass simulation of multiple configurations for an SMP [23]. While this approach achieves some of MTR's goals, it would be difficult to invoke in a parallel simulation as each cache must analyze each memory access as it is issued, and it does not address directory-based systems.

Finally, *statistical/synthetic* approaches can dramatically reduce runtimes[17, 18]. With respect to multiprocessors, a 10–15% error in instruction throughput was observed depending on workload [17], though trends observed with synthetic workload follow those seen in a detailed baseline model.

6 Conclusion

We have introduced the Memory Timestamp Record (MTR) and algorithms for its use. Despite its extensive description, the MTR is a simple concept which exploits fundamental properties of caches and cache coherence. When combined with statistical sampling, the MTR enables faster execution of multiprocessor simulations: up to 1.45× faster than a multiprocessor functional warming model (FFW) and 7.7× faster than our detailed baseline. In addition, the MTR snapshot representation is not tied to specific microarchitectural details. We have shown how it can be used to reconstruct multiple cache configurations and coherence protocols. An MTR-enabled simulator will allow computer architects to rapidly evaluate a wide range of complex multiprocessor architectures.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Trans. on Computer Systems*, 6(4), Nov 1988.

[2] A. R. Alameldeen and D. A. Wood. Addressing workload variability in architectural simulations. In *HPCA-9*, Feb. 2003.

[3] L. Ceze et al. Full circle: Simulating Linux clusters on Linux clusters. In *Fourth LCI Int'l Conference on Linux Clusters: The HPC Revolution*, June 2003.

[4] T. M. Conte, M. A. Hirsch, and W. W. Hwu. Combining trace sampling with single pass methods for efficient cache simulation. *IEEE Trans. on Computers*, C-47(6), June 1998.

[5] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Int'l Conference on Computer Design*, Oct 1996.

[6] M. Durbhakula, V. S. Pai, and S. V. Adve. Improving the accuracy vs. speed tradeoff for simulating shared-memory multiprocessors with ILP processors. In *HPCA-5*, Jan. 1999.

[7] N. Hardavellas at al. Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):31–35, March 2004.

[8] J. Haskins, Jr. and K. Skadron. Memory reference reuse latency: Accelerated sampled microarchitecture simulation. In *ISPASS*, pages 195–203, Mar. 2003.

[9] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Trans. on Computers*, 38(12):1612–1630, 1989.

[10] R. E. Kessler, M. D. Hill, and D. A. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Trans. on Computers*, 1994.

[11] S. Laha, J. A. Patel, and R. K. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Trans. on Computers*, Feb 1988.

[12] G. Lauterbach. Accelerating architectural simulation by parallel execution of trace samples. Technical Report TR-93-22, Sun Microsystems Laboratories, Dec. 1993.

[13] K. Lawton et al. Bochs. `http://bochs.sourceforge.net`, July 2004.

[14] C. Leiserson et al. Cilk 5.3.2. `http://supertech.lcs.mit.edu/cilk`, June 2000.

[15] P. S. Magnusson et al. Simics: A full system simulation platform. *IEEE Computer*, Feb. 2002.

[16] S. S. Mukherjee et al. Wisconsin Wind Tunnel II: A fast, portable parallel architecture simulator. *IEEE Concurrency*, 8(4):12–20, October-December 2000.

[17] S. Nussbaum and J. E. Smith. Statistical simulation of symmetric multiprocessor systems. In *35th Annual Simulation Symposium*, Apr. 2002.

[18] M. Oskin, F. T. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor designs. In *ISCA-27*, Jun 2000.

[19] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *PACT*, Sep 2003.

[20] A. Pimentel and L. Hertzberger. Distributed simulation of multicomputer architectures with Mermaid. In *Symposium on Performance Evaluation of Computer and Telecommunication Systems*, July 1998.

[21] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-10*, pages 45–57, 2002.

[22] R. A. Sugummar. *Multi-Configuration Simulation Algorithms for the Evaluation of Computer Architecture Designs*. PhD thesis, University of Michigan, Aug. 1993. Technical Report CSE-TR-173-93.

[23] J. G. Thompson. *Efficient analysis of caching systems*. PhD thesis, University of California at Berkeley, 1987.

[24] M. Van Biesbrouck, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *ISPASS*, Mar. 2004.

[25] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. TurboSMARTS: Accurate microarchitecture simulation sampling in minutes. Technical Report 2004-3, Computer Architecture Lab at Carnegie Mellon (CALCM), 2004.

[26] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of ACM SIGMETRICS '96: Conference on Measurement and Modeling of Computer Systems*, 1996.

[27] R. Wunderlich et al. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA-30*, June 2003.

[28] G. Zheng, G. Kakulapati, and L. V. Kalé. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *IPDPS-18*, Santa Fe, NM, April 2004.