Accelerating Multiprocessor Simulation

Kenneth C. Barr
MIT Computer Science and Artificial Intelligence Laboratory
200 Technology Square
Cambridge, MA 02139
kbarr@csail.mit.edu

Abstract

Detailed simulation of interesting benchmarks can take days when running on a multi-processor simulation target. We introduce the Memory Address Record (MAR), a structure which allows for rapid simulation of directory-based cache coherent processors. The MAR allows the directory state and caches to be constructed quickly and correctly after a period of fast warming. Use of the MAR boosts speeds by up to 1.9 times on a set of scientific computing benchmarks.

1. Introduction

Computer architects rely on simulators to evaluate, refine, and validate new designs before they are implemented. The speed of these simulations dictates the number of design points that can be studied, the number of experiments that can be run, or the amount of validation that occurs, so it is crucial that a simulation complete quickly. Unfortunately, complex systems dictate complex simulators, and we have reached the point where detailed simulation of interesting benchmarks can take days (or longer), even for a uniprocessor target.[1] When simulating a parallel target, execution time can be even greater as the simulation must take into account cache coherence, interconnect timing models, and many processors executing instructions. If not kept in check, this necessary complexity can slow down the simulation feedback cycle to the point of making simulation useless as a debugging or design tool.

This paper proposes the Memory Address Record (MAR), a structure for speeding up the simulation of directory-based cache-coherent processors. During a fast "warming" phase (the bulk of simulation), the simulator replaces detailed cache, interconnect, and directory models with quick writes to the MAR. The data from the MAR can be used to reconstruct processor data caches and directory state. After reconstruction, a shorter period of detailed simulation can begin with correct data in the caches and directory, leading to relevant simulation results even as simulation time is decreased.

We begin with a review of related work (Section 2), including a recent result related to statistical sampling in microarchitectural simulation. Section 3 presents the MAR and explains how it is used. Section 4 gives the intuition behind its speed and correctness. In Section 5 we discuss the properties of our chosen scientific benchmarks and show that the MAR cuts the execution time of these benchmarks. Performance is measured using the MAR we have implemented as an addition to a cycle-accurate, execution-driven simulation of the SGI Origin 3000. Sections 6 and 7 conclude the paper with some caveats and suggestions to address them in future work. Finally, we summarize the current state of the MAR.

2. Related Work

There are countless ways to attack slow simulators ranging from coding optimizations (e.g., the macro-based instruction execution of SimpleScalar [1]) to parallel hosting of simulation in which each simulated processor runs on a different physical processor. Rao and Wilsey provide a good survey of research used to speed simulation of large systems with components that operate in parallel. Their paper also discusses problems specific to such large-scale simulation including the slowdown imposed by synchronization of the blocks that operate in parallel, or imposed by replay when this synchronization is relaxed [8]. The Wisconsin Wind Tunnel project is another source of ideas for parallel simulation [6]. One of the project's advances is having the host computer execute target instructions directly rather than through emulation. Windtunnel also takes advantage of a parallel host to simulate a parallel machine, using portable message passing and synchronization directives rather than machine-specific ones. Speed boost typically comes at the expense of accuracy and/or complexity so techniques which can minimize this expense are preferred. The RSIM group examines such tradeoffs in the context of shared-memory multiprocessors [3] and [7].

The technique presented in this paper is motivated by two interesting techniques which strive for speed with minimal impact on accuracy and a not un-reasonable amount of complexity. In the first, "Rigorous Statistical Sampling," short periods of slow, detailed simulation are alternated with long periods of fast, reasonably accurate simulation. If carefully performed, the technique yields accurate results (less than 1% error rates versus complete simulation) with dramatic speedup (35-60 times faster) [11]. Multi-configuration simulation is a second underlying concept. It allows a single simulation run to report the effects of many different processor configurations with a single run. By setting up proper data structures and noting structural properties, statistics for many configurations can be produced quickly without specifically simulating each configuration. For example, the properties of fully associative (FA) caches imply that a FA cache is a subset of all larger FA caches (with the same line size) [9]. Thus, large FA caches need not be probed if one observes a hit in a smaller FA cache.

[1] In this paper, the simulation *host* is a single-processor workstation running the simulation. We are concerned with *targets* that simulate multi-processor machines.

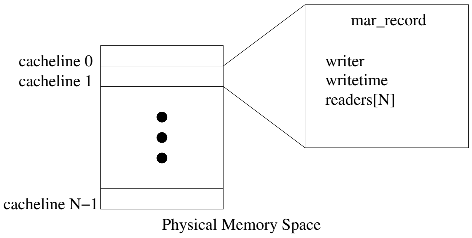

Figure 5: An abstract view of the MAR during fast warming. For each touched cacheline in physical memory, the MAR contains a "mar_record."

3. Structure of the MAR

The MAR is intended to support Rigorous Statistical Sampling as described in [11]. As such, it supports very fast updates during the warming phase. When it is time to switch to the detailed simulation phase, these updates can be used to quickly fill the caches and directory with correct contents. Since the MAR contains a superset of data in the caches, it supports multi-configuration simulation (e.g., different cache sizes, replacement policies, or coherence protocols). This section explains the structure of the Memory Address Record and gives an example of its use. We also sketch a $O(\text{sets} * \text{ways} * \text{CPUS})$ algorithm which can be used to replace the current $O(\text{sets} * (\text{ways} * \text{CPUS})^2)$ procedure *and* allow one to distinguish between all four directory states in a MESI protocol.

3.1 Data structure and Algorithms

The MAR is updated whenever a processor issues a read or write request into its memory hierarchy. When the simulator leaves the fast warming phase, data from the MAR is used to fill in the caches of all processors and the directory information for each cache line. For simplicity, the implementation described in this paper uses a single-level cache, and each processor is assumed to have the same cache parameters.

The MAR makes heavy use of C++ standard template library containers and iterators. The MAR is implemented with a hash_map in order to support $O(1)$ updates (on average) and limit the space use to $O(\text{touched lines})$. A "touched" cache line is a single cache line that that may be accessed many times over the course of execution. Time-ordered data is kept in priority queues or (multi)sets. An STL set is an associative container that (contrary to its mathematical counterpart) stores its contents in a well-defined order. This allows the quick filtering-out of data at the beginning of the set. Sets, unlike priority queues, support iterators, specifically the reverse iterator used to retrieve the most recent accesses that belong in the cache. Operations on a set require $O(\log n)$ worst-case time.

Figure 5 shows the conceptual structure of the Memory Address record. In the implementation, we do not store a `mar_record` for every cache line in physical memory. Rather, a hash table (keyed on address) is used to store only the `mar_record`s for those lines that are accessed by the program. A `mar_record` stores the ID of the last processor to write that line (if any) and the time at which the write occurred. An array of readers, indexed by processor ID, stores the last read time for each processor. This hash-based structure provides for fast updates during the warming phase using the algorithm in Figure 1. Data may be written to the MAR by the simulated processors in any order as the timestamps allow for sorting later. When it is time to enter the detailed phase, we can use the data in the MAR to reconstruct the caches and directory.

Cache reconstruction begins by coalescing the data for each cache line that maps to a given set. We transfer the data from the MAR into the system of priority queues displayed in Figure 6 by following the COALESCE-CACHELINES algorithm of Figure 2. The process continues with the RECONSTRUCT-CACHES algorithm shown in Figure 3. One may notice that the $latest\_write[i][j]$ queues contain information that is already present in the $pqueue$: the tag and time of the latest write to set i by CPU j. However, by storing this information outside of the $pqueue$, we can quickly filter the $pqueue$ so that only accesses following the latest writes are considered. An improved set of algorithms that could do without this redundant information is described in section 3.3. Once RECONSTRUCT-CACHES is complete, each processor's cache is nearly up-to-date and will be finalized during the RECONSTRUCT-DIRECTORY phase.

RECONSTRUCT-DIRECTORY, shown in Figure 4, uses the contents of the caches and the rules in Table 1 to reconstruct the directory state. Current, a three-state MSI protocol is used. Since a line may be evicted from cache, we are unable to distinguish between a line in the Exclusive state and one in the Shared state (where all other copies have been evicted). Cache state is also updated during this phase to reflect write-backs (dirty $\rightarrow$ clean transitions) caused by share requests.

3.2 Example

To see the MAR in action, we present an example. Consider a two-processor system that makes the stream of memory references shown in Table 2. For simplicity, assume each read (R) and write (W) is to the same cache line with address A.

At time 6, we wish to reconstruct the caches and directory state. The data in table 2 is stored in the MAR, but as a single stream of triples: A: (0,CPU1,R), (1,CPU1,W), (2,CPU2,R), ... COALESCE-CACHELINES places the data into two priority queues and notes that latest write to the set containing A is at time 3. Next, we execute RECONSTRUCT-CACHES. All accesses prior to time 3 are thrown away, and the data remaining in the $pqueue$s tells us that there is a "read" copy of the data in CPU1 and a "write" copy in CPU2. When we run RECONSTRUCT-DIRECTORIES, two copies are noted, so both must be marked clean, and the line must be Shared by CPU1 and CPU2.

3.3 An improved algorithm

The MAR implementation described above will likely serve as a straw man for future implementations. In fact, we are developing a new scheme that is both faster in theo-

Condition	Resulting State
Line not present	I
Line present in one place (dirty)	M
Line present in one place (clean)	S or E
Line present in many places (all clean)	S

Table 1: Rules for reconstructing directory state. All lines start in the invalid state. Only *one* of the above conditions should apply.

```
UPDATE(time, address, load, cpu)
if isLoad(load)
   then MAR[address].cpu[cpu] ← time
   else
        MAR[address].writer ← cpu
        MAR[address].writetime ← time
```

Figure 1: This procedure is called during the warming phase to update the MAR.

```
COALESCE-CACHELINES()
for each mar_record, i, in MAR
     do set ← setContaining(addressOf(i))
        if isWrite(i)
          then p ← writeCPU(i)     determine last writer of i
               record i's tag and time in priority queue last_write[set][p]
               record WRITER and i's tag and time in pqueue[set][p]
          else     it's a read
               for each CPU p that reads i
                   do record READER and i's tag and time in pqueue[set][p]
```

Figure 2: Procedure to coalesce cache lines into an array of per-set priority queues

```
RECONSTRUCT-CACHES(numsets, numways, numcpus)
for set ← 0 to numsets
     do for p ← 0 to numcpus
            do times[0 . . numways] ← the times of up to numways most recent writes in last_write[set][p]
               Discard from pqueue[set][p] any access before times
               for w ← 0 to numways
                   do place the tag, dirty, and valid bits from the numways most recent
                      pqueue[set][p] entries into way w of set set in the cache of CPU p
```

Figure 3: Procedure to rebuild caches using data from the MAR.

```
RECONSTRUCT-DIRECTORY(numsets, numways, numcpus)
for i ← 1 to numcpus
     do for j ← 1 to numsets
            do for k ← 1 to numways
                   do look in way k of set j in each of the other numcpus −1 caches, and apply the rules in Table 1
```

Figure 4: Procedure to rebuild directory from data in caches.

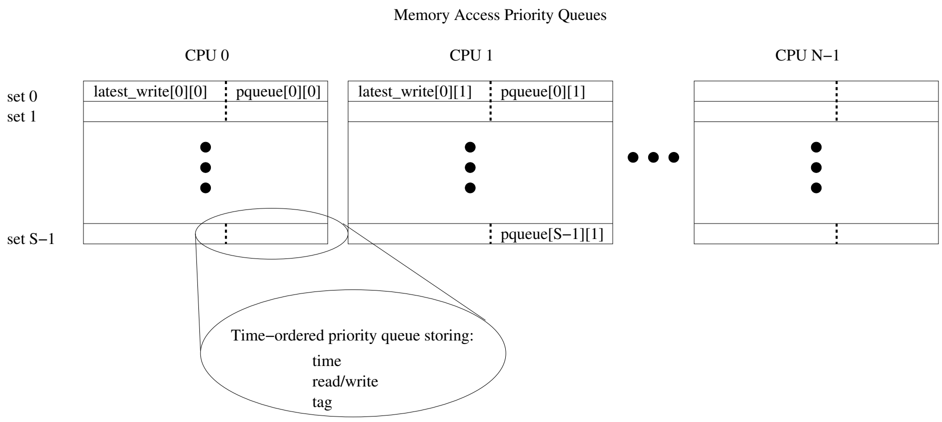

Figure 6: Priority queues used for constructing cache contents.

Time	0	1	2	3	4	5
CPU 1	R	W				R
CPU 2			R	W		

Table 2: Stream of accesses referred to in example.

retical runtime and more accurate in terms of allowing MESI instead of MSI. A brief description of this algorithm follows, whetting the appetite for future improvements.

UPDATE now counts how many times each cache line is accessed. COALESCE-CACHES no longer maintains *last_write*. However, it is modified to record the last reader or writer of each cache line. This additional information can be used to distinguish the S and E state and eliminate the squared term from the runtime of RECONSTRUCT-DIRECTORIES. Two passes are made across each set. In the first, we build sharing vectors, in the second we consult the new counts and last write tables to get directory state.

The new RECONSTRUCT-DIRECTORIES works as follows... We scan through a set and consider each line l. If the last access was a read, and l is a write, then it is known that l is shared without having to check all other CPUs and ways. If the last access was a write and l is write, then we can stop checking: l must be Modified. The counters allow us to distinguish the Shared and Exclusive state. Consider a line that has been read > 1 time (as recorded by UPDATE). If there is only one instance of it in the cache (and assuming the computer does not do auto-upgrading of lines), then the line must be Shared. However, if we have recorded only one read, then the line is in the Exclusive state.

4. Why the MAR works

There are two steps to understanding the MAR. First, one must see that it faithfully restores the state of both the caches and the directory. Next, one must understand how this restoration can occur in the absence of detailed cache and directory models.

4.1 Correctness

A cache contains the most-recently used subset of the main memory of the computer. We can use induction to show that the caches created by the MAR and its algorithms provides the same subset that would arise during traditional simulation. We start with a base case and show that the algorithms preserve necessary and sufficient information for reconstruction and that they apply this information appropriately.

For the base case, assume the contents of the caches are valid. As memory is accessed, it is recorded in the MAR. Only one processor may write a cache line at a time. The MAR stores that single most recent writer for each line. Many processors can read a cache line at once, and the MAR also keeps a record of all CPUs that tried to read a line. Thus no information needed for cache-construction is lost by UPDATE in the warming phase.

Since a cache is a subset of main memory, several lines of memory may map to the same cache set. The COALESCE-CACHELINES procedure performs the lines-to-sets mapping and sorts the accesses, but does not discard any information.

In an w-way set associative cache, only the w most recently accessed lines mapping to a set are present in the cache. The RECONSTRUCT-CACHES procedure consults the per-cpu/per-set queues created by COALESCE-CACHELINES placing the w most recent accesses into the appropriate cache. By discarding accesses that occur before the latest write, this reconstruction procedure respects sequential consistency in that reads see the value of the most recent write to the same address. Thus, after RECONSTRUCT-CACHES has completed, the caches' contents are valid and accurately reflect the state of the machine after the most recent access.

Turning to the directory, note that the state of a line and its sharing vector merely reflect the information stored in the cache of each processor. For example, the only way many processors can have a valid copy of a cache line is if the line is Shared by each. In an actual system, the directory information is stored at a home node in order to coordinate memory accesses of multiple processors and preserve sequential consistency. Unlike a real system, the simulator is "omniscient" in that it has access to the cache of every simulated processor. By looking at the contents of all caches, the directory bits can be correctly inferred using the rules in Table 1.

4.2 Speed

The directory-based cache coherence protocol used by the SGI Origin is carefully described in [5]. When the number of messages-per-transaction is enumerated, one finds 2,3, or 4 messages in many cases, but $(2 + 2 \cdot \text{sharers})$ messages when a Read-Exclusive or Upgrade is issued to a line in the Shared state. While optimizations (intervention forwarding or reply forwarding) may be applied [2], optimizations attack messages along the critical path rather than the total number of messages exchanged. A faithful timing simulator must model each message and the contention it faces on an accurate model of the interconnect. Furthermore, cache timing models must be consulted and the caches and directory bits must be updated appropriately. The MAR eliminates these details from the warming phase, but allows accurate reconstruction of the caches and directory that simple fast-forwarding does not.

At a high level, the time of default simulator can be expressed by: (all memory accesses $*\ T$(simulate cache $+$ simulate directory $+$ simulate network))
where each "simulate" term is dependent on factors such as associativity, nodes, messages, et cetera.

The time required in a MAR-based simulator is broken into two parts: update and replay. Update time is (all memory accesses $*\ T$(hash table update)) where T(hash table update) $= O(1)$. Playback time is greater than update, and a glance at the algorithms used suggest that it is quicker than performing a detailed model of a single request. Even if the detailed model was fast, MAR playback has the advantage of only depending on the number of lines touched and cache memory size rather than each dynamic access. Section 5 (Figure 9) shows that touched lines is a much smaller number than all memory accesses. Referring to figures 2, 3, and 4 we see that playback time with the current algorithms is:
(touched lines $*\ O$(sharers) $+\ O$(sets $*$ (ways $*$ CPUS$)^2$)).

5. Evaluation

To evaluate the MAR, we use the same subset of the Splash 2 benchmarks [10] found in Hennessy and Patterson's computer architecture textbook [4]. The authors chose these particular four benchmarks because they represent common techniques in scientific computing (LU and FFT) and important types of parallel communication (Barnes and Ocean). Furthermore, the benchmarks represent a spectrum of performance characteristics and scaling properties (Table 3) which help to exercise the MAR in different situations. All applications were run with default parameters with the following exceptions: a four-processor configuration was specified and problem sizes scaled to those listed in Table 4. These scaling choices were made so that the benchmarks could complete in 1-5 minutes.

To demonstrate the benchmarks' contrasting behavior, Figure 7 shows the data cache miss rate on a 4-way machine with 64KB, 2-way set associative, 32B block caches. The rates vary widely between the benchmarks and are relatively consistent as processor count is varied. The benchmarks can also be compared with respect to their sharing behavior. Tests with an unmodified UVSIM reveal different sharing patterns shown in Figure 8. Low sharing can reduce the time spent modeling communication. Though 0 or 1 sharer-per-cacheline is common, both LU and Barnes exhibit non-negligible amount of higher sharing degrees. Figure 9 shows the high accessed-to-touched ratio common to all benchmarks. Recall that a "touched" cache line is a single cache line that that may be accessed many times over the course of execution. This high ratio means that many accesses will be "compressed" in the MAR during the update phase. When we arrive at the playback phase, work is done only for each line in the MAR.

	Parameter	Value
FFT	Points to transform	16384
LU	Matrix dimension	128 x 128
Barnes	Particles	512
Ocean	Grid points per dimension	34

Table 4: Splash 2 Parameters

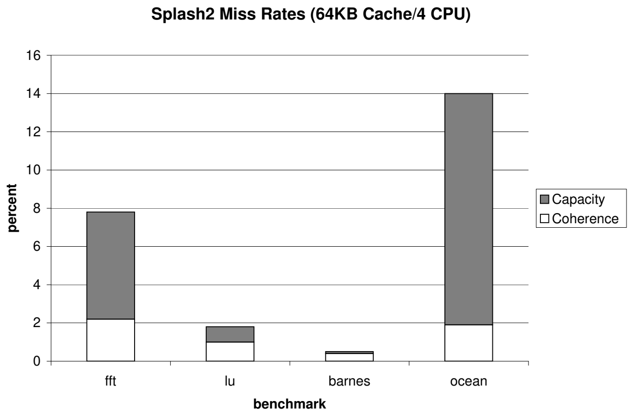

Figure 7: Varying cache misses in Splash2 benchmarks. From Hennessy and Patterson

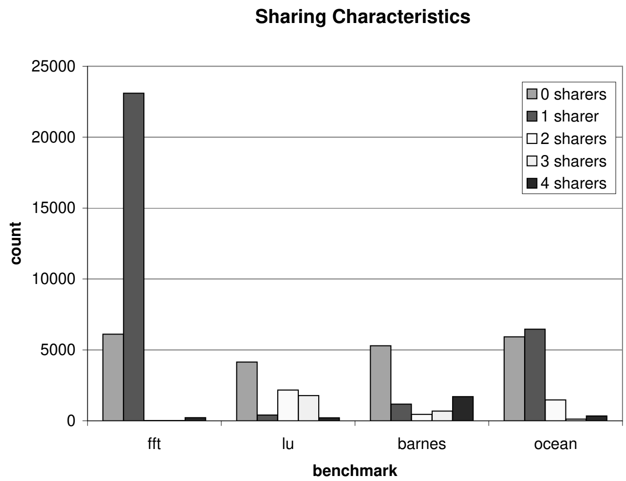

Figure 8: When run on four processors, most cache lines are shared between 0 or 1 CPU.

Figure 10 shows wall-clock execution time of the benchmarks running both with and without the MAR. All times were measured on a 2.20GHz Intel Xeon. Benchmarks were run to completion. For the MAR tests, the MAR is updated throughout execution and cache/directory state is reconstructed upon completion. The detailed DRAM, Hub, and Processor models are disabled, and gprof is used to ignore time spent in cache simulation. For detailed simulation, the detailed models are turned on, but the simulator is no longer slowed by the MAR.

	Scaling of Computation	Scaling of Communication	Scaling of computation-to-communication
FFT	$\frac{n \log n}{p}$	$\frac{n}{p}$	$\log n$
LU	$\frac{n}{p}$	$\frac{\sqrt{n}}{\sqrt{p}}$	$\frac{\sqrt{n}}{\sqrt{p}}$
Barnes	$\frac{n \log n}{p}$	$\approx \frac{\sqrt{n} \log n}{\sqrt{p}}$	$\approx \frac{\sqrt{n}}{\sqrt{p}}$
Ocean	$\frac{n}{p}$	$\frac{\sqrt{n}}{\sqrt{p}}$	$\frac{\sqrt{n}}{\sqrt{p}}$

Table 3: Communication and computation scaling of Splash 2 benchmarks

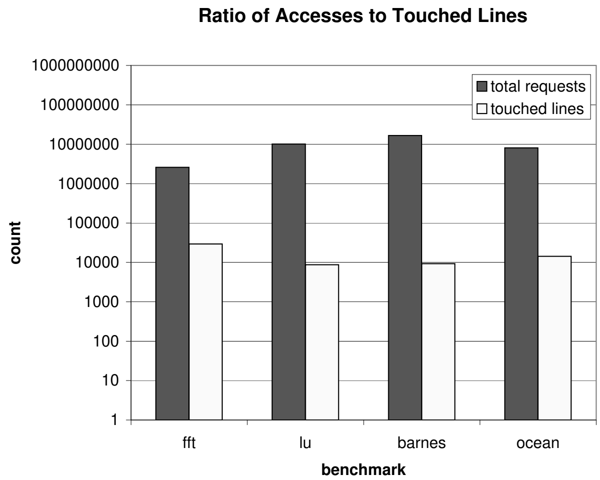

Figure 9: All benchmarks have either 1:100 or 1:1000 touched lines-to-accesses ratios

To help understand the time overhead of the MAR versus detailed modeling, consider that the current implementation of the MAR adds about 28.8% of overhead to a detailed simulation of LU, but since the MAR allows us to skip detailed memory and processor simulation during warmup, the net result is a 20% speedup. This speedup can be expected to improve when the $O(\text{sets} * (\text{ways} * \text{CPUS})^2)$ RECONSTRUCT-DIRECTORY algorithm is replaced with the $O(\text{sets} * \text{ways} * \text{CPUS})$ version described in Section 3.3. In addition, since the structure of the MAR allows each set to be operated on independently, the reconstruction algorithm can be parallelized when running on compatible hardware.

6. Future Work

Implementing the MAR and describing it to others has prompted several new research directions. It appears that one of the key benefits of the concept is the ease with which it can be parallelized. As the implemention is refined, perhaps with the scheme from 3.3, the independence of sets during reconstruction should be a primary goal. Another area of refinement is space-efficiency. The current MAR makes liberal use of memory, but larger workloads on more simulated processors could require more judicious data structures to avoid paging them to disk.

We would like to investigate whether the MAR and associated structures could support an incremental approach to update. That is, once we have reconstructed contents and have meta-data, perhaps this data can be retained to help speed subsequent replays.

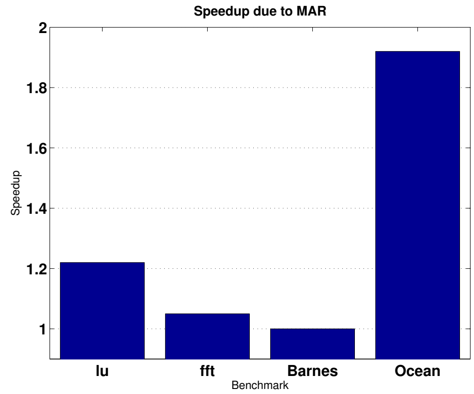

Figure 10: Speedup due to MAR. Normalized to execution time of detailed simulation without MAR

Though the description of the MAR explains how it correctly implements cache coherence, the MAR has not been tested *"in vivo."* Validating the default simulation state with that generated by the MAR is crucial before the MAR can be used in long simulations.

There is concern that certain communication primitives in the simulated application, such as `join`, could hurt the performance of the MAR. In the case of spin-lock during warming, each processor streams requests into the MAR whereas a detailed, system-level simulator could account for the scheduler allowing one processor to make progress on the critical section before letting a different processor try to acquire the lock. The MAR-based simulation result would be sequentially consistent but not representative of a realistic execution. To correct this would require the simulator to be aware of the primitives being used by the simulated application. In a parallel-hosted implementation of the MAR, this awareness would be essential to correct operation.

While this section has introduced issues with which the developer must be aware, each appears tractable and we are encouraged about the future of a MAR-like technique.

7. Conclusion

We have introduced the Memory Address Record (MAR) and algorithms for its use. The MAR is shown to allow quick and correct restoration of the caches and directory of a multiprocessor simulator. In no case does the MAR slow down the simulation. In fact, we see up to 1.9X speedup in the case of the Ocean scientific benchmark. Speeds are likely to improve as the algorithms are refined and parallelized. When combined with Rigorous Statistical Sampling, the MAR will enable faster execution of multiprocessor simulations.

8. Acknowledgments

This paper stems from conversations with my advisor, Krste Asanovic.

9. References

[1] Doug C. Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[2] David E. Culler and Jaswinder Pal Singh with Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1999.

[3] M. Durbhakula, V. S. Pai, and S. V. Adve. Improving the accuracy vs. speed tradeoff for simulating shared-memory multiprocessors with ILP processors. In *3rd International Symposium on High-Performance Computer Architecture*, January 1999.

[4] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, third edition, 2003.

[5] J. Laudon and D. Lenoski. The SGI origin: A ccNUMA highly scalable server. In *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

[6] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. D. Hill, J. R. Larus, and D. A. Wood. Wisconsin Wind Tunnel II: A fast, portable parallel architecture simulator. *IEEE Concurrency*, 8(4):12–20, October-December 2000.

[7] V. S. Pai, P. Ranganathan, H. Abdel-Shafi, , and S. V. Adve. The impact of exploiting instruction-level parallelism on shared-memory multiprocessors. *IEEE Transactions on Computers*, 48(2), February 1999.

[8] D. M. Rao and P. A. Wilsey. An ultra-large scale simulation framework. *Journal of Parallel and Distributed Computing*, 2002.

[9] Rabin A. Sugummar. *Multi-Configuration Simulation Algorithms for the Evaluation of Computer Architecute Desig ns*. PhD thesis, University of Michigan, August 1993. Technical Report CSE-TR-173-93.

[10] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[11] Roland Wunderlich, Thomas Wenisch, Babak Falsafi, and James Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.