Trace-driven Simulation of Memory System Scheduling in Multithread Application

Pengfei Zhu[1,2] Mingyu Chen[1] Yungang Bao[1] Licheng Chen[1,2] Yongbing Huang[1,2]

[1]State Key Laboratory of Computer Architecture
Institute of Computing Technology
Chinese Academy of Sciences
Beijing, China 100190

[2]Graduate School of Chinese Academy of Sciences
Beijing, China 100190

{zhupengfei, cmy, baoyg, chenlicheng, huangyongbing }@ict.ac.cn

ABSTRACT

Along with commercial chip-multiprocessors (CMPs) integrating more and more cores, memory systems are playing an increasingly important role in multithread applications. Currently, trace-driven simulation is widely adopted in memory system scheduling research, since it is faster than execution-driven simulation and does not require data computation. On the contrary, due to the same reason, its trace replay for concurrent thread execution lacks data information and contains only addresses, so misplacement occurs in simulations when the trace of one thread runs ahead or behind others. This kind of distortion can cause remarkable errors during research. As shown in our experiment, trace misplacement causes an error rate of up to 10.22% in the metrics, including weighted IPC speedup, harmonic mean of IPC, and CPI throughput. This paper presents a methodology to avoid trace misplacement in trace-driven simulation and to ensure the accuracy of memory scheduling simulation in multithread applications, thus revealing a reliable means to study inter-thread actions in memory systems.

Categories and Subject Descriptors

B.3.3 [Memory Structures]: Performance Analysis and Design Aids—simulation; C.4 [Computer Systems Organization]: Performance of Systems—measurement techniques; I.6 [Computing Methodologies]: Simulation and Modeling

General Terms

Experimentation, Measurement

Keywords

Trace-driven simulation, memory scheduling algorithm, trace misplacement, multithread application.

1. INTRODUCTION

As commercial chip-multiprocessors (CMPs) integrate more and more cores, memory system studies, including data hierarchy, memory scheduling, and DRAM architecture, are playing an increasingly important role in computer architecture. Memory system research, especially memory scheduling algorithms which handle requests to the major shared resource DRAM, is mainly based on simulation. Since the execution of benchmarks in a real system can hardly evaluate every design point in the memory system, and it is impossible to implement the salient points by prototyping the innovations at the early-research stage, the software simulator becomes a common tool to explore the memory system's design space.

However, simulating such a memory system through accurate execution-driven tools is difficult. The simulation for the memory scheduling, which aims to improve the memory system performance by prioritizing requests to the shared resources in DRAM (i.e., channel/rank/bank/row) [1, 8, 9, 10, 11], is increasingly challenging since the number of integrated cores and concurrent threads in CMP systems keep growing, as does data size.

Unlike execution-driven simulation, trace-driven simulation takes the program instructions and address traces as the input. Without the heavy burden of computation and data movements, the trace-driven simulator often runs faster and gives the researcher more flexibility to employ simulation experiments. Furthermore, in the memory scheduling research area, trace-driven simulation is very suitable for memory scheduling studies since memory access traces can be produced by workloads in platforms with different ISAs to simulate the memory system performance in a variety of machines, as well as in heterogeneous systems, examples of which are described in [2, 3].

However, when used to simulate memory scheduling algorithms for multi-threaded workloads, a trace-driven tool also has an inherent limitation. The trace-driven simulation cannot capture or model inter-thread actions in parallel multithread applications, including inter-thread order and synchronizations. Meanwhile, the goals of the memory scheduling algorithm are to resolve the memory contention or interference, and to enhance both the performance and the fairness of a shared DRAM system by arbitrating the memory accesses from multi-threads [10].

When a memory scheduling simulator replays a multithread application trace, the key problems in the simulation are: since no data value in the trace can be used to determine the inter-thread behavior, 1) the thread's actions in the interleaved critical regions are hardly aligned with the trace collection process, and 2) barriers do not take effect to force asynchronous threads to act as if they were synchronous. In this sense, the traces will be skewed and misplaced. The trace misplacement refers to a situation that occurs when a misaligned lock acquiring order or an ineffective barrier causes a thread/core to run ahead or behind others in trace replaying. We define misplaced interference as interference from a thread running a misplaced trace. When the trace misplacement occurs, the memory system evaluated with the proposed scheduling algorithm will be stimulated by the pressure of misplaced conflict and interference in the shared resource, such as memory channels, row buffers, and DRAM banks. Moreover, misplaced multi-traces may reflect a particular disorder that may not even occur in the real system. Consequently, the trace-driven simulation will report a misleading evaluation metric result for a proposed new memory scheduling algorithm in multi-thread applications. As shown in our experiment (see the details in section 4), trace misplacement causes maximum error rate of 10.22% in terms of the metrics: weighted IPC speedup, harmonic mean of IPC, and CPI throughput.

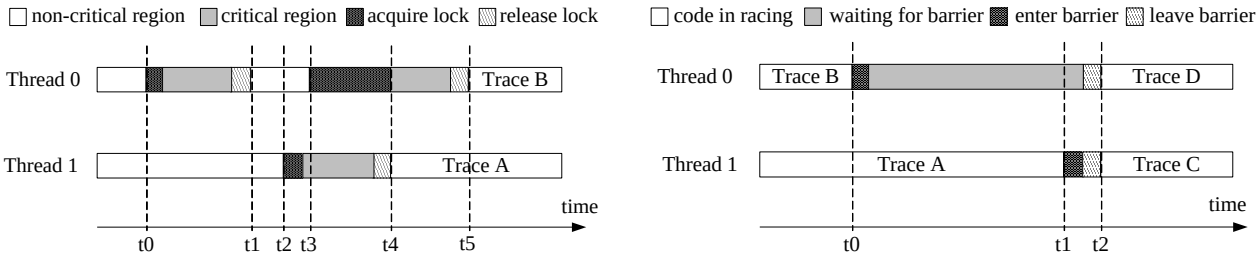
Our goal is to design a methodology that maintains inter-thread ordering and synchronization consistency in trace capturing and replaying, and improves the preciseness of memory scheduling trace-driven simulation in parallel multithread applications.

This paper makes contributions as follows. We propose a simple methodology to avoid trace misplacement when simulating a memory scheduling algorithm in parallel multithread applications. At the trace collection time, we define some critical instrumentation points to capture the inter-thread actions in a multithread application memory trace, especially through intercepting the lock acquiring order and barrier synchronization in the execution. Then, at the trace-replaying time, the lock acquiring order and barrier synchronization are precisely reproduced. Additionally, the simulator injects an appropriate pre-written instruction trace generated from the template into the execution points. This results in the trace replay not skewing the contention in the shared resources. This simulation methodology takes full advantage of the existing instrumentation technology and trace-driven simulation, opens a new opportunity to simplify the inter-thread ordering and synchronization in trace capturing and replaying, and helps to improve the precision of memory scheduling trace-driven simulation in parallel multithread applications.

The rest of this paper is organized as follows: Section 2 gives a description of our motivation, we present the details of our methodology in Section 3, and we show our experiments and results in Section 4. Section 5 is a discussion of the coverage and opportunity, and Section 6 describes related work. Finally, we conclude our paper with the future possible work in Section 7.

2. MOTIVATION

The lock and barrier are commonly used in multi-threaded applications to protect a critical region and to avoid synchronization problems, respectively. The lock ensures the critical region's mutual exclusion, which means that at any given time only one thread acquires the lock successfully and executes the critical region, and other threads attempting to access the same lock must wait until the lock is released. Thus, the traces produced from the execution of the critical region have the same global order as the lock acquiring. Figure 1 (a) shows a common scenario of the lock in a multithreaded application: operations between a producer thread and a consumer thread. In such type of operations, the data processed by the producer thread 1 are transferred to the consumer thread 0 through a shared list. The insertion and removal of the ready data are protected by the critical region to prevent competition that will corrupt the list. In that execution, the consumer thread 0 is always waiting for the producer thread 1 to fulfill the list. In the example, thread 0 acquires a lock at t0, and runs the critical region code to fetch the element from the list, but finds that the list is empty, then releases the lock at t1. Thread 1 acquires the lock at t2, injects a new element into the list, and then releases the lock at t4. In that process, thread 0 again tries to acquire the lock at t3, but waits to enter the critical region until t4, then fetches the new element from the list, and finally releases the lock at t5 and begins to process it. In the trace replay stage, since there is no data value in the trace of the critical region, the simulator will not determine if the list is empty or not and just blindly runs through the trace. Thus, the interactions between the two threads recorded in the trace will not be precisely reproduced in the simulation. For example, in the simulation, if thread 0 runs faster than thread 1, thread 0 will acquire the lock and successfully enter the critical region twice, then begin to run trace B in which thread 0 processes a new element. But thread 1 has not acquired the lock and has not entered the critical region to insert the new element into the list. This disorder of the critical regions will never occur in a real machine since the consumer will not process a new element before the producer generates it. From the memory scheduling point of view, the interference in the shared memory resources occurring in the element processing progress (trace A) and the element generation (trace B) will be lost when the thread 0 trace is replayed ahead of the thread 1 trace. This is trace misplacement. The worse situation is that once



(a) critical region and lock (b) barrier and waiting

Figure 1 Trace alignment determined by lock and barrier

trace misplacement occurs, misplacement will increasingly take place in the remaining parts of the trace files. Fortunately, however, if we maintain the same lock-acquiring order as in the trace capturing process, the simulation will ensure a correct critical region order. Thus, the interference in the memory shared resource will occur on trace A and trace B. Otherwise, the stimulus to memory scheduling algorithm is skewed and does not represent the precise interference through the traces.

A barrier in multi-thread application traces is similar with the lock. Since a thread that reaches a barrier must be blocked until all other threads reach the barrier, this implicitly maintains a global order between the traces on both sides of the barrier. This order must be reproduced in the simulation. Figure 1 (b) shows an example of two threads under the synchronization of a barrier. The trace D behind the barrier must be replayed after trace A when all threads reach the barrier. No memory scheduling algorithm will break this restriction, which means the interference in the shared resource to be optimized by memory scheduling will always exist between trace A and B, or between trace C and trace D. It is impossible for the memory scheduling algorithm to attempt to improve system speedup by optimizing the interference between trace A and D.

From these examples, we can conclude that the trace-driven simulation of memory scheduling in a multi-threaded application must ensure the inter-thread order and synchronization consistency with the trace producing process to maintain the correct interference and contention in the shared memory resources.

3. METHODOLOGY

This paper proposes a new methodology that allows trace-driven tools to precisely simulate the memory scheduling algorithm in a multithread application. To achieve this goal, we designed two components to ensure the inter-thread order consistency between the trace collecting and trace replaying. The first component is a set of instrumentation positions that recognize two important types of execution points: the lock execution point and the barrier execution point. At these execution points, the instrumentation annotates the inter-thread ordering sequence in the trace. The second component is a precise trace replaying method in a trace-driven tool. This method aims to eliminate the trace skew or misplacement in the simulation, as well as to inject an appropriate dynamic instruction trace into the simulation model according to the inter-thread action information.

Figure 2 illustrates the conceptual scheme of the methodology, which consists of two major steps from top to bottom. The first step collects the workload's traces by the instrumentation, which particularly monitors the lock and barrier execution points in the dynamic instruction stream. The application is instrumented at the key positions to record the inter-thread ordering and synchronization, which are annotated in the trace. The second step feeds the trace, including the ordering and synchronization annotation, into a memory scheduling simulation model, along with a set of pre-analyzed trace piece templates. The order and synchronization are faithfully maintained by the simulator. When the simulation reaches the annotated execution point, the simulator assembles the annotation into the trace piece template and injects the appropriate annotated trace pieces into the simulation according to the ordering requirement. By this means, the simulator avoids the misplacement-caused disorder of critical region traces and the synchronization problem. Thus, the memory scheduling simulation will be stimulated by the precise interference among threads.

The following two sections will describe the detailed implementations of the two components in the methodology.

4.1 Instrumentation positions

Thanks to the compilation technology, we can instrument additional functions into any workload execution points, statically or dynamically. Our proposed methodology makes full use of the instrumentation technology, and annotates the important ordering and synchronization information in the trace. Two important execution points need be annotated: the lock execution point and the barrier execution point, which are in charge

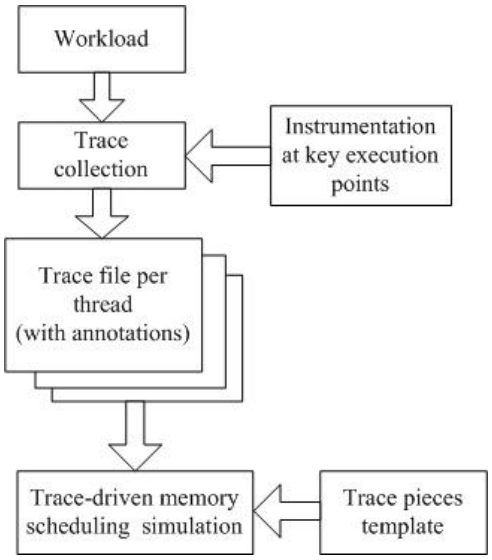


Figure 2 Conceptual scheme of methodology

of two types of trace misplacement: the disorder of the critical regions and the thread's synchronization problem, respectively.

Figure 3 shows the instrumentation positions in the workload code to annotate the execution points discussed above. Since there are two kinds of lock operations, the lock execution points consist of the acquiring lock and the releasing lock. Any point has one entrance and one exit. Thus, there are a total of four instrumentation positions for the lock: before-acquiring, after-acquiring, before-releasing, and after-releasing. The before-acquiring represents the potential start of waiting on a lock when this thread fails acquisition, while the after-acquiring means the deterministic end of waiting on a lock when this thread wins acquisition. On the contrary, the before-releasing serves as the potential end of waiting on a lock if other threads are acquiring a lock in the race, while the after-releasing plays a role of expressing the deterministic end of waiting on a lock if one of the other threads wins the competition. The barrier instrumentation positions consist of before-entering and after-leaving. The before-entering indicates the potential waiting on a barrier if the thread wins the race, while the after-leaving hints at the deterministic start of the race for the next barrier. Additionally, both the lock and the barrier have the before-initializing positions, which are used by the instrumentation to build an internal data structure for the trace collection.

All these execution points have different annotation responsibilities in the instrumentation function. The goal is to collect the ordering and synchronization in the trace file, as well as to filter the trace between the positions of before-and-after pairs. The instrumentation at the lock execution points needs to maintain one counter per lock. At the lock

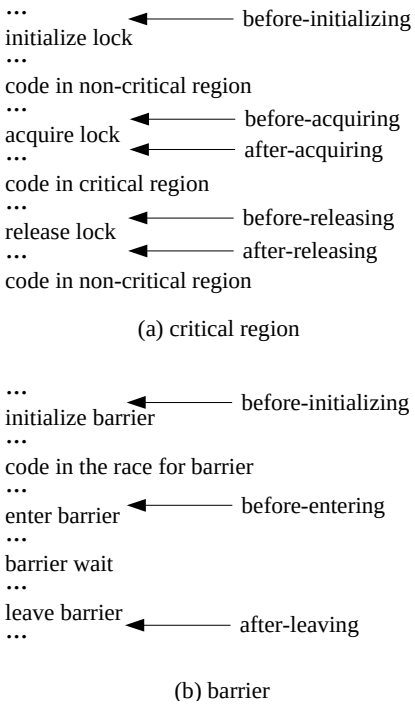
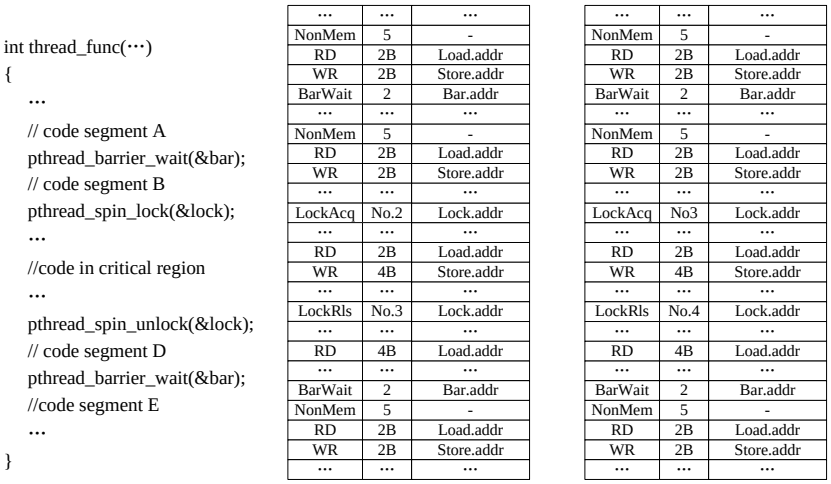


Figure 3 Key execution points to be instrumented

before-acquiring position, the instrumentation annotates the lock ID (typical lock address) in the trace, and stops collecting the trace. At the lock after-acquiring position, the instrumentation gathers the value of the lock's corresponding counter in the trace and restarts the trace collection. At the lock before-releasing position, the instrumentation annotates the lock ID in the trace and again stops trace collection. More importantly, the lock's counter will be increased by one at this position and tagged in the trace again, which means that the next thread order will be allowed to enter the critical region. The increment need not be re-protected by a new lock in the instrumentation since this operation is still under the protection of an unreleased application lock. At the lock after-releasing point, the instrumentation needs to do no more work than just restarting trace collection. The instrumentation at the barrier execution point is simpler than at the lock, and only needs a variable per barrier, which is initialized at the barrier's before-initializing position as the total number of threads that the barrier defends against. At the barrier's before-entering position, the instrumentation tags both the barrier ID (typical barrier address) and the number of racing threads in the trace, and stops the trace collection. After all threads reach the barrier, the instrumentation restarts the trace collection at the barrier's after-leaving position. In this way, the dynamic instruction trace between the pair of before-and-after positions will be filtered out in the final trace file. Only the lock ordering and barrier synchronization tags are annotated in the trace. Figure 4



```
int thread_func(…)
{
    …
    // code segment A
    pthread_barrier_wait(&bar);
    // code segment B
    pthread_spin_lock(&lock);
    …
    //code in critical region
    …
    pthread_spin_unlock(&lock);
    // code segment D
    pthread_barrier_wait(&bar);
    //code segment E
    …
}
```

…	…	…
NonMem	5	-
RD	2B	Load.addr
WR	2B	Store.addr
BarWait	2	Bar.addr
…	…	…
NonMem	5	-
RD	2B	Load.addr
WR	2B	Store.addr
…	…	…
LockAcq	No.2	Lock.addr
…	…	…
RD	2B	Load.addr
WR	4B	Store.addr
…	…	…
LockRls	No.3	Lock.addr
…	…	…
RD	4B	Load.addr
…	…	…
BarWait	2	Bar.addr
NonMem	5	-
RD	2B	Load.addr
WR	2B	Store.addr
…	…	…

…	…	…
NonMem	5	-
RD	2B	Load.addr
WR	2B	Store.addr
BarWait	2	Bar.addr
…	…	…
NonMem	5	-
RD	2B	Load.addr
WR	2B	Store.addr
…	…	…
LockAcq	No3	Lock.addr
…	…	…
RD	2B	Load.addr
WR	4B	Store.addr
…	…	…
LockRls	No.4	Lock.addr
…	…	…
RD	4B	Load.addr
…	…	…
BarWait	2	Bar.addr
NonMem	5	-
RD	2B	Load.addr
WR	2B	Store.addr
…	…	…

(a) Pthread lock/unlock/barrier (b) thread 0 trace (c) thread 1 trace

Figure 4 Simple example trace with locks and barriers

shows a simple example trace with locks and barriers, collected from the Pthread program by the instrumentation through Pin [19]. Each entry in the trace consists of three fields: operation, size and address. In the annotation entry, the operation field gives the type of the execution point, and the size field represents the lock order and the number of threads per barrier, respectively.

4.2 Precise trace-replaying

To provide the precise order consistency with the trace collection, the trace-driven simulator must control the trace replay progress according to the order and synchronization tags annotated in the trace, as well as the stimulus to the memory system. To achieve the first goal, the simulator needs to build an internal counter and a mapping table to track the lock order and the barrier synchronization. We propose that the precise trace-replaying simulation in a multi-thread application runs as follows. The simulator runs all traces on the cores at the beginning as the traditional trace-driven tools until any thread meets the annotated tags (i.e. lock acquiring, lock release, and barrier). At that time, the simulator compares the internal state with the tag to determine if the next trace segment from the trace file will be run, instead of blindly continuing to run through the trace. This means that the lock and barrier operations are determined by both the tags and the current simulation status. For example, if the simulation reaches a lock acquiring tag, the decision of whether to enter the critical region or not is made by the comparison result of the internal order counter and the tagged lock order.

To achieve the second goal, the simulator cannot merely stop the instruction fetching from the trace in the situation in which the thread should be waiting for the lock, but loads the proper trace into the pipeline since the cache hierarchy (particularly the cache-coherent protocol) needs the stimulus from the trace filtered out at the collection stage. We propose building a trace piece template [5] since the traces at the lock and the barrier execution points are always uniform and repeated. These traces can be reproduced from the template. Furthermore, the simulator can build an internal table mapping trace to thread to simulate the non-uniform memory access in a CMP when cores have different distances to the memory controller.

4. EVALUATION

In order to show the effect of the proposed methodology when evaluating multithread applications, we implemented the instrumentation function using Pin [19] and a cycle-accurate trace-driven memory scheduling simulator (re-written from the simulator in [10]). We used a subset of the PARSEC [20] benchmark with locks and barriers, where each benchmark runs on 8 threads in parallel. We collected traces of 100 million instructions per thread at the same execution period in the region of interest (ROI). Then, we fed the traces to memory scheduling simulators twice. The first time, we used the proposed trace-replaying method to simulate four memory scheduling algorithms: FR-FCFS [1], ATLAS [8], STFM [9] and PA-BS [10]. These scheduling algorithms have already been evaluated by multi-programmed workloads in [8, 9, 10], where the trace misplacement as described does not exist. We re-evaluated them in the context of multithread workloads to show how the trace-misplacement problem impacts the evaluation of multithread applications. The second time, we ignored the disorder of critical regions and the synchronization of barriers in the trace and reran the simulation. In both simulations we measured three major metrics: weighted IPC speedup, harmonic mean of

IPC, and CPU throughput, which are commonly used in the memory scheduling research area. Finally, we compared the two sets of metric results. Although these metrics do not precisely show which scheduling algorithm is the best one in multithread applications, we can use them to prove the remarkable error caused by trace-misplacement existing in two simulation methodologies. Figure 5 shows the normalized error between them. As can be seen, each metric has an error rate of nearly 10%. Putting four memory scheduling algorithms together, the geometric average error on weighted IPC speedup, harmonic mean of IPC, and CPI throughput are 10.22%, 9.30%, and 10.22%, respectively. This experiment shows that if we do not avoid the trace misplacement in the trace-driven simulation in multi-thread applications, the errors in the metric will be large enough to lead to erroneous conclusions, thus misleading the memory scheduling algorithm design.

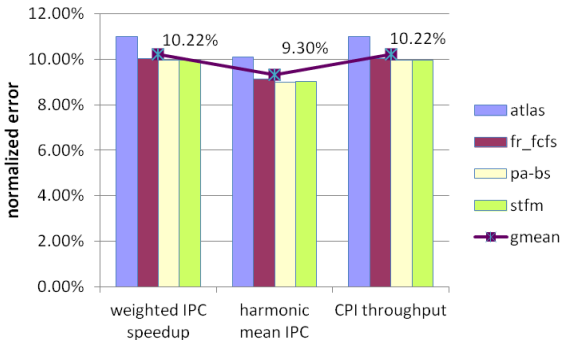


Figure 5 Normalized error between two simulations

5. COVERAGE AND OPPORTUNITIES

Our proposed methodology covers memory scheduling simulation in a multithread application, which contains the clear encapsulation of lock and barrier operations, such as spinlock/mutex in the low level programming model Pthread. For a multithread application using a high level programming model, such as OpenMP, the instrumentation technology will find these important execution points and our methodology can also be applied to them. However, the lock and synchronization embedded by a user in application code, such as the condition variable in Pthread, needs more care to select the execution points to be instrumented. It may be additional work for the researcher to analyze the application code, but this is the best one can do to obtain an opportunity to study a new memory system in it.

Non-determinism [12] in simulating a multithread application is an important challenge to deal with. One of a number of reasons is that the system scheduling and traffic on a cache-coherent system may cause a different order of threads acquiring locks. For multithread applications with dynamic load balance, the order of locks determines the amount of work performed by threads when they leave critical regions. It is hard, even impossible work to faithfully replay changes in the amount of work in trace-driven simulation. Thus, for given traces captured from these applications in a real system, recording the order of lock events and replaying them is a necessary and correct method regarding the traces after releasing locks. The trace-driven simulation in our methodology completely eliminates non-determinism since the blocks, which consist of the instructions enclosed by a pair of execution points in a trace, are replayed in the same order as they are captured. Thus, this deterministic character ensures that a new memory scheduling algorithm or a new memory system is under the stimulus from the appropriate traces, instead of from the misplaced. The determinism in our methodology opens a door to study such applications' inter-thread behavior on new memory systems through trace-driven tools. However, for multithread applications with static load balance, in which changes to the order of locks will not impact the amount of work of each thread, the deterministic replay methodology will lose opportunities to study acceleration efforts on non-critical threads. It will be future work to differentiate these two types of multithread applications on-the-fly in trace-driven simulations.

6. RELATED WORK

Because of the reasons mentioned in Section 1, in the past few years, memory scheduling studies [1, 8, 9, 10, 11] have commonly used the trace-driven methodology to evaluate proposed scheduling algorithms in multi-program applications. Since the multi-programs running in multi-core systems do not need precise inter-program ordering and synchronization, the trace-driven simulation becomes the preferred methodology to evaluate the proposed memory scheduling algorithm. An appropriate policy rule to prioritize multi-program's memory access in the traces will remove [6, 7] the interference and contention in shared resources. This misplacement represents the precise effect of the memory scheduling algorithm in multi-program applications. However, the multithread application has inherent restrictions on memory access interference among threads because of inter-thread actions, and the simulation should not misplace the trace regardless of the scheduling algorithm. A very recent study [14], which uses full processor and memory controller simulation to evaluate the effect of memory scheduling and devise new memory scheduling algorithms for multithreaded applications, has taken into account effects of memory scheduling in parallel multithreaded applications. This paper models threads waiting for locks and barrier synchronization events. As such it provides a complementary method that avoids trace misplacement by simulating the synchronization that happens between threads.

In a recent trace-driven simulation study of multithread applications [13], a cooperative mechanism is proposed in which a methodology of combining execution-driven simulation and trace-driven simulation provides complementary ways to resolve the weakness of trace-driven simulation in scheduling the parallel code execution in different threads. In memory scheduling research areas, sampling simulation [17, 18] is widely used to select the representative memory access traces as the stimulus. In this sense, the number of threads is always constant (equal to the total number of cores); therefore, there is no need for dynamic thread scheduling.

An emergent variant of the trace-driven simulation in multithread application is the frameworks using dynamic binary instrumentation tools or a functional emulator to feed the "online-trace" to a special target simulation model. CMP$im [15] belongs to this type of simulation in multithread applications. Under the help of an AMD functional engine, COTSon [16] decouples the functional and timing simulation. Compared with the pure trace-driven simulation, this approach saves disk space requirements of the trace file in the cost of simulation run time, since each exploration in the design space will re-run the native workload plus the simulation.

7. CONCLUSION AND FUTURE WORK

The methodology presented in this paper puts the emphasis on how to avoid trace misplacement in multithread applications when simulating the memory scheduling algorithm. We propose annotating the lock order and barrier synchronization in the trace to replay the deterministic inter-thread actions in the simulation. The experiment shows that without our methodology the accuracy of memory scheduling simulation in parallel multithread applications will be impacted by the remarkable errors. To the best of our knowledge, this is the first attempt to maintain the consistency order of trace pieces in simulation to ensure correct interference in a memory system.

Finally, the described methodology provides a reliable method to simulate the memory scheduling algorithm in a multithread application. In the future, more complex and important execution points can be instrumented, and the method to annotate behaviors of applications, such as dynamic or static load balance, from a higher level will be studied.

8. ACKNOWLEDGMENTS

We would like to thank all reviewers for improving this paper. This work is partially supported by the National Science Foundation of China under grant numbers 61003062, 60903046, 60925009 and 60921002, and the National Basic Research Program of China (973 Program) under grant No. 2011CB302502.

9. REFERENCES

[1] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. 2000. Memory access scheduling. *In Proceedings of the 27th International Symposium on Computer Architecture*, pp.128-138, June 2000.

[2] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. 2012. Bottleneck identification and scheduling in multithreaded applications. *In Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12).* ACM, New York, NY, USA, 223-234.

[3] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. 2009. Accelerating critical section execution with asymmetric multi-core architectures. *In Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09).* ACM, New York, NY, USA, 253-264.

[4] R. A. Uhlig and T. N. Mudge. 1997. Trace-driven memory simulation: a survey. *ACM Comput. Surv*. 29, 2 (June 1997), 128-170.

[5] M. Xu, R. Bodik, and M. D. Hill. 2003. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. *In Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA '03).* ACM, New York, NY, USA, 122-135.

[6] H.-Y. Cheng, C.-H. Lin, J. Li, and C.-L. Yang. 2010. Memory Latency Reduction via Thread Throttling. *In Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, Washington, DC, USA, 53-64.

[7] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. 2010. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. *In Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*. ACM, New York, NY, USA, 335-346.

[8] Y. Kim, D. Han, O. Mutlu, and M. H.-Balter. 2010. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers, *In IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*, vol., no., pp.1-12, 9-14 Jan. 2010

[9] O. Mutlu and T. Moscibroda. 2007. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. *In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 40)*. IEEE Computer Society, Washington, DC, USA, 146-160.

[10] O. Mutlu and T. Moscibroda. 2008. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. *In Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, Washington, DC, USA, 63-74

[11] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. 2006. Fair Queuing Memory Systems. *In Proceedings of the 39th Annual IEEE/ACM International Symposium on*

Microarchitecture (MICRO 39). IEEE Computer Society, Washington, DC, USA, 208-222.

[12] A. R. Alameldeen and D. A. Wood. 2003. Variability in Architectural Simulations of Multi-Threaded Workloads. *In Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA '03)*. IEEE Computer Society, Washington, DC, USA, 7-.

[13] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, and M. Valero. 2011. Trace-driven simulation of multithreaded applications. *In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '11)*. IEEE Computer Society, Washington, DC, USA, 87-96.

[14] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt. 2011. Parallel application memory scheduling. *In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44 '11)*. ACM, New York, NY, USA, 362-373.

[15] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. 2008. CMP$im: A Pin-based on-the-fly multi-core cache simulator. *In the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, co-located with ISCA'2008

[16] E. Argollo, A. Falcon, P. Faraboschi, M. Monchiero, and D. Ortega. 2009. COTSon: infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev.* 43, 1 (January 2009), 52-61.

[17] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. 2002. Automatically characterizing large scale program behavior. *In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*. ACM, New York, NY, USA, 45-57.

[18] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. 2004. Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation. *In Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 37)*. IEEE Computer Society, Washington, DC, USA, 81-92.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190-200.

[20] C. Bienia, S. Kumar, J. P. Singh, and K. Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. *In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 72-81.