

HMTT: A Hybrid Hardware/Software Tracing System for Bridging the DRAM Access Trace's Semantic Gap

Yongbing Huang, Licheng Chen, Zehan Cui, State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, China; University of Chinese Academy of Sciences, China
Yuan Ruan, Yungang Bao, Mingyu Chen, and Ninghui Sun, State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, China

DRAM access traces (i.e., off-chip memory references) can be extremely valuable for the design of memory subsystems and performance tuning of software. Hardware snooping on the off-chip memory interface is an effective and non-intrusive approach to monitoring and collecting real-life DRAM accesses. However, compared with software-based approaches, hardware snooping approaches typically lack semantic information, such as process/function/object identifiers, virtual addresses, and lock contexts, that is essential to the complete understanding of the systems and software under investigation.

In this paper, we propose a hybrid hardware/software mechanism that is able to collect off-chip memory reference traces with semantic information. We have designed and implemented a prototype system called HMTT (Hybrid Memory Trace Tool), which uses a custom made DIMM-connector to collect off-chip memory references and a high-level event-encoding scheme to correlate semantic information with memory references. In addition to providing complete, undistorted DRAM access traces, the proposed system is also able to perform various types of low-overhead profiling, such as object-relative accesses and multithread lock accesses.

Categories and Subject Descriptors: B.8 [Performance and Reliability]: Performance Analysis and Design Aids; C.4 [Performance of Systems]: Measurement Techniques; D.1.3 [Programming Techniques]: Concurrent Programming—Parallel Programming; D.4 [Operating Systems]: Miscellaneous

General Terms: Design, Measurement, Performance

Additional Key Words and Phrases: Hybrid tracing mechanism, DRAM access trace, High-level event, Semantic gap, Lock, Object, Function

Authors' addresses: Yongbing Huang, huangyongbing@ict.ac.cn; Licheng Chen, chenlicheng@ict.ac.cn; Zehan Cui, cuizehan@ict.ac.cn; Yuan Ruan, ruanyuan@ict.ac.cn; Yungang Bao, baoyg@ict.ac.cn; Mingyu Chen, cmy@ict.ac.cn; Ninghui Sun, snh@ict.ac.cn.

Extension of Conference Paper: (1) Yungang Bao, Mingyu Chen, Yuan Ruan, Li Liu, Jianping Fan, Qingbo Yuan, Bo Song, Jianwei Xu, HMTT: A platform independent full-system memory trace monitoring system, *International Conference on Measurement and Modeling of Computer Systems*, 2008. (2) Yongbing Huang, Zehan Cui, Licheng Chen, Wenli Zhang, Yungang Bao, Mingyu Chen, HaLock: Hardware-assisted lock contention detection in multithreaded applications, *International Conference on Parallel Architectures and Compilation Techniques*, 2012. (3) Licheng Chen, Zehan Cui, Yungang Bao, Mingyu Chen, Yongbing Huang, Guangming Tan, A lightweight hybrid hardware/software approach for object-relative memory profiling, *International Symposium on Performance Analysis of Systems & Software*, 2012.

This work is supported by the National Natural Science Foundation of China (NSFC) under the grant number 60925009, 61272132 and 60903046, the National Basic Research Program of China (973 Program) under the grant number 2011CB302502, the Strategic Priority Research Program of the Chinese Academy of Sciences under the grant number XDA06010401, and the Huawei Research Program under the grant number YBCB2011030. Yungang Bao is partially supported by the CCF-Intel Young Faculty Research Program (YFRP) Grant.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© YYYY ACM 1544-3566/YYYY/01-ARTA \$15.00

DOI: <http://dx.doi.org/10.1145/0000000.0000000>

1. INTRODUCTION

With almost two decades of intensive research activities, “memory wall” [Wulf and McKee 1995] continues to challenge the processor and memory system designers. Analysis of memory traces, especially DRAM access traces, is an important method for both hardware and software designs addressing the memory wall problem. DRAM access traces are typically collected through software simulators, source code or binary instrumentation, hardware emulators, and hardware monitoring devices. Each approach has its pros and cons. The fastest approach [Bao et al. 2008] is to use hardware monitoring devices.

Hardware monitoring devices for DRAM traces normally snoop on the memory ports. They are able to collect complete, undistorted physical addresses sent to the memory system. In contrast to software-based approaches, conventional hardware-snooping approaches cannot detect the program information only seen within the processor chip, causing a semantic gap between the collected traces and program events such as thread context, object identification, virtual addresses, lock operation, and I/O operations.

To bridge this semantic gap, we propose a hybrid hardware/software mechanism that is able to collect DRAM references and then correlate them with semantic events. This mechanism integrates a flexible software high-level-event-encoding mechanism into a conventional hardware-snooping mechanism. It uses a hardware snooping device connected to DRAM slots to capture all DRAM accesses. Semantic events are transferred to the snooping device using designated memory requests. When software such as the runtime system or the OS detects the occurrence of an event of interest, it injects special memory requests to pass the needed details of the event to the monitoring hardware. These special memory requests are called synchronous requests. Synchronous requests contain semantic information and are produced by a high-level-event-encoding mechanism, called HLE2M. The hardware components capture both normal memory reference requests and synchronous requests in its traces. To simplify the design, the traces are post-processed to remove the special requests and correlate semantic information to the real memory references.

Based on the design above, we have implemented a prototype system called HMTT (Hybrid Memory Trace Tool). In the prototype system, a board of HMTT is plugged into a DIMM slot. The prototype supports only DDR/DDR2/DDR3 memory interfaces, and monitors only memory addresses but not memory data. We have used several techniques to overcome the challenges that the design of this system presents. (1) To keep up with the speed of memory, the DDR state machine [JEDEC Solid State Technology Association 2004] has been simplified to match the high speed. (2) To handle the enormous size of DRAM access traces, we use a combination of a PCI-E cable and a RAID system to transfer and record these traces. Meanwhile, to reduce the offline analysis time, we trim the DRAM access traces and pick out only some representative memory slices, according to the hardware performance events collected for each memory slice at runtime. (3) To make the hardware aware of the occurrence of high-level events, we apply our high-level-event-encoding mechanism to various scenarios.

Based on the primitive functions that the HMTT system provides, we are able to analyze the behavior of many high-level events efficiently with little overhead. Comprehensive validations and evaluations have shown that the HMTT system has the advantages of both hardware and software [Bao et al. 2008]. In summary, the HMTT system has the following key characteristics.

- *Detail.* The DRAM access traces include both virtual and physical addresses, access type (read, write, etc.), timestamp, and semantic information, such as object id, function id, thread id, virtual machine id, and lock operation contexts.

- Low overhead.* The hardware part of HMTT can itself collect undistorted memory addresses with nearly no overhead. The overhead of the HMTT system comes only from the software part. The software part creates a small overhead because there is mainly one extra memory request for each single event. The total overhead can be controlled flexibly by the software part by dynamically selecting the events to be monitored. Thus, HMTT can achieve little interference in most cases compared with similar tools, and correspondingly obtain undistorted DRAM access traces with only a small slowdown.
- Portability.* The hardware component of HMTT can work on any systems with DDR DIMM slots, regardless of the ISA, the processor, and the network. The software components of HMTT support both Linux and Windows.

The HMTT system has already been utilized in many situations, from basic DRAM access trace collection to various types of analysis of semantic information. Generally, HMTT can be used in two different ways. The first way is to obtain annotated DRAM access profiles. The other way is to use it as a low-overhead output-data-streaming framework by transforming high-level events into memory addresses. By leveraging the HMTT system, we have produced several high-level-event-profiling tools, for purposes such as object profiling, function profiling, and lock profiling.

In this paper, we have chosen two typical case studies to describe the above two ways of using HMTT and to illustrate its efficiency. In the first case study, we introduce object profiling, which studies the DRAM access patterns for various objects such as arrays and matrixes. By annotating memory addresses with object information, we can distinguish regular access patterns for certain objects, and consequently point out optimization directions. In the second case study, we used the HMTT system to detect and record lock operations in multithreaded applications. For each lock operation considered as a high-level event, we encoded its semantic information into a single memory address and stored it on another machine, instead of storing this information directly into the local memory buffer as is done in software-based profiling tools. Although many multithreaded applications are quite sensitive to cache and memory resources, our method causes little cache and memory interference. By comparing the lock behavior obtained with HMTT and previous lock-profiling tools, we have shown that many lock-profiling tools have nonnegligible distortion and inaccuracy problems. In summary, the case studies show the feasibility of a hybrid hardware/software tracing mechanism, and also demonstrate that our HMTT system is capable of profiling various high-level events with low overhead.

The rest of this paper is organized as follows. Section 2 discusses the semantic gap between DRAM traces and high-level events. Section 3 describes the hybrid hardware/software tracing mechanisms. Section 4 presents techniques addressing the challenges in the design and implementation of HMTT. The prototype of HMTT is described in Section 5 and its evaluation is given in Section 6. The object-profiling tool is illustrated in Section 7, and the lock-profiling tool is in Section 8. Section 9 contains an overview of related work. Section 10 summarizes our work.

2. THE SEMANTIC GAP BETWEEN DRAM ACCESS TRACES AND HIGH-LEVEL EVENTS

Memory trace analysis is a useful method to guide optimization both for architecture researchers and for application programmers. The commonly used memory traces can be divided into two types: full memory traces and DRAM access traces. A *full memory trace*, or *memory trace* for short, is a sequence of memory addresses that are touched by all load and store instructions. In contrast, a *DRAM access trace* contains only memory references that access DRAM memory modules. A DRAM access trace is the subset of a full memory trace that misses out all cache levels, plus any references generated by

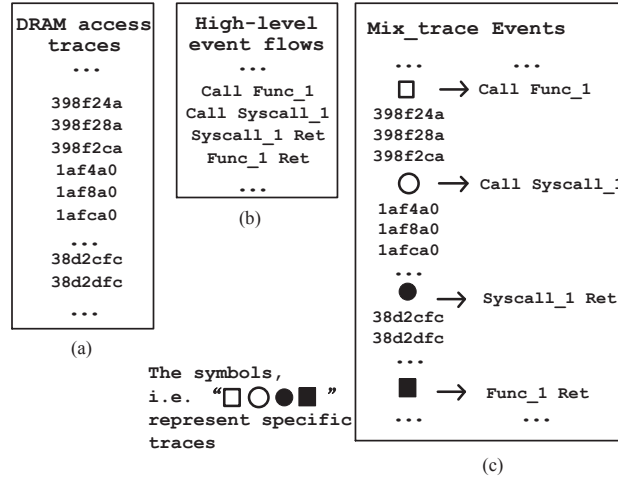


Fig. 1. The semantic gap between DRAM access traces and high-level events. (a) Conventional DRAM access address traces; (b) a typical high-level event flow; (c) correlation of DRAM access traces with high-level events.

hardware prefetching mechanisms. Therefore, a memory trace is quite different from a DRAM access trace. Generally, memory traces are appropriate for studying program behavior such as data dependency, and for studying processor structures such as the TLB and caches. On the one hand, DRAM access traces are widely used as a tool in research on memory systems, such as in the study of memory scheduling, memory materials, and organization. On the other hand, DRAM access traces can point out those memory operations that should be preferentially optimized because of their large access penalties. In this paper, we put emphasis on studying memory systems and optimizing applications' high-latency events. Thus, in the rest of our paper, we describe the collection and analysis of DRAM access traces.

Figure 1(a) shows a conventional DRAM access trace (in which timestamp, read/write, and other information has been removed). Since trace-driven simulation is an important approach to evaluating memory systems and has been used for decades [Uhlig and Mudge 1997], this kind of DRAM access trace has played a significant role in advancing memory system performance. As described in the introduction, DRAM access traces can be collected in various ways, among which hardware snooping is a more efficient approach than others. Usually, the hardware-snooping approach is able to collect complete, undistorted DRAM access traces that include the VMM, OS, library, and applications. Nevertheless, those DRAM access traces contain only low-level (machine-level) information, which is difficult to use for further study.

From the perspective of the system level, a computer system generates various events, such as function calls, object accesses, and lock operations. Figure 1(b) illustrates a typical event flow. To capture the high-level event flow, profiling tools may be used to instrument the source code or the binary at points of these events manually or automatically. In contrast to DRAM access traces, these events are at higher levels and contain more semantic information, which can obviously be used for further investigation. However, the high-level events are usually insufficient for studying the performance and behavior of a system in depth.

Based on the above observations, we can conclude that there is a semantic gap between conventional DRAM access traces and high-level events. If they can be correlated with one another, as shown in Fig. 1(c), it should be significantly helpful for both

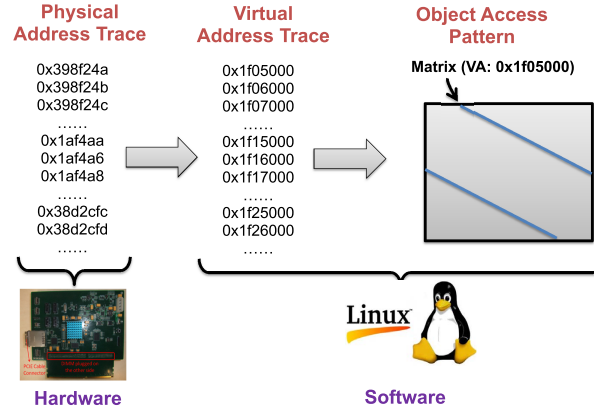


Fig. 2. Example of a hybrid hardware/software tracing mechanism

low-level (DRAM access traces) and high-level (system or program events) analysis. Although a similar gap exists between cache requests and high-level events, many mechanisms such as Intel's PEBS [Intel Corporation 2012] can solve it by directly locating the sources of events for cache requests.

However, the current trace tools can only collect either DRAM access traces or high-level events such as function call graphs and OS events. Some hardware monitors are only capable of collecting whole memory requests by snooping on the memory bus; examples are those described by Alexander et al. [1986] and Fuentes [1993], and Monster [Nagle et al. 1992], MemorIES [Nanda et al. 2000], and ACE [Hong et al. 2006]. For high-level events, gprof can provide only call graphs, and Linux Trace Toolkit [Desnoyers and Dagenais 2006] focus on collecting OS events; however, these have a substantial amount of overhead. In addition, by instrumenting the target program with additional instructions, some instrumentation tools such as ATOM [Srivastava and Eustace 1994], Pin [Luk et al. 2005], and Valgrind [Nethercote and Seward 2007] are capable of collecting more information, for example memory traces and function call graphs. However, such instrumentation may cause extra overhead, disturb program execution, and cause distortion problems. As a result, the information collected by such instrumentation tools may be inaccurate or even useless. Moreover, it is hard to instrument virtual machine monitors and operating systems.

In summary, there is a semantic gap between conventional DRAM access traces and high-level events, but almost none of the existing tools are capable of bridging the gap effectively.

3. A HYBRID HARDWARE/SOFTWARE TRACING MECHANISM

To address the above semantic gap, we propose a hybrid hardware/software mechanism that is able to collect memory reference traces and high-level event information simultaneously.

Figure 2 shows an overview of the use of a hybrid hardware/software mechanism to study the pattern of object accesses. A semantic gap exists here between the DRAM access trace and the object identifiers. On the one hand, low-overhead hardware is used to collect the DRAM access trace. On the other hand, the detailed information used to bridge the gap, such as page table and object management information, is collected by the software at runtime. In order to correlate each physical memory address with the object identifier, we must first translate it into a virtual memory address, with the aid of the OS page table information. Then, according to the object management mapping

information between virtual addresses and objects, we can remap each virtual address to an object identifier.

To perform the above procedures, there are several issues we need to solve. From the point of view of hardware, we adopted a hardware-snooping method in order to collect complete, undistorted DRAM access traces. Considering the importance of portability for hardware monitors, we chose to snoop on memory buses, especially DDR-protocol-based memory buses. In our system, the snooping hardware is plugged into a DIMM slot to monitor only the memory signals that are sent from the memory controller to the memory modules. By referring to the DDR protocols, memory addresses can then be extracted from these signals. Both DDR protocols and DIMM slots are commonly used on various platforms. Hence, this approach is quite general and can easily be ported to other platforms. However, since the frequency of memory such as DDR3 has reached values up to 1600 MHz, the first challenge for the hardware is *how to keep up with the speed of memory*.

Meanwhile, the DRAM access traces produced by a fast memory bus can be extremely large. Usually, the buffer in the snooping hardware is relatively small [Uhlig and Mudge 1997]. If the DRAM access trace cannot be transferred to external storage in time, the system must be stalled when the buffer is full, otherwise the trace will be discarded. Thus, the whole system requires a high-speed trace-recording method. The troubles caused by the large size of DRAM access traces relate not only to recording, but also to offline analysis. Since each DRAM access trace has to be correlated with the high-level information, the time spent on analyzing a large trace can be rather long. Here comes the second challenge, which is *how to handle the enormous size of DRAM access traces*.

So far, we have discussed the hardware shown in the left part of Fig. 2. As for the software, this is responsible for collecting software information related to high-level events, such as the OS page table information and the mapping information between objects and their virtual address zones for object profiling. However, this information is not constant but may be updated when events occur. For instance, the mapping information between the objects and their virtual addresses is altered if any objects are created or destroyed. When we correlate a DRAM access trace with high-level events in the offline analysis, we need to know of the existence of these events or to know the modifications of the software information that occur, from the DRAM access trace. So, the third challenge is *how to make the hardware aware of the occurrence of high-level events* and record these events in the DRAM access trace.

4. DESIGN OF HYBRID HARDWARE/SOFTWARE TRACING MECHANISM

To overcome the above three challenges, we have used several techniques in the design of our hybrid hardware/software tracing system. We shall give details of these challenges and their solutions in the following subsections.

4.1. Challenge 1: Keeping up with memory speed

The speed of memory is improving quickly, from a maximum of 800 MHz for DDR2 memory to 1600 MHz for DDR3 memory. It is difficult to keep up with such memory speeds directly. For instance, most state-of-the-art field-programmable gate array (FPGA) logic cannot achieve such a high frequency. Furthermore, for multibank memories, the memory controller can interleave DDR commands to different banks and consequently access them concurrently, to increase memory bandwidth. The logic used to snoop memory signals has to be made more sophisticated in order to interpret such interleaved commands. Thus, fast, efficient control logic is required to keep up with the speed of memory.

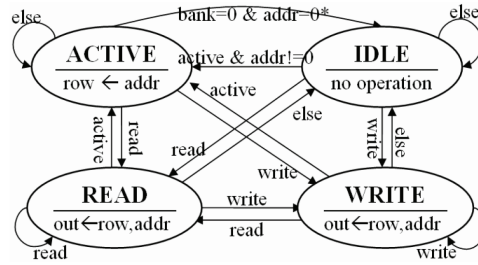


Fig. 3. Simplified state machine. To keep up with memory speeds, the standard DDR state machine [JEDEC Solid State Technology Association 2004] is simplified to match the high speed. (* Note that “addr” is used to filter specific addresses of the configuration space.)

We have adopted two approaches to optimize the control logic. On one hand, since only the memory addresses are indispensable for tracking a trace, we can monitor only the DDR commands at half the memory data frequency. For example, if we use DDR3-800 MHz memory, the control logic can operate at a frequency of 400 MHz, at which most advanced FPGAs can work.

On the other hand, the DDRx SDRAM specification [JEDEC Solid State Technology Association 2004] defines seven commands, and a state machine that has more than twelve states, for interpreting read/write operations. Commercial memory controllers integrate even more complex state machines, which cost both time and money to implement and validate. Nevertheless, we have found that only three commands, i.e., ACTIVE, READ, and WRITE, are necessary for extracting memory reference addresses. Thus, we have designed a simplified state machine to interpret the read/write operations for one memory bank. Figure 3 shows this simplified state machine. It has only four states and performs state transitions based on the above three commands. This state machine is so simplified that an implementation in a common FPGA is able to work at a high frequency. Our experiments show that this state machine implemented in a Xilinx Virtex-6 FPGA is able to work at a frequency of over 400 MHz.

4.2. Challenge 2: Handling the enormous size of DRAM access traces

There are two aspects to handling the enormous size of DRAM access traces. The first is how to buffer and record the large size of the DRAM access trace produced by the snooping logic. The other is how to preprocess the DRAM access trace before correlating it with high-level events in the the offline analysis.

4.2.1. Trace-recording method. Usually, memory reference traces are generated at very high speed. Our experiments show that most applications generate DRAM access traces at bandwidths of more than 30 MB/s even when DDR 200 MHz memory is utilized. Moreover, the high frequency of DDR2/DDR3 memory and the prevalent multichannel memory technology further increase the bandwidth of trace data generation, to up to 800 MB/s.

In order to cope with the high bandwidth of trace generation, our system uses a PCI Express (PCI-E) cable to send DRAM access traces and a RAID system to receive them. The bandwidth of a PCI-E 16x cable can be as high as 8 Gbps, which is capable of handling memory buses whose frequency is less than that of DDR3-800 MHz. A PCI-E module is integrated into the snooping hardware. The PCI-E cable connects the snooping-hardware board with another machine, which acts as a receiver for the DRAM access traces. In the receiver machine, we use the RAID technique to construct very fast, high-capacity storage.

There are several advantages to transferring a DRAM access trace to a specific receiver machine. Because many machines use normal hard disks as their storage medium, their write bandwidth is too limited to record the enormous traces produced by memory modules. Thus, the first advantage is that we can support the profiling of any machine, no matter what kind of storage is used. Another advantage is related to interference. Recording a large amount of data into storage utilizes CPU and memory resources, which consequently contends with the program being profiled and causes interference, such as cache interference and memory interference. If profiled program is sensitive to these resources, the tracing system will disturb its normal execution. By recording the data on another machine, we get rid of this problem.

4.2.2. Trace reduction method. As illustrated in the previous section, the trace generation bandwidth is quite high. For instance, the typical trace bandwidth for the PARSEC [Bienia et al. 2008] benchmark is about 800 MB/s. Particularly for commercial programs run for hundreds or thousands of seconds, the trace dataset is too big to store or move efficiently. Fortunately, most programs have similar patterns during different phases, because of the existence of shared objects such as functions and loops. Thus, we can trim the trace and study only the behavior of representative slices. However, for memory addresses that contain specific semantic information, trace trimming may disturb their semantics.

To pick out representative slices, there are two issues to be solved. The first issue is what metrics should be used to characterize trace slices. The analysis of hardware performance events is a commonly used approach to classifying the microarchitectural characteristics of programs. Here, we have used performance events to distinguish different memory slices. If we understand the behavior of the target program, we can specify performance events with a strong relation to the programs. Otherwise, basic metrics such as cycles per instruction (CPI), cache misses, and memory bandwidth can be measured. The second issue is how to obtain the memory slices with their correlated metrics. Unlike SimPoint [Hamerly et al. 2005], which finds representative program slices and collects memory traces in two phases, our method requires only one phase. The performance metrics and DRAM access trace slices are collected simultaneously in fixed intervals of time. In order to distinguish the DRAM access trace slices belonging to different intervals of time, special memory addresses called *synchronous tags* are injected into the DRAM access trace to mark the start and finish of the trace for each interval. The synchronous tags can be produced using HLE2M, as illustrated in Section 4.3. Figure 4 depicts the principle of trimming a DRAM access trace. The hardware performance event trace for each interval is collected using a lightweight performance tool called TopMC [TopMC 2011]. The trace collected by HMTT contains two parts: the memory data and the synchronous tag.

In the offline analysis, the DRAM access trace is trimmed as follows. First, we split the whole trace into 2 ms trace slices. Then, for each trace slice, we calculate the metrics from the performance event values stored in the performance counter trace. Next, we use a cluster algorithm such as K-means adopted by SimPoint [Hamerly et al. 2005] to classify all the trace slices and pick out representative slices from each classification.

In our experiments, when we used the above trace reduction method to trim a 10 TB raw DRAM access trace and obtain a typical 416GB trace which contains about 4 billion trace items, for a memory-intensive workload running for 40 seconds.

4.3. Challenge 3: Making hardware aware of the occurrence of high-level events

To make the hardware aware of the occurrence of high-level events, we use a high-level event encoding mechanism (HLE2M). The principle of HLE2M is that high-level events are encoded into the memory address space. Each memory address not only

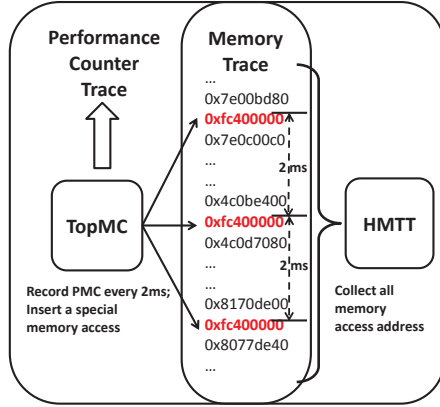


Fig. 4. Principle of trimming a DRAM access trace.

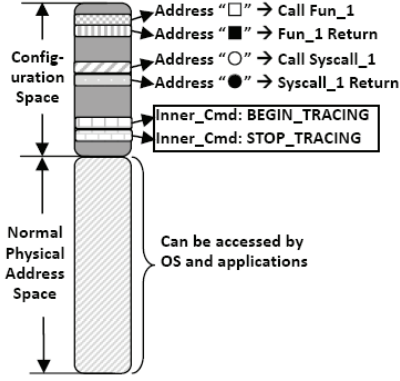


Fig. 5. Two parts of the physical address space of a system: configuration space reserved for HMTT, and normal address space.

indicates one specific place in the memory modules, but also carries semantic information.

For each high-level event, HLE2M produces a unique memory address representing it and then triggers an access to that memory address, which is immediately captured and stored in the DRAM access trace by the snooping hardware logic, as illustrated in Section 4.1. Via this mechanism, the execution flow of high-level events is transformed into a specific memory address sequence in the DRAM access trace.

To perform the above procedure, there are two issues that we must solve. First, HLE2M and the snooping hardware interact with memory addresses, but there exist both normal memory addresses and the addresses used to represent the high-level events. Then, we need to address the question of how the HLE2M software and the hardware should interact with each other using the memory addresses. In addition, the semantic information implied by the various high-level events is varied, so the second issue is how to encode different kinds of semantic information into memory addresses.

4.3.1. Interaction between software and hardware. We address this problem by introducing a specific physical address region, reserved as the hardware device's configuration space, which is transparent to all programs and OS modules except for the tracing-control components and the software-encoding components, as illustrated in Fig. 5. The addresses within the configuration space can be predefined as internal commands of the hardware device, such as `BEGIN_TRACING` and `STOP_TRACING`. They can also represent high-level events, such as function calls and syscall returns. Usually, the size of the configuration space is small.

4.3.2. Workflow of HLE2M. Figure 6 shows the workflow of HLE2M. When a high-level event happens, the runtime system or OS detects this event and then collects the corresponding semantic information, which is often represented using variables (① in the figure). Sometimes, this semantic information is too long to be encoded into a reserved memory address space. Although we can use multiple memory addresses to hold the semantic information for a single event, this may occupy more memory bandwidth, which is a critical resource. However, the essence of a variable is that it is a location to hold an object plus an identity of the object. Since the tracing system does not care about the location of data, HLE2M can replace long variables with short identities (②). The mapping information between the variables and identities is saved for offline

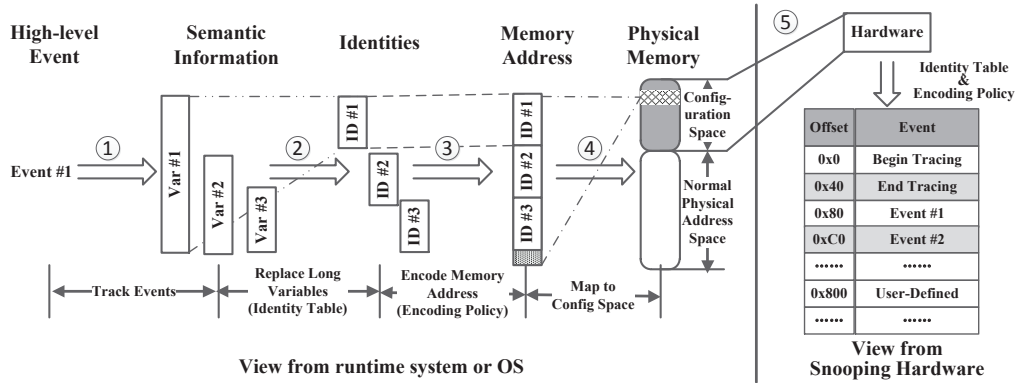


Fig. 6. Workflow of the high-level-event-encoding mechanism HLE2M.

analysis. Next, the short identities are encoded into the memory address space according to the encoding policy (③). Obviously, for different types of high-level events, the encoding policies applied in HLE2M must be adjusted according to the semantic information delivered by the events. In a typical encoding policy, the memory address space is partitioned into several regions. Each region has several bits, and stores one part of the semantic information. Finally, we map the memory address produced in (③) into the uncacheable configuration space (④) and trigger an access to the memory address (⑤) monitored by the snooping hardware.

5. IMPLEMENTATION OF THE HMTT TRACING SYSTEM

Based on the above hybrid hardware/software tracing mechanism, we have designed and implemented a prototype system called HMTT (Hybrid Memory Trace Tool). There are many implementation issues, such as designing the hardware logic using an FPGA [Bao et al. 2008], detecting the memory address space available for HLE2M [Huang et al. 2012], and accessing the configuration space by software [Bao et al. 2008]. In this section, we will only introduce the implementation of event detection, and the framework of HMTT.

5.1. Detecting high-level events

We have already described in the previous section how the hardware is notified when high-level events happen. However, besides this notification, many other types of information need to be collected for high-level events, such as the OS page table information required to translate physical addresses to virtual addresses as shown in Fig. 2. Because the hardware cannot obtain detailed event information directly, the software components are responsible for collecting this correlated information.

The first step is to detect the annotation point of each high-level event. The ideal method is to monitor high-level events dynamically, without any modification to the target program. For example, single-step execution or the use of breakpoints can achieve this goal. However, although these methods are quite flexible, they come at some cost, in terms of overall execution slowdown and interference. Instead, we statically annotate the high-level events. Extra instructions are inserted around high-level events for each application in order to accomplish two tasks. The first task is to notify the snooping hardware of the occurrence of events using HLE2M. The second task is to gather additional correlation information, that are used to assist the mapping between memory addresses and high level events.

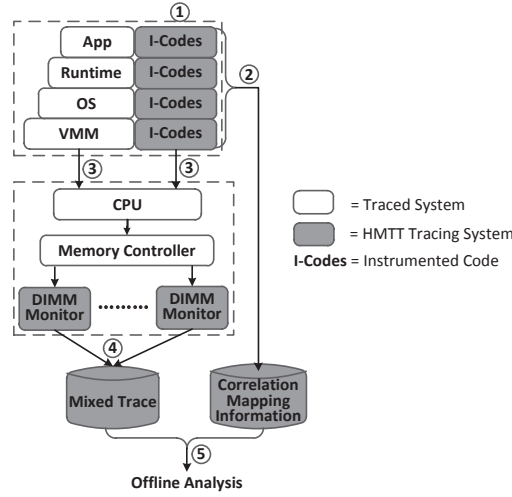


Fig. 7. Framework of the HMTT tracing system. This contains five procedures: ① instrumenting the target program manually or automatically to generate I-Codes and correlation-mapping information (②); ③ generating memory references; ④ using hardware-snooping devices to collect and dump the mixed trace to storage; and ⑤ replaying the trace for offline analysis.

Static code annotation can be performed at three levels: the source (assembly) level, the object-module level, and the executable (binary) level. If the source code of the target program is available, source-level annotation is the best choice, because the task of relocating the code and data of the annotated program can be handled by the compiler. For instance, we can directly modify the memory management codes of the OS to detect the page table update events and collect page table information, as described in Section 7.

Performing annotation at the object-module level implies that the original objects are replaced directly with new objects. Instructions are inserted into the new objects. For example, with the support of the *LD_PRELOAD* environment parameter, we can substitute the Pthread library and overlay the functions in the library. The object-level annotation method is exploited in our lock-profiling technique, introduced in Section 8.

Code annotation at the executable level is difficult to implement because executable files are often stripped of symbol-table information. A significant amount of analysis may be required to properly relocate code and data after tracing-generation instructions have been added to a program. For example, in profiling function calls, we may need to understand the ELF format of the Linux binary and find all the addresses of function entries before inserting our profiling codes.

5.2. Top-level framework

At the top level, the HMTT tracing system consists mainly of five procedures for DRAM access trace tracking and replaying. Figure 7 shows the system framework and the five procedures.

As shown in Fig. 7, the first step for mixed-trace collection is instrumenting the target program (i.e., the application, library, OS, and VMM) with instrumented codes (I-Codes) by hand and by means of scripts or compilers (① in the figure). The I-Codes inserted at the points where high-level events occur will generate specific memory references (③) and correlation-mapping information (②). The correlation-mapping information contains two parts: the information produced by HLE2M, such as the mapping

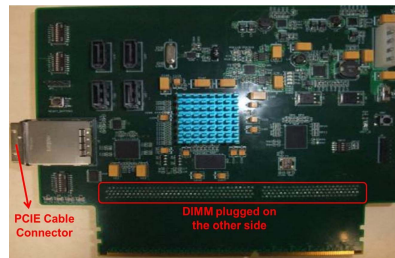


Fig. 8. The HMTT tracing system. This is plugged into a DIMM slot of the traced machine. The main memory of the traced system is plugged into the DIMM slot integrated into the HMTT system.

between the long semantic variables and short identities, and detailed information about high-level events, for example the page table of the OS.

For the hardware components, the HMTT system uses several hardware DIMM-monitoring boards plugged into the DIMM slots of the machine to be traced. The main memory modules of the traced system are plugged into DIMM slots integrated on the hardware monitoring boards (see Fig. 8). These boards monitor all memory commands via the DIMM slots (④). An on-board FPGA converts the commands into DRAM access traces in the format $\langle \text{timestamp}, \text{read/write}, \text{address} \rangle$. Each hardware monitor board generates a trace separately and sends the trace to its corresponding receiver via a Gigabit Ethernet or PCI-Express interface (④ in Fig. 7). With synchronized timestamps, the separate traces can be merged into a total mixed trace.

If necessary, a large DRAM access trace can be trimmed using the trace reduction method described above. Then, by correlating the mixed DRAM access trace collected by the hardware and the mapping information obtained by the software, we can construct the high-level-event execution flow, the access pattern, and other observations in the offline analysis (⑤). For example, page table information can be used to reconstruct the physical-to-virtual mapping relationship. Consequently, all of the memory address trace can be translated into virtual addresses for each process.

5.3. Putting it all together

Figure 8 illustrates the hardware board of HMTT. Currently, the HMTT system supports DDR-200 MHz, DDR2-400 MHz, and DDR3-800 MHz. In order to keep a high signal quality, we suggest the use of memory systems with a frequency lower than 800 MHz. We have also developed several toolkits for trace dumping and analysis. The HMTT system has been successfully tested on various Linux and Windows platforms. However, each HMTT card can monitor only one channel. So, multiple HMTT cards are required to monitor multiple channels. A high-speed interface connecting multiple HMTT cards can be added in order to synchronize the cards using timestamp information and allow them to share the PCI-E cable adapters. Then, the DRAM access traces collected by the separate HMTT cards can be merged together according to their timestamps.

In the future, we intend to optimize the signal integrity of the hardware to support DDR3-1333 and even higher frequencies. The data bus will also be monitored to assist in software debugging, fault diagnosis, and security testing.

Although there exist similar hardware tools, such as products from FuturePlus [FuturePlus 2012] and LeCroy [Teledyne LeCroy 2013], which have better compatibility and a higher supported DDR frequency, they are mainly designed for hardware debugging. They lack the ability to collect high-level semantic information and have a limited tracing buffer. Unlike HMTT, whose sustained tracing time is limited only by the external storage, their sustained tracing times are several seconds or less.

6. ANALYSIS OF HMTT TRACING SYSTEM

Having introduced the design and implementation of our HMTT system, we will now analyze the overhead and limitations of the system in this section.

6.1. Overhead

The hardware board of the HMTT system allows us to collect undistorted DRAM access traces without interference. Only the software component, including the high-level-event detection procedure and the high-level-event-encoding mechanism (HLE2M), can result in additional overhead. In the event detection procedure, both object-module-level and executable-level code annotation may add several extra function calls. Depending on the semantic information required for different events, the procedure for gathering semantic information may also require simple memory accesses. In addition, HLE2M issues one or several additional uncacheable memory requests to the memory system for each event. Overall, the overhead corresponding to each high-level event is small, only several memory requests and function call operations.

The effects of interference incurred in our proposed approach depend mainly on two factors, namely the memory characteristics of the profiled application and the frequency of high-level events. For memory-intensive applications, high-level events will trigger many synchronous memory requests, which will exacerbate memory contention in memory system. For CPU-intensive applications, a large number of high-level events and corresponding synchronous memory requests may obviously increase runtime spending on these memory accesses. However, if the high-level events do not happen frequently, the runtime overhead may be negligible, no matter whether the application is memory-intensive or CPU-intensive. Also, not all events are important enough to affect the performance of applications. Thus, we can filter out some of the unimportant events to reduce interference.

Taking functional profiling as an example, we can distinguish DRAM access addresses from other functions by monitoring the function call and function return events. The semantic information for function profiling contains a function identity and an operation flag (i.e., “function call” or “return”). Table I illustrates runtime overhead caused by HMTT for several applications included in the SPEC CPU 2006 benchmark. MCPI indicates the memory characteristics of applications, where high value implies for memory intensive and low value implies for CPU intensive. Additional memory requests mainly come from the synchronous requests triggered by function calls, and therefore indicate the number of function call events. As shown in the table, although 401.bzip2 is CPU-intensive, its frequent function calls lead to a runtime overhead of about 2.4 times. 429.mcf and 462.libq are both memory-intensive applications, but their runtime slowdowns are different because of different numbers of function call events. 429.mcf suffers from 170% more runtime because large amounts of memory interference are incurred in our approach. However, only 20% extra runtime is added to 462.libq, because there are just 2% additional memory requests.

Table I. Runtime overhead for functional profiling incurred with HMTT

Application	MCPI (memory cycles per instruction)	Ratio of additional memory requests (function call events)	Normalized runtime overhead
401.bzip2	0.49	729%	2.4
429.mcf	6.45	64%	2.7
462.libq	9.10	2%	1.2
470.lbm	3.57	1%	1.0

6.2. Limitations

There are several limitations of our proposed approach. First, HMTT cannot monitor full memory traces, only off-chip DRAM access traces. Many types of memory access behavior shown by applications may be filtered by caches. Second, a portion of memory, which is transparent to the OS, is required to be reserved as our hardware's configuration space. The size of the configuration space determines the available encoding bits included in each memory address. Third, the semantic information implied for each high-level event is restricted. Since our approach encodes all semantic information into memory addresses, the available space included in each memory address determines the length of the semantic information. Although multiple addresses can be used to represent one high-level event, the encoding efficiency is reduced. An alternative way is storing large amounts of semantic information in a local memory buffer and then mark the existence of this information using synchronization memory requests in the DRAM access trace. Fourth, the sequence of memory addresses collected by HMTT may be inconsistent with that issued by processors, because of the out-of-order execution model for modern processors and the memory requests scheduling mechanism for memory controllers. This phenomenon may lead to inaccurate event behaviors (e.g., reordered events) in post-analysis, due to the fact that HMTT makes use of memory addresses to represent the occurrence of events. However, memory requests produced by HMTT don't depend on each other, and consequently are executed in order under most situations. Meanwhile, the latency of memory requests incurred by memory scheduling is limited because of small memory scheduling buffers. Therefore, the inaccuracy problem of our profiling results can happen, but only with small probability. Finally, there exists a trade-off between the number of monitored high-level events and memory interference with the profiled application. More events to be monitored imply more interference with applications.

6.3. Usage scenarios

The HMTT system can be utilized in many situations, from basic DRAM access trace collection to various types of analysis of semantic-information. By leveraging the HMTT system, we have produced several high-level-event-profiling tools, for purposes such as object profiling, function profiling, and lock profiling.

Generally, HMTT can be used in two different ways. The first way is to obtain annotated DRAM access profiles. Memory addresses are extended with rich semantic information such as the pid and object identity. These extended DRAM access traces can be used to drive memory simulators, guide optimization by distinguishing different memory access patterns for different objects, and so on. Object profiling and function profiling belong to this type of use. The other is as a low-overhead output-data-streaming framework. Because HMTT enables one to collect and restore a large memory trace with small interference with the profiled application, we can consider HMTT as an efficient data storage medium that operates by transforming high-level information into uncacheable memory requests. Lock profiling is an example of such a use.

In this paper, we present two typical usage scenarios in the following sections to illustrate how to adapt HMTT to specific purposes, namely object profiling (Section 7) and lock profiling (Section 8).

7. SCENARIO 1: OBJECT-RELATIVE MEMORY PROFILING

For application programs, physical memory addresses do not provide an intuitive way to reveal the behavior of programs. Objects containing a group of data items, for example an array or a structure, are the data units commonly allocated by programmers.

Object-level behavior is therefore more straightforward to programmers than memory addresses are. Therefore, correlating memory addresses with their corresponding object information is important. However, efficient object-relative memory-profiling tools do not exist. Although simulators and dynamic instrumentation tools such as Pin can achieve this kind of profiling, their large overhead may disturb the execution of the application being profiled. Hardware performance counters can attribute the occurrence of certain hardware events, such as TLB misses, to specific objects using sampling and interrupt mechanisms, but they still have a trade-off between the accuracy of object behavior and runtime interference.

Our HMTT system supports the collection of full DRAM access traces and correlated object information, including the process identity and object identity, with little interference and runtime overhead. This object-relative memory profiling enables us to distinguish various objects from memory addresses. Different types of behavior for different objects can easily be discovered, for example page table walks caused by TLB misses, and access patterns. The annotated memory addresses can also be utilized to drive memory simulators in order to optimize aspects of the memory system such as the memory scheduling policy.

7.1. Design issues

To retrieve object information from physical memory addresses, at least two levels of mapping are required. Figure 2 shows an example of these mappings. The first level of mapping is from physical memory addresses to virtual memory addresses. This can be achieved by dumping the page table of the OS for each process. In the offline analysis, we can reconstruct a reverse page table and look up the corresponding virtual address for each physical address in the DRAM access trace. However, the page table may be updated when physical pages are released or reallocated, and the mapping between physical addresses and virtual addresses is not unique. We need to mark these updates in the HMTT DRAM access trace. Thus, the first semantic gap relates to how to synchronize the page table update events with the DRAM access trace.

The second level of mapping is from virtual addresses to objects. In a program, there are three different types of objects: temporary objects whose space is automatically allocated in the stack, static objects whose space is automatically managed in the data segment, and dynamic objects whose space is manually allocated in the heap. The temporary objects are ignored because they have little effects on performance. For each static object, we can get the virtual entry address and its size from the symbol table of the execution file, for example the ELF format file in a Linux system. For dynamic objects, their virtual entry addresses and information about their sizes can easily be obtained from the object management functions, such as the commonly used *malloc()*. Hence, we need to monitor only these functions. At this point, we can construct the mapping from objects to their virtual address ranges. Nevertheless, just like the updating of the page table, the mapping between virtual addresses and objects may change, accompanied by the *malloc()* and *free()* functions. Hence, the second semantic gap relates to how to synchronize the object creation or release events in the DRAM access trace.

So far, with the aid of the OS and the object runtime, by collecting page table and object management information, respectively, the semantic gap between the high-level object access events and the DRAM access trace has been transformed into a gap between page table update events plus object management events and the DRAM access trace. To fix this gap, we apply our high-level-event-encoding mechanism, HLE2M. The fundamental semantic information implied by a high-level event here is the type of event, i.e., a page table update or an object update. The basic encoding policy is shown in Fig. 9(a). There also exist many alternative types of semantic information, for exam-

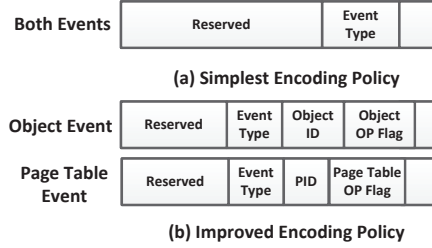


Fig. 9. Use of HLE2M for object behavior profiling. (a) Available basic encoding policy; (b) improved encoding policy, containing more semantic information.

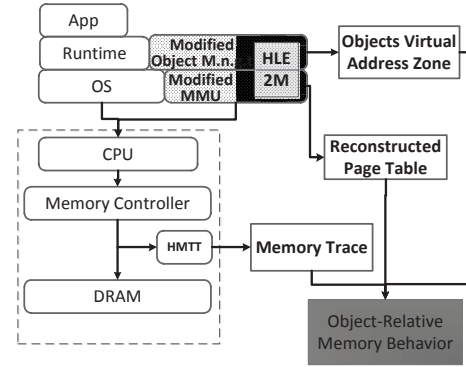


Fig. 10. Framework for object-relative memory profiling.

ple the object identity and object operation type for object management events. We can also encode this information into the memory address space, as in the example shown in Fig. 9(b).

7.2. Implementation

Figure 10 shows our framework for object-relative memory profiling [Chen et al. 2012]. This framework uses our HMTT system and thus comprises hardware and software components. In terms of hardware, the HMTT card monitors the memory access requests to the DRAM system and dumps all physical memory address traces. In terms of software, to detect the updating of the page table, we directly modify the source code of the page table operation functions in the memory management unit of the OS. But, for the object management functions, we utilize the *LD_PRELOAD* environment variable in the Linux system to overlay the *malloc()* and *free()* functions, and insert our codes in the overlaid functions.

The inserted codes accomplish two tasks. The first task is to collect the detailed mapping information related to the high-level events. In the memory management module of the OS, we collect the page table information and update the information corresponding to the page table update events. This information is used to reconstruct the *reverse page table*, which translates physical addresses to virtual addresses. For object management, we record the entry address and the size of each dynamic object, which constitute the *virtual address zone* of the object. In the offline analysis, by combining the DRAM access trace with the reverse page table, we can distinguish the DRAM access traces for each process and extract per-process virtual address traces. Next, by combining the virtual address trace of each process with the virtual address zone of its object, we can finally obtain the object-relative memory behavior. The second task of the inserted codes is to allow HLE2M to notify the hardware of the occurrence of page table updating or object operations. The memory addresses issued by HLE2M and captured by the snooping hardware are called the *synchronous tags trace*.

It should be noted that in our approach, objects are not limited to the user space. In fact, with careful annotation, memory accesses to kernel objects can also be identified.

7.3. Experiments

We have done experiments on a system with two 2.00 GHz Intel Xeon E5504 processors. Each E5504 processor has four physical cores. The total capacity of the physical memory is 4 GB, with one dual-ranked DDR3-800 RDIMM, and the peak memory

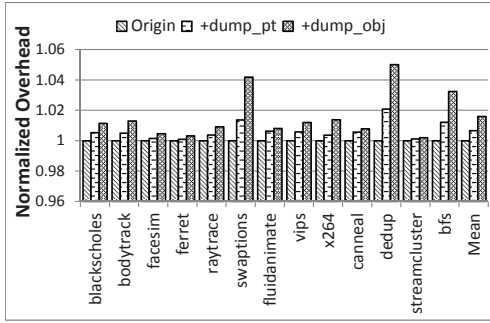


Fig. 11. Runtime overhead of object-relative memory profiling. “+dump_pt” and “+dump_obj” stand for objects in *Canneal*, and the normalized performance the overhead of dumping page table update events, speedup by using huge pages for the *elements* object, and dumping both page table update and object management events, respectively.

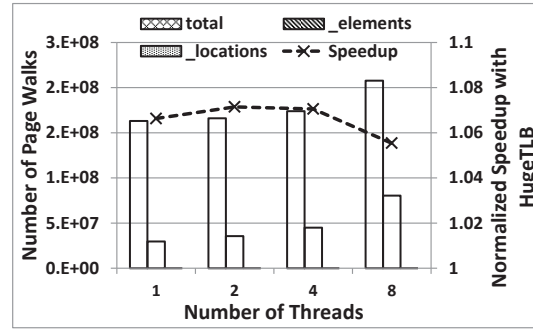


Fig. 12. The number of page memory walks for main objects in *Canneal*, and the normalized performance the overhead of dumping page table update events, speedup by using huge pages for the *elements* object, and dumping both page table update and object management events, respectively.

bandwidth is 6.4 GB/s. We reserved 0.25 GB of memory space as HMTT’s configuration space and page table buffer, and thus the actual memory available is 3.75 GB. The operating system is 64-bit CentOS 5.3 with Linux kernel 2.6.32.18.

First of all, we will evaluate the overall runtime overhead of the object-relative memory profiling performed with the HMTT system. Figure 11 shows the runtime overhead for some applications in the widely used PARSEC benchmark, running with eight threads. Since the main memory accesses of many multithread programs focus on a few large objects [Chen et al. 2012], we only need to monitor the relative memory allocation information for those objects. In our experiments, we chose to monitor only objects that were larger than 4 KB. The results showed that the average overhead of dumping the page table was about 0.66% and the average overhead of dumping object management information was about 1.60%. The largest overhead was 5.00%, for dedup, which contains more than 1.2 million dynamic memory allocations and deallocations during its execution. However, the overhead of an object-level memory profiler with dynamic instrumentation (Pin) was nearly 30 to 80 times, even with 10% sampling [Lu et al. 2009]. Overall, we can conclude that the object-profiling overhead incurred with the HMTT system can be made relatively small or negligible for most applications by filtering out small objects.

Here, we give two examples to demonstrate the usage of object profiling. In the first example, we make use of object behaviors to optimize the TLB miss problem. Since page memory walks caused by TLB miss consume much time and stall processors’ pipeline, effective TLB behaviors are important for high performance. The object-relative memory profiling can pick up the page walk memory requests and correlate them with different objects. Figure 12 describes the number of page memory walks for main objects of *Canneal* in PARSEC benchmark. The *elements* object results in the largest number of page walks compared to other objects, nearly 40% of total page walks under 8 threads. Based on this observation, we adopt huge pages and huge TLB entries to hold the *elements* object. As shown in Fig. 12, the normalized performance speedup can reach up to 7%.

The goal of second example is distinguishing access patterns for different objects. We performed object-relative memory profiling on a serial version of SpMV (Sparse Matrix-Vector multiplication), a program to multiply a sparse matrix (in CSR format) by a dense vector. This program executes the computation $y = ax * xhost$. The nonzero elements in the matrix are stored in the array *ax*, and the vector is stored in the array *xhost*. The *ax* object is accessed consecutively, whereas the *xhost* object may be accessed

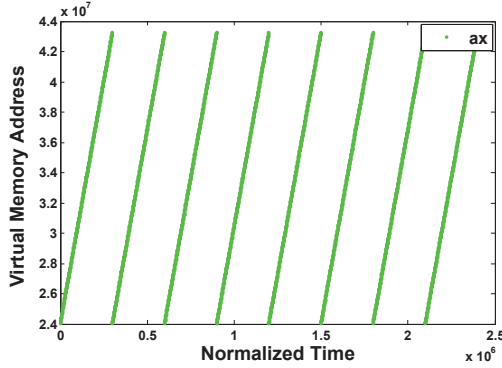


Fig. 13. The regular access pattern of the *ax* object in SpMV.

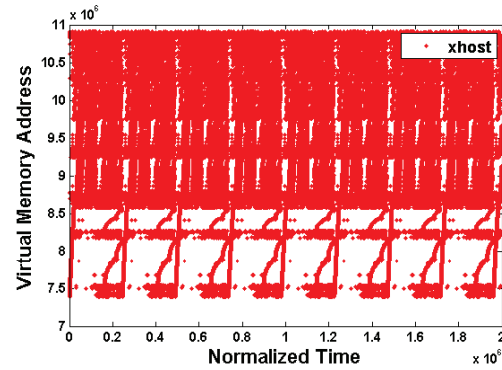


Fig. 14. The irregular access pattern of the *xhost* object in SpMV.

randomly. However, we cannot find any regular access pattern from the mixed DRAM access traces obviously. With the object-relative memory profiling, we can distinguish the access traces of the *ax* object and the *xhost* object from other objects. Figures 13 and 14 illustrate the regular access pattern of the *ax* object and the irregular access pattern of the *xhost* object, respectively. The virtual addresses of *ax* object and *xhost* object shown in the y-axis of above figures begin with the same preface, and therefore we only display their offsets. Based on the access patterns of different objects, we can do a large amount of optimization focused on different objects. For example, we can use page-coloring techniques to allocate unbalanced cache resources to different objects and isolate their cache accesses, in order to preserve the locality of the *ax* object.

8. SCENARIO 2: LOCK PROFILING IN MULTITHREADED APPLICATIONS

Since multithreaded applications use locks, such as the mutex locks of the POSIX thread library, to safeguard the consistency of shared data, lock contention has long been considered as a key impediment to the scalability of applications. Therefore, profiling lock information and diagnosing lock contention are still of great interest.

Generally, a lock-profiling tool operates in two steps. Firstly, it uses either instrumentation or a performance counter to capture runtime lock information such as thread id, operation type, and time information. Secondly, it records the profiling data for offline analysis. To the best of our knowledge, almost all of the current tools store profiling data in local memory buffers or on disk. Basically, a profiling tool itself should not significantly affect the execution of its target program. However, writing a large amount of data into memory will cause additional cache pollution and extra pressure on memory, which may significantly perturb the runtime behavior of the targeted program, thereby resulting in distorted profiling, especially for memory-sensitive applications. Thus, we have proposed a hardware-assisted lock-profiling mechanism, named HaLock [Huang et al. 2012], to reduce the memory interference that occurs in lock profiling.

8.1. Design issues

There are at least two different characteristics of lock operation events that make them unlike the above object and function events. Firstly, lock operations are more sensitive to cache and memory interference. For example, assume that there are two threads potentially competing for a lock: if one thread is delayed by cache or memory interference, it will require the lock later and consequently fail to hold the lock. In this situation, the lock behavior may be quite different from that in the original execution.

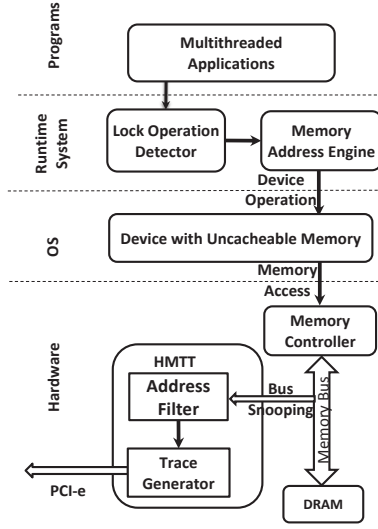


Fig. 15. The framework of HaLock.

Rsvd #1 (6 bits)	TID (10 bits)	Lock ID (9 bits)	Rsvd #2 (1 bit)	Flag (3 bits)	Rsvd #3 (3 bits)
---------------------	------------------	---------------------	--------------------	------------------	---------------------

Fig. 16. High-level-event-encoding policy for lock events.

Although the HMTT tracing system converts all the high-level events into memory requests, all the memory requests that it produces are uncacheable, and thereby do not cause cache interference. In order to alleviate the problem of memory interference, HMTT stores all the memory traces in another machine via a PCI-E cable.

Secondly, the semantic information implied by each lock event is more abundant than that implied by the object and function events illustrated in the above sections. For each lock operation event, at least three pieces of information are required: the lock address, thread id, and lock operation type (such as lock or unlock). All of this information must be encoded into the memory address space simultaneously. Generally, lock addresses and thread ids are 64-bit variables. Using HLE2M, illustrated in Fig. 6, these variables are transformed into short identities. For current multithreaded applications, the optimum number of threads cannot exceed about a thousand, 1024 for example. Hence, we can use a 10-bit identity to substitute for the 64-bit thread id. In addition, a map between the thread id and the 10-bit identity is maintained.

8.2. Implementation

HaLock makes use of HMTT to study the lock behavior of multithreaded programs. Figure 15 depicts the framework of HaLock. Lock operation detection is implemented simply by overlaying the Pthread library, which is transparent to applications and is suitable for applications without source code, such as databases. We instrument routines that could potentially cause lock contention, namely *pthread_mutex_{lock, trylock, unlock}*, in the overlaid Pthread library. To override a routine in a dynamically linked program, we use the library *preloading* parameter indicated by the *LD_PRELOAD* environment variable in Linux. When the target program calls one of the overlaid routines, the instrumented version of the routine takes over the execution. The overlaid routine first gathers the current thread id and lock address, and then determines the flag for each lock type.

Figure 16 shows one typical address-coding format in our experiments. If we use a 1024-entry hash table for the thread id and a 512-entry hash table for the lock id, their corresponding hash indices have 10 and 9 bits, respectively. The Flag attribute represents the type of operation, such as lock, trylock, or unlock. There are three reserved fields. The length of Rsvd #1 depends on the size of HaLock's region; for example, 64 MB indicates that the high 6 bits are fixed and the lowest 26 bits are available for HaLock. The Rsvd #3 attribute depends on the memory bus width, and 3 bits means an 8-byte memory transfer unit. The existence of Rsvd #2 is determined by the memory address mapping. In our experimental platform, this bit identifies the memory channel number, as illustrated in Fig. 16. Since one HMTT card can monitor only one memory channel, the channel bit in the memory address must be set to the channel that HMTT is plugged into. The field of Rsvd #2 is determined by the memory address mappings, which vary greatly between different platforms.

There are five main steps in the lock-profiling process. (1) HaLock detects lock operations using the *LD_PRELOAD* environment parameter and tracks necessary information such as thread id, lock address, and operation type in the runtime system. (2) HaLock encodes semantic information for each lock operation into a specific uncacheable memory address by means of a memory address engine using HLE2M, and triggers an access to it. (3) HMTT is configured to capture only memory address signals generated by the memory address engine. A complete trace is constructed by combining the memory addresses captured by HMTT with their global clock times. (4) HaLock leverages HMTT to record the above traces by sending them to another machine via a PCI-E cable. In this way, HaLock supports the recording of a large number of traces without utilizing local CPU and memory resources. Thus, it can provide provable, strong guarantees: namely, it eliminates interference with running programs, no matter how large the traces are. (5) Using off-line analysis, HaLock can display the lock contention distribution of different locks among all the threads.

8.3. Experiments

Our experiments were conducted on Intel Xeon E5504 processors with 4 GB DDR3-800 memory. As the bandwidth of the PARSEC benchmark is not high, we used only one DIMM for two sockets. We reserved 64 MB of memory for HaLock, which cost only 1.5% of total memory and did not affect program execution. We used a selection of benchmarks from the PARSEC benchmark. We compared HaLock with two software-based mechanisms, called RDTSC-Lock and LiMiT-Lock [Demme and Sethumadhavan 2011], which store profiling data into memory and on disk. Whereas HaLock exploits HMTT's hardware clock to provide timestamps for the lock operations, RDTSC-Lock used the *rdtsc* instruction and LiMiT-Lock used the LiMiT tool to acquire timestamps in our experiments.

Figure 17 shows the memory interference and overall behavior of the different lock-profiling mechanisms for several different multithreaded programs with eight threads. In general, HaLock causes less perturbation than the two other mechanisms for all programs tested. HaLock yields only about 1% extra memory requests and 1.2% extra cache misses, whereas RDTSC-Lock results in more than 4.4% extra memory requests and 3.9% extra cache misses on average. For each program, the increased number of memory requests and the changes in cache miss ratio incurred with HaLock are less than those incurred with RDTSC-Lock. In RDTSC-Lock, large amounts of profiled data are first buffered in memory, which causes additional cache eviction operations and thus extra memory requests. When the memory buffer is full, these data are dumped onto disk, and this procedure consumes both memory and buffer cache. However, HaLock issues only a one-byte uncacheable memory request for each lock operation during the whole recording phase. Thus, the numbers of memory requests and

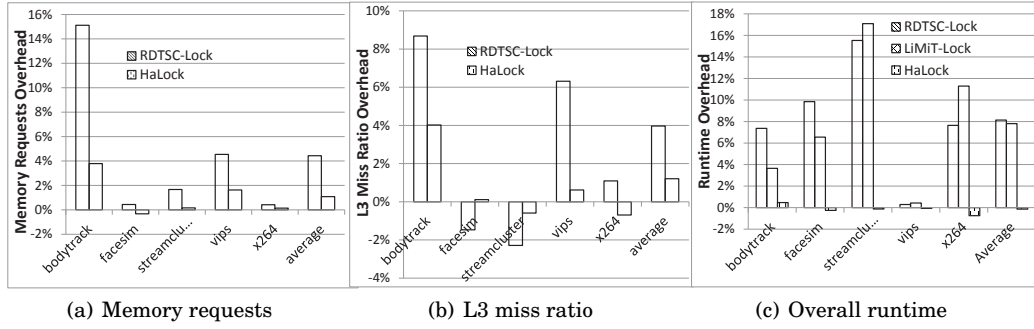


Fig. 17. Comparison of memory interference for different lock-profiling mechanisms when multiple PARSEC benchmarks are run using eight threads. As LiMiT cannot monitor off-chip events itself, and conflicts with TopMC, the results of LiMiT-Lock are not shown in (a) and (b).

L3 cache miss ratios shown in Figs. 17(a) and (b) are lower than for RDTSC-Lock. By studying the memory interference incurred with RDTSC-Lock and HaLock, we could obtain information about its side effects in terms of runtime, as shown in Fig. 17(c). On average, the runtime overheads incurred with RDTSC-Lock and LiMiT-Lock are 8.1% and 7.8%, respectively, but HaLock has only 0.1% runtime overhead. These results demonstrate that RDTSC-Lock and LiMiT-Lock indeed seriously alter the execution of programs compared with HaLock.

In order to demonstrate the importance of the effect of memory interference on program execution, we used HaLock, RDTSC-Lock, and LiMiT-Lock to collect profiling data and compared the results in terms of the execution time related to lock operations. Figure 18 presents an overview of a breakdown of the execution time by synchronization region for all programs tested. The free time is the total number of cycles in which the threads are not related to any lock operation; the lock and unlock times are the numbers of cycles spent in *pthread_mutex_lock* and *pthread_mutex_unlock*, respectively, for all threads; and the lock hold time is defined as the sum of the numbers of cycles for which each thread holds for each lock. Traces that had a very large or negative number of cycles were placed in the “unknown” region. All the time regions shown in Figure 18 are normalized to the total execution cycle for each thread. We observe that the lock behaviors collected by HaLock, RDTSC-Lock, and LiMiT-Lock are substantially different. Firstly, the proportions of the above regions profiled by these mechanisms vary, and the amount of variation is determined by the memory interference. As shown in Fig. 17(a), bodytrack and vips suffer from the most serious memory interference, and thus their lock behavior shows large differences between the various profiling methods. Taking bodytrack as an example, the free time is 58.1% of the total number of cycles according to RDTSC-Lock, only 24.6% according to LiMiT-Lock, but nearly 81% according to HaLock. Secondly, the unlock times of all programs measured by HaLock are not negligible, whereas the corresponding times obtained with LiMiT-Lock and RDTSC-Lock are very small. Since an unlock operation requires invoking system calls to awake those threads which are waiting on the lock, and hence traps into interrupts, the unlock time should not be as small as the values shown by LiMiT-Lock and RDTSC-Lock.

Although all current profiling tools inevitably cause memory interference in the target program, we have shown above that HaLock causes less memory interference than do the current software-based mechanisms. Thus, we can conclude that the current mechanisms have nonnegligible distortion and inaccuracy problems, but that HaLock can provide more accurate lock behavior than other current mechanisms can.

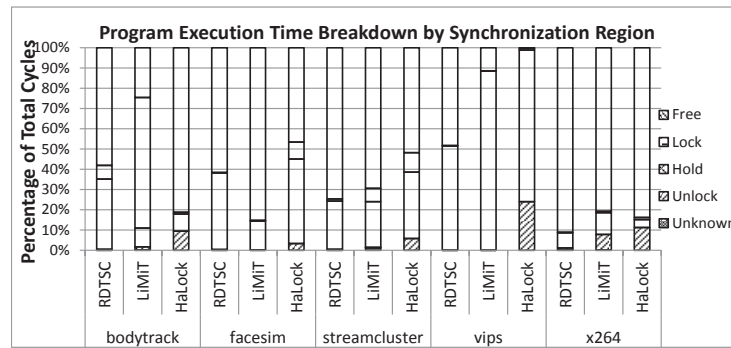


Fig. 18. Comparison of lock behaviors collected by different mechanisms for PARSEC benchmarks running with eight threads.

9. RELATED WORK

There are several areas of effort related to DRAM access trace monitoring: software simulators, binary instrumentation, hardware counters, hardware monitors, and hardware emulators.

- *Software simulators.* Most researches of studying memory performance and system power are based on simulators. These utilize cycle-accurate simulators to generate DRAM access trace and then feed trace to trace driven memory simulators (e.g. DRAMSim [Wang et al. 2005], MEMSim [Rajamani 2000]). SimpleScalar [Austin et al. 2002] is a popular user-level simulator, but it cannot run an operating system for the analysis of full-system behavior. Several full-system simulators (such as SimOS [Rosenblum et al. 1995], Simics [Magnusson et al. 2002], GEM5 [Binkert et al. 2011], BOCHS [Lawton 1996], and QEMU [Bellard 2005]), which can boot commercial operating systems, are commonly used in research studies of OS-intensive applications. However, software simulators usually have speed and scalability limitations. As computer architectures become more and more sophisticated, more detailed simulation models are needed, which may lead to a slowdown of 1000–10000 times [Barroso 1999]. Moreover, simulations of complex multicore and multithreaded applications may suffer from inaccuracies and could lead to misleading conclusions [Nanda et al. 2000].
- *Binary instrumentation.* Many binary instrumentation tools (e.g., OProfile [Levon and Elie 2004], ATOM [Srivastava and Eustace 1994], DyninstAPI [Buck and Hollingsworth 2000], Pin [Luk et al. 2005], and Valgrind [Nethercote and Seward 2007]) are popularly utilized to profile applications. These are able to obtain virtual access traces of applications even without source code. Nevertheless, few of them can provide full-system DRAM access traces, because instrumenting kernels is very tricky. PinOS [Bungale and Luk 2007] is an extension of the Pin dynamic instrumentation framework for full-system instrumentation. However, PinOS can only run on the IA-32 architecture in uniprocessor mode. Moreover, the binary-instrumentation method usually slows down the execution of the target programs, leading to time distortion and memory access interference.
- *Hardware counters.* Hardware counters are able to provide accurate event statistics (e.g., cache misses and TLB misses). Itanium2 [Intel Corporation 2004] is even able to collect traces via sampling. The approach based on hardware counters is fast and has low overhead, but it cannot provide complete, detailed memory reference traces.
- *Hardware monitors.* Various hardware monitors, which can be divided into two classes, are able to monitor DRAM access traces online. One class consists of pure trace

collectors, and the other of online cache emulators. BACH [Grimsrud et al. 1993] is a trace collector. It utilizes a logic analyzer to interface with the host system and to buffer the collected traces. When the buffer is full, the host system is halted by an interrupt and the trace is moved out. Then, the host system continues to execute the program. BACH is able to collect traces from long workload runs. However, the halting mechanism may alter the original behavior of the program. There are several similar tools, such as those of Alexander et al. [1986] and Fuentes [1993], Monster [Nagle et al. 1992], DASH [Torrellas et al. 1992] and SuperTrace Probe [Green Hills Software 2013]. The hardware-based online cache emulation tools (such as MemoriES [Nanda et al. 2000], RACFCS [Youn et al. 1997], and ACE [Hong et al. 2006]) are very fast and have low distortion and no slowdown. Logic analyzers are also powerful tools for capturing signals (including DRAM signals) and can be very useful for hardware testing and debugging. However, these hardware monitors have several disadvantages. (1) Except for BACH, they are not able to dump full mass traces but can only produce short traces, owing to their small local memory. (2) There is a semantic-gap problem for hardware monitors because they can only collect physical addresses. (3) They depend on proprietary interfaces: for example, MemoriES relies on the IBM 6xx bus, and BACH, PHASER, ACE, HACS, etc. use a logic analyzer, which is quite expensive. RACFCS uses a latch board that is connected directly to the output pins of specified CPUs. So, they all have poor portability.

- *Hardware emulators.* Several hardware emulators are complete FPGA-based systems that have been constructed to accelerate research; they utilize a number of FPGAs to construct uniprocessor/multiprocessor platforms. For example, RPM [Barroso et al. 1995] emulates the entire target system within its emulator hardware. Intel proposed an FPGA-based Pentium system [Lu et al. 2007] that was an original Socket-7-based desktop processor system with typical hardware peripherals and could run modern operating systems. RAMP [Wawrzynek et al. 2007] was also a new scheme for architecture research. Although these hardware emulators do not currently produce DRAM access traces, they are capable of tracking full-system traces. But they can only emulate a simplified, slow system with relatively fast I/O, which enlarges the CPU-to-memory and memory-to-disk gaps that may be bottlenecks in real systems.

10. CONCLUSIONS

In this paper, we have proposed a hybrid hardware/software mechanism that is able to collect memory reference traces as well as semantic information. Based on this mechanism, we have designed and implemented a prototype system called HMTT (Hybrid Memory Trace Tool), which uses a DIMM-snooping mechanism to snoop on the memory bus and a software-controlled high-level-event-encoding mechanism. Comprehensive validation has shown that HMTT is a feasible and convincing system for monitoring DRAM access traces. Several profiling tools derived from HMTT have shown that it is also effective and has wide applicability. Thus, the HMTT system demonstrates that a hybrid tracing mechanism can leverage the advantages of both hardware (e.g., no distortion or pollution) and software (e.g., flexibility and more information) to perform various types of low-overhead profiling. Moreover, this hybrid mechanism can be used by other tracing systems.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their helpful comments. We would like to thank Kai Li (Princeton University), Xiaodong Zhang (Ohio State University), Howard David (Intel Corporation) and Lixin Zhang, Peiheng Zhang, Guangming Tan, Dan Tang, Yan Zhu, and Jinyong Zhang from Institute of Computing Technology, Chinese Academy of Sciences for their helpful discussions and suggestions.

REFERENCES

- C. Alexander, W. Keshlear, F. Cooper, and F. Briggs. 1986. Cache memory performance in a UNIX environment. *Computer Architecture News* 14 (1986), 14–70.
- Todd Austin, Eric Larson, and Dan Ernst. 2002. SimpleScalar: An infrastructure for computer system modeling. *Computer* 35, 2 (2002), 59–67.
- Yungang Bao, Mingyu Chen, Yuan Ruan, Li Liu, Jianping Fan, Qingbo Yuan, Bo Song, and Jianwei Xu. 2008. HMTT: A platform independent full-system memory trace monitoring system. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 36. ACM, 229–240.
- Luiz André Barroso. 1999. *Design and Evaluation of Architectures for Commercial Applications*. Technical Report. Western Research Laboratory.
- Luiz André Barroso, Sasan Iman, Jaeheon Jeong, Koray Öner, Michel Dubois, and Krishnan Ramamurthy. 1995. RPM: A rapid prototyping engine for multiprocessor systems. *Computer* 28, 2 (1995), 26–34.
- Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Usenix Annual Technical Conference*.
- Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai 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, 72–81.
- Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, and others. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.
- Bryan Buck and Jeffrey K. Hollingsworth. 2000. An API for runtime code patching. *International Journal of High Performance Computing Applications* 14, 4 (2000), 317–329.
- Prashanth P. Bungale and Chi-Keung Luk. 2007. PinOS: A programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE'07)*. 137–147.
- Licheng Chen, Zehan Cui, Yungang Bao, Mingyu Chen, Yongbing Huang, and Guangming Tan. 2012. A lightweight hybrid hardware/software approach for object-relative memory profiling. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'12)*. IEEE, 46–57.
- John Demme and Simha Sethumadhavan. 2011. Rapid identification of architectural bottlenecks via precise event counting. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*. IEEE, 353–364.
- Mathieu Desnoyers and Michel R. Dagenais. 2006. The lttng tracer: A low impact performance and behavior monitor for GNU/Linux. In *OLS (Ottawa Linux Symposium)*. Citeseer, 209–224.
- C. Fuentes. 1993. *Hardware Support for Operating Systems*. Technical Report. University of Michigan.
- FuturePlus. 2012. FuturePlus Systems. <http://www.futureplus.com/>. (2012).
- Green Hills Software. 2013. SuperTrace Probe. <http://www.ghs.com/products/supertraceprobe.html>. (2013).
- K. Grimsrud, J. Archibald, M. Ripley, K. Flanagan, and B. Nelson. 1993. BACH: A hardware monitor for tracing microprocessor-based systems. *Microprocessors and Microsystems* 17, 6 (1993).
- Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism* 7, 4 (2005), 1–28.
- Jumnit Hong, Eriko Nurvitadhi, and Shih-Lien L. Lu. 2006. Design, implementation, and verification of active cache emulator (ACE). In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays (FPGA '06)*. ACM, New York, NY, USA, 63–72.
- Yongbing Huang, Zehan Cui, Licheng Chen, Wenli Zhang, Yungang Bao, and Mingyu Chen. 2012. HaLock: Hardware-assisted lock contention detection in multithreaded applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. ACM, 253–262.
- Intel Corporation 2004. *Intel Itanium2 Processor Reference Manual*. Intel Corporation.
- Intel Corporation 2012. *Intel 64 and IA-32 Architectures Software Developers Manual, vol. 3A & 3B*. Intel Corporation.
- JEDEC Solid State Technology Association. 2004. Double Data Rate (DDR) SDRAM Specification. JEDEC Solid State Technology Association. (2004).
- Kevin P. Lawton. 1996. Bochs: A portable PC emulator for Unix/x. *Linux Journal* 1996, 29es (1996), 7.
- John Levon and Philippe Elie. 2004. Oprofile: A system profiler for Linux. (2004).
- Qingda Lu, Jiang Lin, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. 2009. Soft-olp: Improving hardware cache performance through software-controlled object-level partitioning. In *Pro-*

- ceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*. IEEE, 246–257.
- Shih-Lien L. Lu, Peter Yiannacouras, Rolf Kassa, Michael Konow, and Taeweon Suh. 2007. An FPGA-based Pentium® in a complete desktop system. In *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays (FPGA '07)*. ACM, New York, NY, USA, 53–59.
- Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices* 40, 6 (2005), 190–200.
- Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A full system simulation platform. *IEEE Computer* (Feb. 2002).
- D. Nagle, R. Uhlig, and T. Mudge. 1992. *Monster: A Tool for Analyzing the Interaction between Operating Systems and Computer Architectures*. Technical Report.
- Ashwini Nanda, Kwok-Ken Mak, Krishnan Sugarvanam, Ramendra K. Sahoo, Vijayaraghavan Soundararajan, and T. Basil Smith. 2000. Memories: A programmable, real-time hardware emulation tool for multiprocessor server design. In *ASPLOS-IX: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, USA, 37–48.
- Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices* 42, 6 (2007), 89–100.
- K. Rajamani. 2000. *Memsim Users' Guide*. Technical Report. IBM.
- Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. 1995. Complete computer system simulation: The SimOS approach. *IEEE Parallel and Distributed Technology: Systems and Applications* 3, 4 (Winter 1995), 34–43.
- Amitabh Srivastava and Alan Eustace. 1994. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 196–205.
- Teledyne LeCroy. 2013. Protocol analyzers. <http://teledynelecroy.com/protocolanalyzer/>. (2013).
- TopMC. 2011. TopMC: Performance Counter Monitor Tool. <http://asg.ict.ac.cn/projects/topmc/>. (2011).
- Josep Torrellas, Anoop Gupta, and John Hennessy. 1992. *Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System*. Vol. 27. ACM.
- Richard A. Uhlig and Trevor N. Mudge. 1997. Trace-driven memory simulation: A survey. *Comput. Surveys* 29, 2 (1997), 128–170.
- David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Katie Baynes, Aamer Jaleel, and Bruce Jacob. 2005. DRAMsim: A memory system simulator. *Computer Architecture News* 33, 4 (Sept. 2005), 20–24.
- John Wawrzynek, David Patterson, Mark Oskin, Shin-Lien Lu, Christoforos Kozyrakis, James C. Hoe, Derek Chiou, and Krste Asanovic. 2007. RAMP: Research accelerator for multiple processors. *Micro, IEEE* 27, 2 (2007), 46–57.
- Win A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: Implications of the obvious. *Computer Architecture News* 23, 1 (March 1995), 20–24.
- Hyung-Min Youn, Gi-Ho Park, Kil-Whan Lee, Tack-Don Han, Shin-Dug Kim, and Sung-Bong Yang. 1997. Reconfigurable address collector and flying cache simulator. In *Proceedings of High Performance Computing Asia*.