`

HaLock: Hardware-Assisted Lock Contention Detection in Multithreaded Applications

Yongbing Huang[†‡], Zehan Cui[†‡], Licheng Chen[†‡], Wenli Zhang[†], Yungang Bao[†], Mingyu Chen[†]
[†]State Key Laboratory of Computer Architecture, Institute of Computing Technology,
Chinese Academy of Sciences, Beijing, China
[‡]Graduate University of Chinese Academy of Sciences, Beijing, China
{huangyongbing, cuizehan, chenlicheng, zhangwl, baoyg, cmy}@ict.ac.cn

ABSTRACT

Multithreaded programming relies on locks to ensure the consistency of shared data. Lock contention is the main reason of low parallel efficiency and poor scalability of multithreaded programs. Lock profiling is the primary approach to detect lock contention. Prior lock profiling tools are able to track lock behaviors but directly store profiling data into local memory regardless of the memory interference on targeted programs.

In this paper, we find that the memory interference is non-trivial and can significantly affect programs' execution as thread number increases. To address this problem, we propose a hardware assisted lock profiling mechanism (HaLock) which leverages a specific hardware memory tracing tool (HMTT) to record large amount of profiling data with negligible overhead and impact on even large scale multithreaded programs.

Experimental results show that HaLock incurs only about 14.8% additional L3 cache misses and 34.3% extra memory requests for a lock-intensive workload (bodytrack of PARSEC benchmark) with 512 threads, while the previous state of the art low-overhead approach causes 25.9% additional L3 cache misses and 73.8% additional memory requests. Compared with HaLock's profiling data, we find that the lock behaviors obtained by the state of art lock profiling tools have substantial distortions, resulting in non-negligible inaccuracy problems.

Categories and Subject Descriptors

C.4 **[Performance of systems]**: Measurement techniques, performance attributes; D.1.3 **[Programming techniques]**: Concurrent Programming--Parallel programming

General Terms

Design, Measurement, Performance

PACT'12, September 19-23, 2012, Minneapolis, Minnesota, USA.

Keywords

Multithreading, Lock Contention, Memory Interference, Performance Analysis, HMTT

1. INTRODUCTION

Multithreaded programming is the dominated programming model on shared memory multicores platform. Recent research [23] argues that coherence protocol which is regarded as the primary reason of poor scalability of shared memory architecture, is able to scale well up to even 512-core. Therefore, multithreaded programming will probably still be the most popular programming model for forthcoming many-core machines.

Since multithreaded applications use locks, such as mutex locks of POSIX thread [13] library, to guard the consistency of shared data, lock contention has long been considered as a key impediment to applications' scalability. For instance, Johnson et al. [22] investigate four popular open source databases (Shore, BerkeleyDB, MySQL, and PostgreSQL) on a multicore system, and identify lock contention as the major bottleneck for scaling up to 32 threads. Multithreaded programming relying on locks puts programmers in a dilemma that coarse-grain locks significantly degrade program's performance and scalability due to lock contention but fine-grain locks are error-prone, thus requiring substantial development efforts. Therefore, profiling lock information and diagnosing lock contention are still of great interest. By identifying lock contention hotspots and shortening or removing them, Johnson et al. [22] achieved 2-4 times performance improvement.

Generally, a typical look profiling tool consists of two steps: Firstly, it adopts either instrumentation or performance counter to capture runtime lock information such as thread id, operation types and time information [5, 12, 16]. Some tools may further support advanced functionalities, such as thread filters or source file filters [28]. Obliviously, collecting more information means paying higher cost. Generally, instrumentation has higher cost but is more flexible, while performance counter costs less but collects less information. Secondly, it needs to record the profiling data for offline analysis. To our best knowledge, almost all of the current tools preserve profiling data in local memory buffers or disks [2, 4, 5, 12, 15, 16, 28]. Basically, a profiling tool itself should not significantly affect its target program's execution.

`

However, writing large amount of data into memory will cause additional cache pollution and extra memory pressures which may significantly perturb the targeted program's runtime behaviors thereby resulting in distorted profiling, especially for memory sensitive applications.

Sampling is an alternative method to reduce the profiling overhead and the runtime interference. However, it is likely to miss small critical sections [11, 16] which are also necessary to be optimized for scalability [19]. But Boyd-Wickizer et al. [11] show that even for very short mutex-lock based critical sections can cause dramatic collapse in the performance of real workloads, when the cores are increasing.

Table 1. Summary of Lock Contention Profiling System

System	Info. Acquisition*	Collection Method	Data Record
HPCToolKit [28]	P.C. & Inst.	Sampling	Memory & Disk
CodeAnalyst[1]	P.C. & Inst.	Sampling	Memory & Disk
VTune (Thread Profiler) [3, 4]	P.C. & Inst.	Sampling	Memory & Disk
Jucprofiler [5]	Inst.	Continuous	Memory & Disk
Lockmeter [12]	Inst.	Continuous	Memory & Disk
LockStat [6]	Inst.	Continuous	Memory
LiMiT [16]	P.C. & Inst.	Continuous	Memory & Disk
HaLock	Inst.	Continuous	HMTT
* P.C is short for Performance Counter, Inst. for Instrumentation			

Table 1 illustrates several prevalent lock profiling tools. All previous tools primarily record profiling data in memory. These approaches work well for small scale of threads but will cause unacceptable memory interferences for dozens of threads which will be common in a many-core system. Our experimental results show that for a current low-overhead lock profiling approach, the additional cache misses increase substantially from 8.7% (8 threads) to 25.9% (512 threads) and the extra memory requests also increase from 15.1% (8 threads) to 73.9% (512 threads).

Since the cache and memory resources per core are decreasing and becoming critical resources with the core number increasing, it makes significant sense to provide a lock detection technique with little memory interference.

In this paper, we propose a hardware assisted lock profiling mechanism (HaLock) to reduce the memory interference of lock profiling.

Basic Idea: As illustrated in Table 1, we gather lock information such as thread id, lock address and lock type via instrumentation. The distinct advantage of HaLock is that it leverages a specific hardware called HMTT [2, 9] to record profiling data. We encode runtime lock information (i.e., thread id, lock address and lock operation type etc.) into specific memory addresses which are reserved for HaLock and are uncacheable to avoid cache pollution. When one lock operation is captured, instead of storing the profiling data in memory as previous tools do, HaLock just issues a one-byte memory read request to an encoded memory address. This specific memory request is monitored by HMTT which is a hardware-software hybrid memory trace toolkit for real systems. HMTT packs each memory address with a timestamp (HMTT's clock cycles) to form one lock operation trace, and finally transmits the trace to another machine via a PCI-e cable.

We implement HaLock in Linux 2.6.32 Kernel and evaluate HaLock on an 8-core SMP system in comparison to the state of the art profiling tool LiMiT which temporarily buffers data in memory and dumps the data into disks afterwards. Experimental results show that for 8 threads HaLock incurs less than 1.2% extra memory requests for multithreaded programs while the overhead of LiMiT is nearly 3.9% on average. Meanwhile, HaLock exhibits much better scalability than LiMiT, and scales well up to 512 threads. Specifically, HaLock incurs only 4.0% additional L3 cache miss and 3.8% extra memory requests for even an extreme lock-intensive workload (bodytrack of PARSEC benchmark [10] exhibiting 4K lock–acquisition operations per second per thread) with 8 threads (14.8% additional L3 cache miss and 34.3% extra memory requests for 512 threads). Furthermore, we observe that the lock behavior collected by HaLock and LiMiT are substantially different. Since HaLock has much less memory interference than LiMiT, the large difference means that current lock profiling tools have non-negligible distortions and inaccuracy problems. Since performance tuning and debugging are essential software development steps, it is worthwhile to call for attention on how to provide efficient and accurate profiling tools. Thus, we argue that hardware-enabled lock profiling mechanism is necessary for forthcoming many-core architecture.

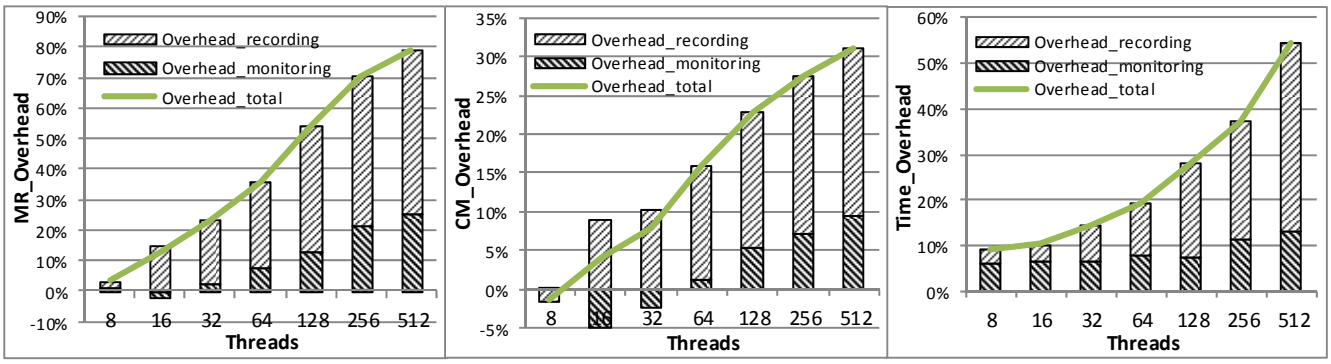
In summary, we make the following contributions:

1) The cache and memory interference of recording profiling data into memory adopted by current lock profiling tools are quantitatively analyzed. And we demonstrate that recording data into memory results in serious memory interference.

2) We propose and implement a hardware assisted lock profiling tool called HaLock which is able to significantly reduce the memory interference and scales well up to hundreds of threads.

3) We reveal that the start of the art lock profiling tools have non-negligible distortions and inaccuracy problems, which implies the necessity of hardware-enabled lock profiling mechanisms for many-core architectures. We also argue that it is worthwhile to pay more attention on how to provide efficient and accurate profiling tools for many-core architectures.

The rest of this paper is organized as follows: Section 2 introduces the background of lock contention detection and shows our motivations for this paper. We describe our mechanism in Section 3. Experiment environment and evaluation results are



(a) **(b)** **(c)**

Figure 1. Profiling overhead of bodytrack program by recording traces through memory.

described and analyzed in Section 4. We summarize the related work in Section 5, and finally conclude this paper in Section 6.

2. BACKGROUND AND MOTIVATIONS

2.1 Lock Profiling Overhead

As illustrated in Table 1, many efforts have already been made on improving profiling accuracy and reducing profiling overhead. Specifically, the overhead of lock profiling depends on three factors as illustrated in the following formula:

$$Overhead_{total} = Overhead_{per\_lock\_trace} * Lock\_Trace\_Number_{per\_thread} * Thread\_Number$$

For $Overhead_{per\_lock\_trace}$, it consists of two phases: 1) monitoring lock information and 2) recording profiling data. Previous researches in Table 1 mainly focus on the first phase, e.g., exploiting performance counter. Nevertheless, for the second phase, almost all of the current tools adopt a straightforward way to record profiling data, i.e., writing data directly into local memory buffers or disks. Sampling techniques is another optimization direction which tries to reduce $Lock\_Trace\_Number_{per\_thread}$, but it is tricky and probably leads to the inaccuracy problem.

Most lock profiling tools claim that they have little overhead (<10%). It is true for small ***Thread_Number***. For example, our experimental results show that the memory overhead of a profiling tool similar to LiMiT is only about 3.9% for 8 threads. But as ***Thread_Number*** keeps increasing in this multicore/many-core era, both $Overhead_{per\_lock\_trace}$ and $Lock\_Trace\_Number_{per\_thread}$ are also influenced, deserving re-evaluation. Intuitively, the total trace number would sharply increase, probably resulting in more memory overhead.

2.2 Challenges of Large Scale Threads

To investigate the impact of the increasing ***Thread_Number*** on lock profiling overhead, we measure and analyze the lock behavior of bodytrack of PARSEC benchmark on an Intel Xeon 8-core machine (details in Section 4). We use an in-house LiMiT-like tool to collect performance metrics (such as cache miss ratio and memory requests) via performance counters, and use the commonly used RDTSC instruction to obtain lock operations' timestamps. Our lock profiling approach also consists of two phase. For the monitoring phase, we instrument bodytrack's binary codes to collect lock information; for the recording phase, we first write profiling data into a memory buffer (64MB) and then dump them into disk when the buffer is full. To facilitate analyzing, we also divide $Overhead_{total}$ into two portions according to the two phases mentioned in Section 2.1, i.e., $Overhead_{monitoring}$ and $Overhead_{recording}$. Thus,

$$Overhead_{total} = Overhead_{monitoring} + Overhead_{recording}.$$

We run the bodytrack benchmark from 8 threads to 512 threads. Figure 1 illustrates the profiling overhead changes in terms of memory requests (***MR_Overhead***), L3 cache miss rate (***CM_Overhead***) and execution time (***Time_Overhead***). According to Figure 1(a), $MR\_Overhead_{total}$ increases substantially from 3.3% (8 threads) to 79.2% (512 threads). This shows that the profiling tools have serious memory interference, which may potentially alter programs' execution behaviors and lead to inaccurate lock behaviors consequently. Figure 1(b) demonstrates the same phenomenon from the view of L3 cache miss.

Generally, execution time is the most straightforward metric of programs' execution behaviors. If we observe $Time\_Overhead_{total}$ shown in Figure 1(c), we can see that the execution time increases sharply with the thread number, and thus can roughly conclude that the execution behaviors of the profiled programs are indeed disturbed. Naturally, the accuracy of current lock profiling tools is doubtful.

Moreover, compared to $MR\_Overhead_{total}$, $MR\_Overhead_{monitoring}$ increases slowly, just from 0.8% (8 threads) to 25.1% (512 threads), which implies that the recording phase becomes dominant of memory interferences with ***Thread_Number*** increasing. $CM\_Overhead_{recording}$ and $Time\_Overhead_{recording}$ also confirm the situation. Thus, we should put more emphasize on the recording phase.

`

Among current recording methods, bypassing cache [26] seems to be able to eliminate cache pollution but does no help with memory interference. What's more, directly dumping traces into files is too slow that the traces may be dropped when locks are highly contended. Thus, currently there are no appropriate solutions to solve the dilemma of the recording phase.

2.3 Our Goal

According to above analysis, current recording mechanisms suffer from serious cache pollution and memory interferences. In contemporary multicore systems, L3 cache and memory system become critical resources, and then memory interference would result in unpredictable execution behavior variations and inaccuracy profiling results. There have been numerous studies focusing on reducing cache interference and memory interference [17, 18, 25]. It is unacceptable for a profiling tool to cause such serious interferences to its target programs. Thus, we propose a hardware assisted lock profiling mechanism (HaLock) which leverages a specific hardware memory tracing tool (HMTT) to record large amount of profiling data with negligible overhead.

3. HARDWARE ASSISTED LOCK PROFILING MECHANISM (HaLock)

3.1 Overview

3.1.1 Introduction of HMTT

HMTT [2, 9] is a platform independent full system memory trace monitoring system. The system adopts a DIMM-snooping mechanism which uses hardware boards plugged in DIMM slots to track virtual memory reference traces of full systems. Thus it is able to track complete, detail and undistorted traces without altering original programs' behavior. In order to dump full mass memory traces, HMTT utilizes multiple Gigabit Ethernets or PCI-e cables and RAIDs to send and receive mass memory traces respectively, which does not perturb host system's execution at all even when traces are generated at high speeds. That is why we choose HMMT to collect lock traces in our lock profiling approach. Meanwhile, HMTT now supports both DDR2 and DDR3 UDIMM and RDIMM interfaces, and thus has good portability.

3.1.2 Principle

The hardware assisted lock profiling mechanism (HaLock) makes use of HMTT to study lock behaviors of multithreaded programs. As indicated by Section 2, purely software methods of lock profiling may lead to heavy memory interferences with running programs, and consequently derive distorted and inaccuracy lock behaviors. In order to alleviate memory interference, HaLock leverages a hybrid mechanism which combines software-based lock detector and hardware-based trace collector. Figure 2 depicts the framework of HaLock. There are five main steps:

1) HaLock detects lock operations and tracks necessary information such as thread id, lock address, and operation type (Section 3.2.1) in runtime system. Instrumentation is an efficient

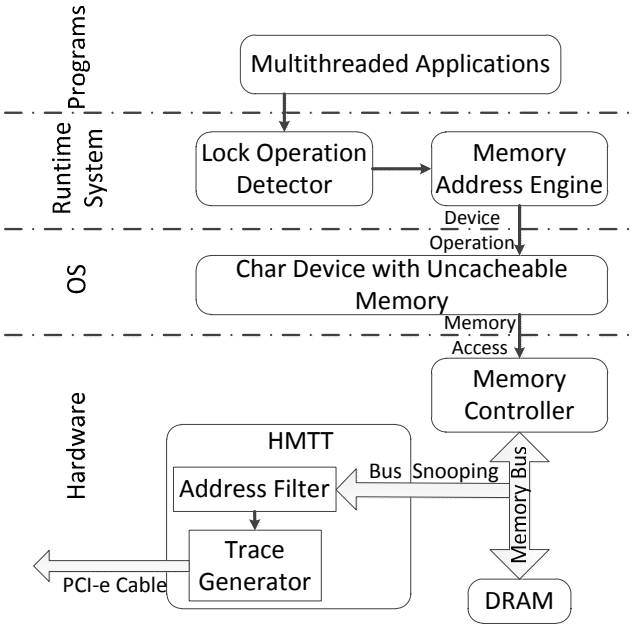


Figure 2. Framework of HaLock.

method to achieve this by modifying the source codes of profiled programs or overlaying dynamic libraries, such as Pthread library for mutex lock. Though time is indispensable for analyzing lock behavior, HaLock resorts to HMTT's hardware clock time.

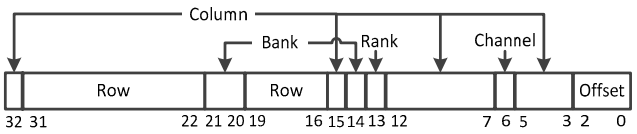
2) After gathering the required data for each lock operation, HaLock encodes them into a specific memory address by memory address engine (Section 3.2.2). Each part of the memory address has its specific meaning and we can pick out thread id, lock address and operation type from it. Then, memory address engine triggers an access to the memory address by issuing a one-byte memory read instruction and hopes it to be captured by HMTT immediately. In order to eliminate negative cache effects and protect these memory addresses from other processors, a virtual device owing uncacheable physical memory is registered into operating system.

3) HMTT is configured to only capture memory address signals generated by the memory address engine. Once HMTT captures a signal on memory bus, it filters the address and only records predefined memory address. Then, a complete trace is constructed by combining the memory address and HMTT's global clock time.

4) HaLock leverages HMTT to record above traces by sending them to another machine via a PCI-e cable or Ethernet. Under this way, HaLock supports to record even a large amount of traces without utilizing local CPU and memory resource. Thus, it achieves provable, strong guarantees, namely, it eliminates the interference to 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.

`



(a) Memory address mapping for dual channels memory

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

(b) Address coding format corresponding to (a)

Figure 3. Memory address mapping and its corresponding address coding format of HaLock.

3.1.3 Advantages

HaLock provides *high accuracy* and *strong scalability* (see Section 4). Firstly, while HaLock stores a large amount of lock behavior traces through HMTT, it introduces few extra memory requests unlike previous methods which store traces through local memory. Therefore, HaLock may degrade little programs' performance. Secondly, with the help of global hardware clock provided by HMTT, HaLock does not need to care about clock variants among different cores. In software based tools, if a thread is migrated to another processor during lock operations, the clock difference between two processors may mess up the lock behavior of this thread, such as negative acquisition time and impossible huge hold time. HaLock has two advantages: saving time to read performance counters and eliminating clock inconsistence among multiple processors.

Finally and the most important one, since HaLock has less memory interference, the lock behaviors reflected by it are more convincing and helpful for programs debugging and optimization.

3.2 Runtime System Extensions

3.2.1 Lock Operation Detector

Source and binary instrumentation are both common ways to obtain runtime contexts such as lock operation behaviors. Since this paper focuses on mutex lock, we implement lock operation detector simply by overlaying Pthread library which is transparent to applications and suitable for those applications without source codes such as databases.

We instrument routines that could potentially cause lock contention: pthread_mutex_{lock, trylock, unlock} in the overlaid Pthread library. To override a routine in a dynamically linked program, we use library preloading parameter indicated by LD_PRELOAD environment variable on 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 current thread id and lock address, and then determines the flag for each kind of lock type.

In order to put emphasis on the effects of recording profiling data but not obtaining them, the experiments in Section 4 just obtain basic profiling data by runtime systems. If more profiling data are collected for each lock operation such as calling stacks, data recording phase may cause more serious interference by software based mechanisms and thus HaLock may gain more benefits.

Though this paper aims at analyzing mutex lock, other locks such as spinlock contained in Pthread library can also be monitored by the same technique. As to other lock types beyond Pthread library such as user defined lock, source code instrumentation is an alternative and feasible way.

3.2.2 Memory Address Engine

Since there exists a semantic gap between software trace data and memory bus signals, memory address engine in Figure 2 solves this challenge.

Memory address engine transforms profiling data for a single lock operation into a specific memory address. This procedure consists of two parts: (1) how to distinguish these specific memory addresses from other processes' memory addresses, and trigger them to be immediately captured by HMTT; and (2) how to encode lock addresses, thread identities and lock types in memory address space.

Device with uncacheable memory: We reserve a small amount of continuous physical memory for HaLock by setting "mem" parameter in boot loader. In our experiments, the value of "mem" is 64MB less than the actual total physical memory size. Then, the remained 64MB memory cannot be allocated to any processes by operating system directly, but are accessible by operating system. We utilize above remained memory as the HaLock's specific memory and HMTT can easily distinguish them from other memory addresses.

However, physical memory addresses cannot be directly accessed without corresponding linear addresses mapped in Linux's page tables. The virtual device shown in Figure 2 fills the address transformation gap. All the remained memories are registered into the virtual char device. Insides the char device's operations such as mmap and read, we construct the address mapping from the user's linear addresses to remained physical addresses making use of kernel function "remap_pfn_range". As a result, the memory address engine achieves to trigger access to remained physical address by operating the char device.

In general, memory requests are satisfied by caches firstly, and only memory requests missing caches are sent to memory bus. In order to get rid of the negative of caches and capture the requests by HMTT immediately, all the remained memory addresses are marked as uncacheable in the char device. This feature also eliminates polluting running programs' caches and reduces the memory interference.

Memory address space formats: HaLock's memory region is partitioned into three regions which delegate thread id, lock address and lock type respectively. Obviously, thread id and lock address, which are 32-bit and 64-bit respectively, cannot encode into memory address space directly. To address this issue, we propose a hash table based data transformation approach. In this approach, HaLock's runtime system creates two hash tables for

`

threads and locks respectively. When new locks or threads are detected, the runtime system inserts them into their corresponding hash tables and uses their hash indices to represent them.

Figure 3(b) shows one typical address coding format in our experiments. If we use a 1024-entry hash table for thread id and 512-entry hash table for lock id, their corresponding hash indices are 10 bits and 9 bits respectively. The "Flag" attribute delegates the operation, such as lock, trylock, and unlock. There are three reserved fields. The length of "Rsvd #1" depends on the size of HaLock's region, e.g., 64MB indicating that the high 6 bits are fixed and the lowest 26 bits are available for HaLock; The "Rsvd #3" attribute depends on memory bus width and 3 bits means 8-Byte memory transfer unit. The existence of "Rsvd #2" is determined by the memory address mapping. In our experiment platform, the 6th bit of memory address identifies the memory channel number as illustrated in Figure 3(a). Since one HMTT card can only monitor one memory channel, the channel bit in the memory address must be set to the channel that HMTT is plugged in. The similar reservation fields of "Rsvd #2" are determined by the memory address mappings which vary largely on different platforms.

3.3 Limitation

Since HMTT is essentially a memory trace toolkit for recording profiling data, it is not surprising that there are limitations affiliated to HaLock.

Delayed timestamp: We propose to exploit global hardware clock of HMTT as lock operations' occurrence timestamps. But this clock has some delays caused by memory controllers which usually buffer memory requests for scheduling in order to improve memory efficiency or power [17, 18, 25, 27]. Since the delays are constrained to at most dozens of cycles by the state-of-art scheduling algorithms, the timestamp is still trustworthy.

Memory address space formats: In HaLock's profiling traces, runtime information are encoded into memory addresses. However, available bits in memory address are limited to the size of reserved physical memory and memory address mapping. For instance, in Figure 3, there are totally 22 bits which can be shared by threads offset, lock offset and lock types. Generally, these bits can be enough for most applications. But there might be some extreme cases that more threads and locks are exploited in a program. One solution is to reserve more physical memory for HaLock at the expense of less available memory for targeted programs.

4. EXPERIMENTS

4.1 Methodology

Platform: Our experiments are conducted on Intel Nehalem processors. Table 2 summarizes the architectural parameters. As the bandwidth of PARSEC benchmark is not high, we use only one DIMM for two sockets. Meanwhile, we reserve 64MB memory for HaLock which costs only 1.5% of total memory and does not affect programs execution.

Table 2: Experimental Platform Configuration

Processor	Intel Xeon E5504 (2.0GHz, 32KB L1I and L1D Cache, 256KB L2 Cache, 4MB L3 Cache)
# of processors	2
#of cores	8
Channel	2
DIMM	2
Memory Type	DDR3-800
Memory Size	4GB (64MB Reserved)

Benchmarks: We use a selection of benchmarks from PARSEC Benchmarks. In this work, we only focus on mutex lock of Pthread library. Thus, we choose the programs which on average have many lock operations from PARSEC benchmark. Table 3 lists the benchmarks we use in the following experiments.

Table 3: Benchmark summary

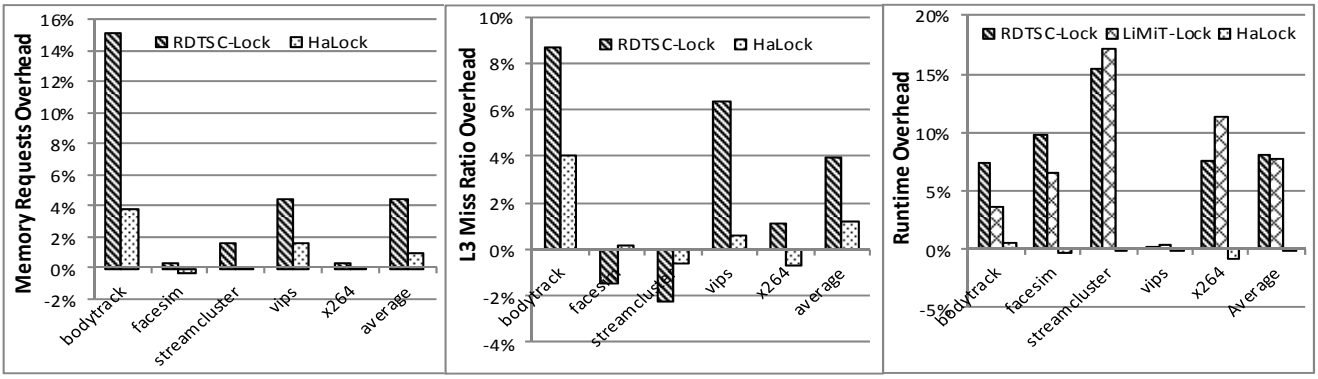
Benchmark	Parallelization Model	Lock Operations per thread per second
bodytrack	data-parallel	4800
facesim	data-parallel	670
streamcluster	data-parallel	1950
vips	data-parallel	55
x264	pipeline	537

Tools: In our experiments, we use hardware performance counter events [8] to evaluate programs' behaviors, such as L3 miss ratio and the number of total memory requests. We utilize TopMC [7] which provides low-overhead access to performance counters including uncore events (i.e. off-chip events) in Nehalem architecture.

Comparisons: In order to show that HaLock can reduce cache misses and memory interferences in recording phase, we compare HaLock with software based mechanisms storing profiling data into memory and disk. In the monitoring phase of all mechanisms, thread id, lock address and lock operation type are collected for each lock operation. Since HaLock exploits HMTT's hardware clock as lock operations' timestamp, we try to use low-overhead, and precise time collecting approaches to ensure fairness in compared mechanisms. Usually, programs use **rdtsc** instruction to obtain time information. LiMiT [16] provides a precise, lightweight mechanism to access on-chip performance counters including each core's accurate clock information. Thus, we compare HaLock with two software based mechanisms which utilize **rdtsc** instruction (RDTSC-Lock) and LiMiT (LiMiT-Lock) to acquire timestamp respectively in our experiments.

4.2 Memory Interferences of Different Lock Profiling Mechanisms

In this subsection, we will evaluate the memory interferences of HaLock, RDTSC-Lock and LiMiT-Lock using the aforementioned programs. For all overhead results in the experiments, we use original programs without any profiling tools as the baselines.



(a) Memory Requests **(b) L3 Miss Ratio** **(c) Overall Runtime**

Figure 4. Memory interference comparison for different lock profiling mechanisms when running multiple PARSEC benchmarks using 8 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).

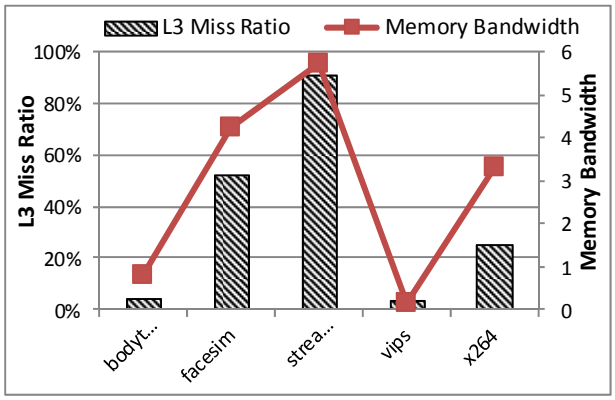


Figure 5. L3 cache miss and memory bandwidth of tested benchmarks.

4.2.1 Memory Interferences for Different Multithreaded Programs

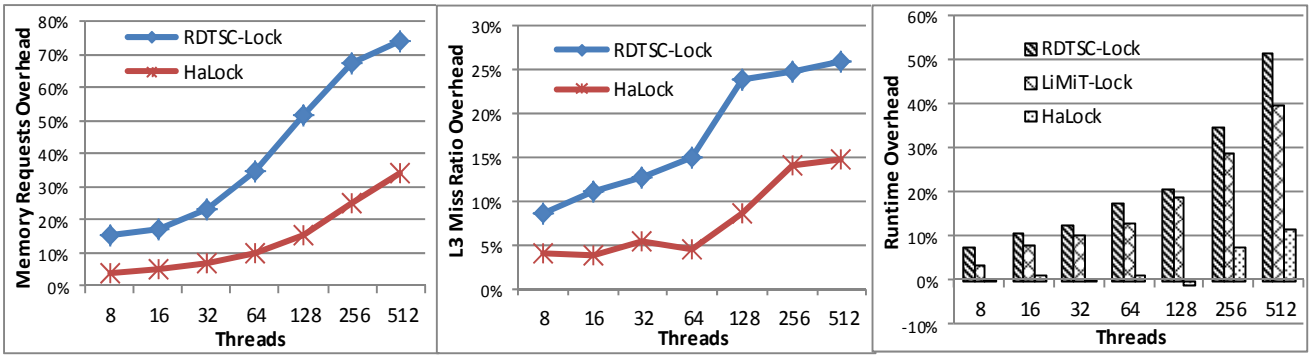
Figure 4 shows the memory interferences and overall behaviors of the different lock profiling mechanisms for different multithreaded programs with 8 threads. In general, HaLock incurs less perturbation than the two other mechanisms for all tested programs.

As shown in Figure 4, HaLock yields only about 1% extra memory requests and 1.2% extra cache misses while RDTSC-Lock results in more than 4.4% extra memory requests and 3.9% extra cache misses on average. For each program, the increased memory requests and changed cache miss ratio incurred by HaLock are less than that incurred by RDTSC-Lock. Since RDTSC-Lock and LiMiT-Lock both use low-overhead instructions (**rdtsc** and **rdpmc** respectively) to gather timestamps without incurring memory requests, the monitoring phase of RDTSC-Lock and LiMiT-Lock are the same as HaLock in terms of memory requests. Thus, the difference shown in Figure 4 attributes to the recording phase.

In the recording phase of RDTSC-Lock, large amounts of profiled data are firstly buffered in memory, which would cause additional cache eviction operations and thus extra memory requests. When the memory buffer are full, these data are dumped into disk and this procedure consumers both memory and buffer cache. However, HaLock only issues a one-byte uncacheable memory request for each lock operation in the whole recording phase. Thus, the memory requests and L3 cache miss ratios shown in Figure 4(a) and 4(b) are lower than RDTSC-Lock.

The memory interference behaviors vary largely for different programs as shown in Figure 4(a). On one hand, the interferences are positively correlated to the lock operation frequency. As the lock operation frequency increases, more memory requests are issued by each profiling tools. Bodytrack has the highest lock operation frequency which means that it has the most potential memory interference. However, the situation seems not suitable for streamcluster which has 1950 operations per second but only 1.67% extra memory requests even for RDTSC-Lock. Besides lock operation frequency, the internal behaviors, such as memory bandwidth and L3 cache miss, of tested programs determine the overall effects of interferences on the other hand. Figure 5 demonstrates the memory bandwidth and L3 cache miss ratio for each tested program. The memory bandwidth of streamcluster is nearly 89% of the total memory bandwidth of our experiment platform. Compared with streamcluster, the memory requests issued by RDTSC-Lock are not so obvious. That's why the memory requests overhead is low for streamcluster though RDTSC-Lock incurs lots of extra memory requests.

The above analysis can also illustrate the cache miss ratio overhead indicated in Figure 4(b). Interestingly, both increased and decreased cache misses exist. The cache miss of tested programs are constrained to three factors: originally cache miss, cache miss of memory requests issued by profiling tools and cache pollution proportion. Generally, the memory requests produced by RDTSC-Lock are sequential, and only one request may be cache miss ideally for each 8 requests. If the tested programs have poor locality and cache pollution is not serious, they may suffer from decreased cache misses, such as facesim and streamcluster, and vice versa.



(a) Memory Requests (b) L3 Miss Ratio (c) Overall Runtime

Figure 6. Memory interference comparison for different lock profiling mechanisms when the thread number increases up to 512.

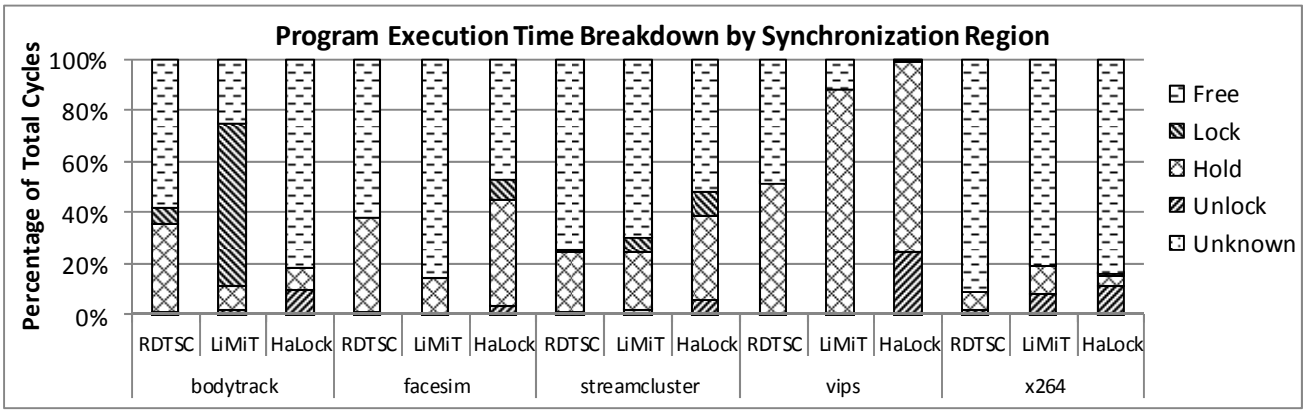


Figure 7. Comparison of lock behaviors collected by different mechanisms for PARSEC benchmarks running with 8 threads.

After studying the memory interference of RDTSC-Lock and HaLock, we can learn the side effects of these interferences in terms of runtime in Figure 4(c). On average, the runtime overheads incurred by RDTSC-Lock and LiMiT-Lock are 8.1% and 7.8% respectively, but HaLock only incurs 0.1% runtime overhead. The results demonstrate that RDTSC-Lock and LiMiT-Lock indeed seriously alter the programs' execution compared to HaLock. Especially for memory intensive programs such as streamcluster, the memory resources become so critical that additional memory interferences of profiling tools can severely change programs' normal execution, increasing up to 17.1% runtime for streamcluster.

Totally, compared with RDTSC-Lock and LiMiT-Lock, HaLock is able to reduce memory interference and just slightly affects programs' execution.

4.2.2 Scalability for Different Lock profiling Mechanisms

As the number of cores continually increases, scalability is significantly important for profiling tools in many-core era. Obviously, lock contention will be exacerbated when thread numbers are increasing in multithreaded programs, and memory interferences would be enlarged for any lock profiling mechanisms. We measure the scalability of each profiling mechanisms running bodytrack from 8 threads to 512 threads in Figure 6.

When bodytrack scales up to 512 threads, HaLock brings about 34.3% additional memory requests, less than half of memory overhead brought by RDTSC-Lock shown in Figure 6(a). The memory interference gap between HaLock and RDTSC-Lock is enlarged as the number of threads increases. Figure 6(b) indicates that L3 cache miss has the same tendency as memory requests. Thus, HaLock has good scalability than other mechanisms. From Figure 6(c), we can conclude that HaLock can actually reduce profiling perturbations on programs' execution behaviors even if thread number increases.

4.3 Lock Behaviors of Programs

In order to demonstrate the importance of memory interference on programs' execution, we use HaLock, RDTSC-Lock and LiMiT-Lock to collect profiling data and compare their results in terms of execution time related to lock operations. Figure 7 illustrates an overview of execution time breakdown by synchronization region for all tested programs. **Free** time is the total cycles when threads are not related to any lock operation; **Lock** and **Unlock** time is the cycles spent in pthread_mutex_lock and pthread_mutex_unlock for all threads respectively; Lock **Hold** time is defined as summation of cycles each thread holds for

`

each lock. We consider those error traces having huge or negative cycles as **Unknown** region. All the time regions shown in Figure 7 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 profiled region by these mechanisms are varied, and the memory interferences determine the gap size. As shown in Figure 4(a), bodytrack and vips suffer from the most serious memory interferences, and thus their lock behaviors are sharply differed. Take bodytrack as an example, free time is 58.1% of the total cycle obtained by RDTSC-Lock, only 24.6% by LiMiT-Lock, but nearly 81% by HaLock. Secondly, **Unlock** time of all programs measured by HaLock are not negligible while the corresponding time are trivial obtained by LiMiT-Lock and RDTSC-Lock. Since unlock operation requires invoking system calls to awake those threads waiting on the lock and hence traps into interrupt, unlock time should not be as small as shown by LiMiT-Lock and RDTSC-Lock.

Since all the current profiling tools inevitably bring about memory interferences to target programs, we have proved that HaLock has the least memory interferences than current software-based mechanisms in above subsections. Thus, we can conclude that current mechanisms have non-negligible distortions and inaccuracy problems. HaLock can actually provide the most accurate lock behaviors than other current mechanisms.

5. RELATED WORK

Dedicated hardware for memory trace: Various hardware monitors are able to monitor memory trace online. Besides HMTT utilized in this work, other hardware monitors such as BACH [20], SHRIMP [24], Alliant System [14], can also collect memory traces online. BACH utilizes a logic analyzer to interface with host system and buffer the collected traces. When the buffer is full, the host system is halted by an interrupt and the traces are moved out. However, this halting mechanism may alter original behaviors of programs. SHRIMP performance monitor is a hardware monitor with several novel features including multi-dimensional histograms, page tags, histogram categories, and a threshold interrupt mechanism. Torrellas et al. [29] present a similar hybrid hardware/software approach which could potentially be used for lock profiling. However their hardware monitor relied on MIPS buses which were proprietary, and the software implementations are totally different.

Performance tools: There are numerous studies on analyzing lock contention. Almost all of them focus on monitoring phase regardless of how to optimize data recording phase.

IBM's jucprofiler [5] analyzes Java's concurrent locks. It mainly identifies threads' contention time and waiting time caused by contending Java's concurrent locks. The collected information by jucprofiler is quite similar with our work. However, jucprofiler records above information in local memory and local disk. HPCToolkit [28] uses runtime information associated with locks to blame lock holders for the idleness of spinning threads. Thread Profiler [3] measures routines' effective parallelism and distinguishes between interaction effects such as cruise, impact and blocking time of each thread. Lockmeter [12] and LockStat [6] are tools for analyzing locks in a multiprocessor Linux kernel.

Others utilize hardware performance events to assist lock profiling. LiMiT [16] enables precise, lightweight interface to on-chip performance counter which allows precise reading of virtualized counters by one or two orders of magnitude faster than current access techniques. The synchronization characteristics of PARSEC benchmark detected by LiMiT are different from characteristics shown by others performance counter tools which have larger overheads. We compare LiMiT with our approach, and show that their synchronization characteristics are also varied a lot because of overheads in data recording phase. Hardware performance events are also exploited by other tools, such as Inter's VTune [4], AMD's CodeAnalyst [1], ProfileMe [15] and so on.

6. CONCLUSIONS

In this paper, we have studied the memory interferences incurred by current lock profiling tools storing profiling data into memory or disk. Then, we have proposed a hardware assisted lock profiling tool (HaLock) which is able to significantly reduce the memory interferences and scales well up to hundreds of threads by recording all the lock behavior traces through the hardware HMTT.

Experimental results show that HaLock incurs only 3.8% extra memory requests and 4% additional cache miss for even a lock-intensive workload with 8 threads, and has well scalability up to 512 threads. We also illustrated that the start-of-art lock profiling tools such as LiMiT have serious memory interferences and non-negligible inaccuracy problems. Since the synchronization behaviors obtained by LiMiT have big difference from those obtained by HaLock, we argue that hardware-enabled lock profiling mechanism is necessary to analyze lock behaviors.

7. ACKNOWLEDGMENTS

We would like to thank Guangming Tan, Mingyang Chen and other teammates from ASL, ICT, and the anonymous reviewers for useful suggestions and insightful feedbacks. This research is supported by the National Basic Research Program of China (973 Program) under the grant number 2011CB302502 and the National Natural Science Foundation of China (NSFC) under the grant number 60925009, 60903046 and 60921002.

8. REFERENCES

[1] AMD CodeAnalyst. http://developer.amd.com/tools/codeanalyst/pages/default.aspx.

[2] HMTT. http://asg.ict.ac.cn/hmtt/index.html.

[3] Intel Thread Profiler. http://software.intel.com/en-us/articles/using-intel-thread-profiler-for-win32-threads-philosophy-and-theory, August 2007.

[4] Intel VTune Amplifier XE. http://software.intel.com/en-us/intel-vtune.

[5] Jucprofiler. http://www.infoq.com/articles/jucprofiler.

[6] LockStat. http://hub.opensolaris.org/bin/view/Community+Group+dtrace/WebHome.

[7] TopMC. http://asg.ict.ac.cn/projects/topmc/.

[8] Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 3A & 3B, 2011.

[9] Yungang Bao, Mingyu Chen, Yuan Ruan, et al. HMTT: a platform independent full-system memory trace monitoring system. In *ACM SIGMETRICS: International Conference on Measurement and Modeling of Computer Systems*, pages 229-240, 2008.

[10] Christian Bienia, Sanjeev Kumar, Jaswinder PalSingh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[11] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2012.

[12] R. Bryant and J. Hawkes. Lockmeter: Highly-Informative Instrumentation for Spin Locks in the Linux Kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 271-282, 2000.

[13] D. R. Butenhof. Programming with POSIX threads. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[14] R. Daigle, C. Xia, and J. Torrellas. Low Perturbation Address Trace Collection for Operating System, Multiprogrammed, and Parallel Workloads in Multiprocessors. *Technical report*, Center for Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign, Mar. 1996.

[15] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Chrysos. Profileme: hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th annual ACM/IEEE International Symposium on Microarchitecture*, pages 292-302, 1997.

[16] J. Demme and S. Sethumadhavan. Rapid identification of architectural bottlenecks via precise event counting. In *Proceedings of the 38th International Symposium on Computer Architecture*, pages 353-364, 2011.

[17] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 335-346, 2010.

[18] Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Parallel Application Memory Scheduling. In *Proceedings of the 44th International Symposium on Microarchitecture*, pages 2011.

[19] Stijn Eyerman and Lieven Eeckhout. Modeling critical sections in amdahl's law and its implications for multicore design. In *SIGARCH Comput. Archit. New*s, pages 38:362-370, 2010.

[20] J. K. Flanagan, B. E. Nelson, J. K Archibald, K. S. Grimsrud. BACH: BYU Address Collection Hardware, The Collection of Complete Traces. In *Computer Performance Evaluation '92: Modeling Techniques and Tools*, 1993.

[21] B. Jacob, S. Ng, and D. Wang. Memory systems: Cache, dram, disk. In *Elsevier*, 2008.

[22] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: A scalable storage manager for the multicore era. In *Proc. 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 24–35, New York, NY, USA, 2009.

[23] Milo M. K. Martin, Mark D. Hill, Daniel J. Sorin. Why On-Chip Cache Coherence is Here to Stay. In *communications of the ACM,* vol 55, no 7, July 2012.

[24] M. Martonosi, D. W. Clark, and M. Mesarina. The SHRIMP Hardware Performance Monitor: Design and Applications. In *Proceedings of SIGMETRICS Symposium on Parallel and Distributed Tools,* pages 61-69, February 1996.

[25] Onur Mutlu and Thomas Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 63-74, 2008.

[26] N. Qu, X. G. Gou, and X. Cheng. Using Uncacheable Memory to Improve Unity Linux Performance. In Proceedings of the 6th Annual Workshop on the Interaction between Operating Systems and Computer Architecture, pages 27-32, 2005.

[27] S. Rixner. Memory Controller Optimizations for Web Servers. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 355-366, 2004.

[28] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing Lock Contention in Multithreaded Applications. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 269–279, 2010.

[29] Josep Torrellas , Anoop Gupta , John Hennessy. Characterizing the caching and synchronization performance of a multiprocessor operating system, In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, p.162-174, 1992.