PTAT: An Efficient and Precise Tool for Tracing and Profiling Detailed TLB Misses

JIUTIAN ZHANG, YUHANG LIU, HAIFENG LI, XIAOJING ZHU, and MINGYU CHEN,
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences; University of Chinese Academy of Sciences, China

As the memory access footprints of applications in areas like data analytics increase, the latency overhead of translation lookaside buffer (TLB) misses increases. Thus, the efficiency of TLB becomes increasingly critical for overall system performance. Analyzing TLB miss traces is useful for hardware architecture design and software application optimization. Utilizing cycle-accurate simulators or instrumentation tools is very time-consuming and/or inaccurate for tracing and profiling TLB misses. In this article, we propose an efficient and precise tool to collect and profile last-level TLB misses. This tool utilizes a novel software method called Page Table Access Tracing (PTAT), storing last-level page table entries of certain workload processes into a reserved uncached memory region. Therefore, each last-level TLB miss incurred by user process corresponds to one uncached page table access to main memory, which can be captured and recorded by a hardware memory bus monitor. The detected information is then dumped into offline storage. In this manner, full TLB miss traces are collected and can be analyzed flexibly. Compared to previous software-based methods, this method achieves higher performance. Experiments show that, compared with a state-of-the-art kernel instrumentation method (BadgerTrap), which lacks complete dumping trace function, the speedup is still up to 3.88-fold for memory-intensive benchmarks. Due to the improved efficiency and completeness of tracing, case studies validate that more flexible profiling can be conducted, which is of great significance for TLB performance optimization. The accuracy of PTAT is verified by both dedicated sequence and performance counters.

CCS Concepts: • **General and reference** → **Performance**; • **Computer systems organization** → **Multicore architectures**; *Embedded software*;

Additional Key Words and Phrases: TLB misses, hardware profiling tool, memory trace collector, efficiency, precision

ACM Reference format:
Jiutian Zhang, Yuhang Liu, Haifeng Li, Xiaojing Zhu, and Mingyu Chen. 2018. PTAT: An Effcient and Precise Tool for Tracing and Profling Detailed TLB Misses. *ACM Trans. Embed. Comput. Syst.* 17, 3, Article 62 (May 2018), 17 pages.
https://doi.org/10.1145/3182174

This work is supported by National Key Research and Development Plan of China No. 2017YFB1001602, NSFC (National Science Foundation of China) No. 61772497 and No. 61521092, State Key Laboratory of Computer Architecture Foundation under Grant No. CARCH2601.
Authors' addresses: J. Zhang, Y. Liu (corresponding author), H. Li, X. Zhu, and M. Chen, State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences; University of Chinese Academy of Sciences, No. 6 Kexueyuan South Road Zhongguancun, Beijing, 100190, China; email: {zhangjiutian, liuyuhang, lihaifeng, zhuxiaoj, cmy}@ict.ac.cn.
Permission to make digital or hard copies of all or part 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 bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 ACM 1539-9087/2018/05-ART62 $15.00
https://doi.org/10.1145/3182174

1 INTRODUCTION

It is well known that the multi-level translation lookaside buffer (TLB) performance impacts the memory system performance, which is critical for overall computing system performance [8], [17]. Virtual memory is used to manage the allocation of physical memory resources to facilitate sharing and enforce protection. Memory resources are managed in chunks (referred to as pages), the granularity of which ranges from several KBs to many MBs or GBs [13]. Page tables are maintained by the OS to store the page table entries (PTEs), which contain per-page virtual-to-physical address translations together with page status bits. The physical counterparts of never-used virtual pages are not created, and the contents of those pages not currently in use may be paged out to tertiary storage so that those physical resources are reallocated to other processes. Each access to main memory must undergo a virtual-to-physical address translation, which slows down the performance and increases the energy. To mitigate such effects of this bottleneck, most CPUs include a hardware structure, TLB, to cache the PTEs of recently accessed pages. Accesses that hit in the TLBs will avoid the latency and energy costs of going to memory to access the page table. Similar to multi-level caches in the memory hierarchy, multi-level TLBs have become a useful leverage for boosting data access performance.

Applications in areas like data analytics have increasingly large memory access footprints, which require increasingly large TLBs. Servers targeting such applications have thus been built with ever larger main memory capacities, but there has been no commensurate growth in TLB sizes. To see the significant impact of this mismatch, consider the adaptive radix tree [28] in-memory database first, of which TLB misses account for 23% of the total index lookup time. Zhang et al. [29] find that TLB translation bottleneck is getting worse when memory capacity grows. Pandiyan et al. [23] show that using larger TLBs improves IPC and L2 utilization. Jacob et al. [15] find that large TLBs are necessary for reducing memory management overhead, and the costs of TLB miss interrupts will increase in future microprocessors. Using huge pages can reduce the number of allocated pages and improve TLB efficiency. For example, in Intel x86_64 architecture, 4KB/2MB/1GB page sizes are supported. If an application requires 4GB of memory space, using 4KB size for page allocation requires 1M PTEs, while using 2MB or 1GB page requires only 2K or 4 entries, respectively. However, huge pages have more serious internal fragmentation problem than small pages, which limits their effort and usage. Chen et al. [6] find that the *column* object is the primary cause of TLB misses of the Graph500 *BFS* program [1], and the performance improvement of using huge pages for the *column* object with multiple threads is much lower than that of single-thread cases.

Designing high-performance and energy-efficient memory hierarchies require insights into the behavior of current designs: when do they work well, and when do they fall short of expectations [19]. Profiling the TLB misses is the prerequisite for memory system optimization. Designing both efficient TLB architecture and TLB-friendly applications requires analysis of TLB miss behavior.

This article proposes an efficient and precise profiling method named Page Table Access Tracing (PTAT). The PTAT method focuses on last-level TLB misses. Based on a hybrid hardware-software method, Hybrid Memory Trace Toolkit (HMTT) [3], [11], the PTAT method monitors TLB misses as memory reference traces. PTAT puts new allocated Linux user-space PTEs of a workload process into a reserved uncached memory region (PTAT region). Since HMTT uses a hybrid hardware/software tracing mechanism, it dumps out off-chip page table memory access traces generated by a modified Linux kernel. With the PTAT trace parser to parse the traces mentioned above, it knows whether a DRAM main memory access is for page table or not. Thus, each last-level TLB miss of a monitored workload process causes a read to PTAT region without caching, which is captured by HMTT and further analyzed by trace parser. As the experiments show in Section 5,

time overhead of the PTAT method is much lower than the BadgerTrap [8] method for memory-intensive benchmarks selected from several commonly used benchmark suites.

The main contributions of this article are as follows:

- —A novel method that dynamically puts page tables of workload processes to a dedicated uncached region is proposed to force all TLB-missed page table entry memory accesses to go to the memory bus and then be monitored by hardware snooping tools. To the best of our knowledge, no methods similar to ours have been proposed for memory-snooping-based TLB miss tracing.
- —A novel tool implementation is proposed for collecting and analyzing detailed TLB miss traces of workloads through a hybrid hardware/software mechanism, with lower overhead (up to 3.88-fold) and less interference compared with other software methods. We testified correctness of the PTAT tool through a linear sequential per-page accessing program. Experiments show that the time overhead of the PTAT method is only 15–54% of BadgerTrap for memory-intensive benchmarks. The PTAT slowdown of the serial version of *RandomAccess* [25] reaches the highest value of 8.67 under 1GB input data size.
- —Two case studies of profiling per-page TLB miss distribution and huge page TLB misses are conducted to identify hotspot data structures that incur the address translation overhead, which provides valuable guidelines for further optimization. For example, we find that SPECCPU2006 [9] 429.*mcf* generate about 63.5% of TLB misses in 5% of PTEs, with orders of magnitude of more TLB misses than other workloads. We also find that while the Graph500 *BFS* program turns to use huge pages for *column* object, the main cause of TLB misses of *BFS* is changed from *column* object to *rowstarts* and *pred* object.

The rest of the article is organized as follows. Section 2 describes the related work. Section 3 presents the design of the PTAT method. Section 4 presents the implementation of the PTAT method, including kernel modifications, PTAT controller, and PTAT TLB miss parser program. Section 5 describes the verification and evaluation results. Section 6 describes the case studies of per-page TLB miss profiling. Section 7 discusses usage extensions of the PTAT method. Finally, Section 8 concludes the article.

2 RELATED WORK

There are several ways to collect traces or statistics for TLB misses, including cycle-accurate software simulation, hardware counters, kernel instrumentation, and hardware snooping.

2.1 Cycle-Accurate Simulation

For memory access behavior profiling, the advantage of software simulation is on accuracy, but the speed is remarkably low. Full-system cycle-accurate simulators, such as QEMU [4], GEM5 [5], SIMICS [20], and MARSSx86 [24], support booting the operating system directly, simulating kernel privileged operations, and emulating hardware devices. Full-system simulators reach a higher accuracy because of these features. However, they are often too slow to generate full-system simulations, especially for measuring the performance impact of TLB misses, making running large-scale program unacceptable. Cycle-accurate simulators for memory management unit (MMU) research have the following disadvantages [8], especially for being compared with our PTAT method. First, rare events such as TLB misses require long simulations for their performance management, especially with tracing. For example, we measured that the speed of GEM5 is 50–500 KIPS for most workloads. Second, a cycle-accurate simulator requires a long startup time to initialize memory for big-data workloads. Third, a simulator often loses accuracy

when it comes to real system optimization, including TLB misses, since the design details (e.g., the replacement policies) of commercial processors are often not available for common users.

2.2 Hardware Performance Counters

Hardware performance counters are useful, fast, and low-overhead tools for memory access behavior profiling, but cannot provide detailed memory addresses. These counters provide accurate event statistics, e.g., cache misses and TLB misses. Oprofile [2], VTune [14], and PAPI [21] read performance counters through interrupts or system calls by sampling. TopMC [10] enables us to read and collect performance counters in Linux user or kernel level directly through low-cost instructions. Liu et al. [18] proposed a data-centric memory behavior analysis method using a hardware performance counter. However, hardware counters do not provide complete and detailed memory reference traces with memory addresses while PTAT provides complete and detailed memory reference traces with memory addresses. For example, hardware counters cannot distinguish TLB misses of different processes or different memory regions.

2.3 Kernel Instrumentation

BadgerTrap [8] is described as a tool that allows online instrumentation of TLB misses. However, it uses a kernel instrumentation method rather than a binary instrumentation method. BadgerTrap v1.0 works under Linux x86-64 kernel v3.12.13 for creating and analyzing TLB miss traces. This tool converts page walks from hardware-assisted to software-assisted and uses software-assisted page walks to instrument TLB misses. BadgerTrap intercepts each hardware-assisted page walk and converts it into a page fault when an x86-64 TLB miss occurs. BadgerTrap sets a reserved bit in a PTE to intercept the hardware page walker, which makes the page walker throw a page fault exception with RSVD flag. A new software-assisted TLB miss handler in a modified kernel, which can be extended for online analysis, deals with the exceptional page fault and is used to distinguish TLB misses of different processes or different regions on the fly. However, extensions to the software-assisted TLB miss handler are still hard to dump out complete detailed TLB miss traces as the PTAT method for memory-intensive workloads. The reason is that writing detailed traces to disk is much slower than generating the TLB misses of memory-intensive workloads. Even without dumping function, the slowdown of workloads generated by BadgerTrap is about 1.58 to 29.1, according to our evaluations.

2.4 Hardware Snooping

Various hardware snooping tools have been proposed to monitor memory trace online, such as Lecroy Kibra 480 analyzer [16], SuperTrace [26] (for embedded systems), and other DDR3/DDR4 bus snooping tools [22]. However, logic analyzers are mainly used for hardware debugging, due to limited data collected and lacking software information.

HMTT [3], [11] is a hybrid hardware/software memory trace monitoring system. HMTT is compatible to DDR3 SDRAM, and it monitors memory reference traces including multi-level page table accesses, using a hardware snooping technology. HMTT was also enhanced to support various information collection [6], [7], [12]. HMTT uses an extra hardware, called HMTT board, which acts as a DIMM adaptor. HMTT board monitors all memory transaction signals on the DDR3 command bus when installing between the motherboard DIMM slot and the DRAM DIMM. The FPGA logic on the HMTT board interprets DDR3 protocol and further reorganizes the corresponding memory references. Thus, all memory traces are captured at cache block granularity (e.g., 64 bytes).

For offline trace analysis, an HMTT trace parser uses a reverse page table (RPT) [6], [11] structure, which is used for looking up the corresponding virtual address for each physical address in the DRAM access trace. The physical-to-virtual mapping is acquired by dumping the page table

of the OS for each process to build a RPT. A high-level event-coding mechanism is also used in HMTT, which encodes semantic information (e.g., page table update) into the memory address space.

Through the event-coding mechanism, HMTT also identifies the memory references of page table walks with a corresponding process identifier from other regular data access. However, the original HMTT cannot distinguish the memory references among different levels of page table walks. Moreover, the original HMTT does not support dumping huge page tables. It should be emphasized that TLB misses often generate fewer capturable page table entry accesses to main memory because of caching. For example, comparing cached page table access traces to uncached page table access traces, based on the HMTT method, our experiments show that at least 96% of TLB misses of *mcf* and *canneal* workload hit in the cache hierarchy. For *cactusADM, astar,* and *raytrace* workload, the TLB misses which hits in the cache hierarchy are even more than 99%. Thus, the original HMTT cannot capture detailed, precise TLB miss traces without a mechanism like PTAT.

The following two issues for snooping-based TLB miss-tracing tool remain unsolved, which will be addressed in our PTAT design. The first issue is how to ensure TLB miss entries are uncached, and to ensure corresponding page table walks are recognizable by snooping-based tools; the second issue is how to identify which user page access triggered the page walk for a current page table entry.

3 DESIGN OF PTAT

For fast and precise last-level TLB miss trace-generating, this article proposes the PTAT method. The main idea is to use hardware snooping tools such as HMTT to capture the page table entry addresses when TLB miss occurs.

For the first issue, as page tables are often being cached, it is likely that a TLB miss may not cause a read access to the page table copy in main memory. However, if last-level PTEs are allocated to an uncached memory region, any corresponding TLB miss will cause an uncached read access to this region. The read accesses to this region are captured and recorded by hardware snooping tools. The PTAT method works as follows. At the kernel boot time, PTAT marks a manually preselected continuous "PTAT region" as reserved and uncached. This region is managed independently after booting is finished. When a target workload process is started, the modified kernel puts its page table to the PTAT region. Then each TLB miss of the target process will cause an external memory access to the PTAT region that enables external monitoring. It is easy for hardware to monitor the PTAT region, which makes page walks caused by TLB misses recognizable. Although the OS might also require multiple memory accesses to PTEs during virtual-to-physical mapping changing, these kinds of cases rarely happen for most common applications, according to performance counter results in Section 5.

For the second issue, to regenerate the corresponding process ID and the virtual page address of TLB misses, physical address traces captured by HMTT as well as dumped mapping information will be combined and analyzed offline. The physical-to-virtual mapping and page table entry storage location are acquired by dumping to a kernel buffer whenever the mapping information changes. Encoded semantic information is sent and captured by HMTT for synchronization at the same time. For detailed offline analysis, the physical locations of PTEs are extended to the original dumping scheme. We also considered fixed 2MB huge page cases. In parts of Linux kernel, fixed 2MB huge PTEs use the same pte_t data type as normal 4KB PTEs. Thus, for fixed 2MB huge page TLB miss support, dumping logic modifications for distinguishing huge PTEs from normal ones are necessary. Figure 1 illustrates our PTAT method.

As illustrated in Figure 1, the monitoring system utilizes one or several hardware monitor boards plugged into DIMM slots of a traced system. The DDR3 memories of the traced system are plugged

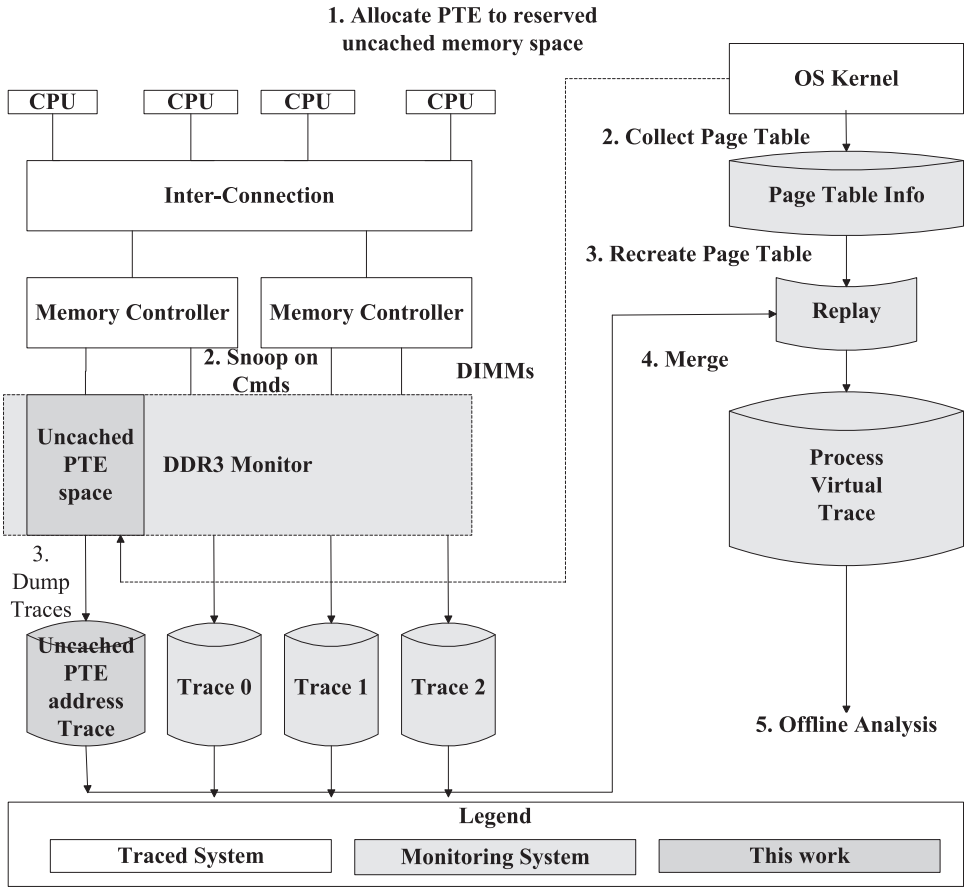


Fig. 1. The PTAT method. This method includes five steps. Note that memory tracing and page table tracing are conducted in parallel.

into the DIMM slots integrated on the hardware monitoring boards. The boards snoop on all memory commands via DIMM slots. For TLB miss tracing, workload PTE allocation to the PTAT region is needed. An on-board FPGA converts the commands into memory traces in this format $< address, r/w, timestamp >$, including PTAT region traces. Each hardware monitor board generates trace separately and sends the trace to its corresponding receiver via PCIe cable. With the synchronized timestamps, the separated traces are merged in the trace replay phase. Meanwhile, a module injected into OS kernel collects page table information, which is extended with PTE addresses and synchronizes the information with the PTAT memory trace dynamically. Then the page table information is used to reconstruct physical-to-virtual mapping information. Based on the information, i.e., memory trace, virtual-physical mapping and synchronization tags, we perform trace replaying procedure precisely and efficiently for offline analysis. A TLB miss trace parser is used for trace replay and offline analysis.

4 IMPLEMENTATION OF PTAT

Based on the main idea given above, our PTAT method is implemented as three components: kernel modifications, the PTAT control program, and the PTAT TLB miss trace parser.

4.1 Kernel Modifications

Kernel modifications play the critical role in PTAT implementation. Besides those modifications used by the original HMTT [11] to implement the PTAT method, primary modifications are described as follows:

4.1.1 Prepare PTAT Region. The beginning address, the end address, and the region size are defined as macros to allocate a continuous memory region in the kernel for PTAT. These macros define a 256MB memory region. It is also required to reserve this region from being used for other purposes and keep this region uncached to keep out of interference. We use a regular kernel function, which sets the region's e820 memory state to *E820_RESERVED_KERN* after common state initialization. The *E820_RESERVED_KERN* state means that this region is used for Linux kernel only (e.g., used for storing page tables and supporting variables).

To set this region as uncached, we add a function in the kernel. This function is called after the normal MTRR setting process while booting.

4.1.2 PTAT Region Allocator and Deallocator. The original Linux kernel uses a buddy memory allocator for allocating regular unreserved physical memory space to PTEs. However, because of the complexity of Linux buddy memory allocator implementation, to extend it for PTAT purpose is difficult. Thus, we use a simple, linear-based management method. A simple memory control block structure takes one-page size (4KB) for each in PTAT region, which is used to identify whether its next page is available or not. PTAT entries use the memory control block structure with PTAT page allocation and deallocation function (works similarly to *malloc*() and *free*()) and supporting variables. A sequential search for available space is used in PTAT page allocation function to allocate a page in PTAT region for storing page table, to find out whether a page is available in the corresponding control block. When PTAT region first works, an initialization function is called, which sets values to supporting variables. Note that the linear-based management of PTAT method does not affect regular (i.e., not PTE) memory allocation and usage behavior of workloads since the Linux buddy allocator still manages regular workload memory requirements.

4.1.3 Instrument Linux Page Table Entry Allocation. Compared with a dedicated global variable, the kernel knows whether a process ID belongs to the required workload process for PTAT page table entry allocation and deallocation. The workload process ID is passed to the module through the *sysfs* file system while the PTAT module gets loaded into the kernel.

However, simple modifications to page table entry allocation functions cause various kernel bugs, parts of which correspond to incorrect spinlock usage and page zones. For this reason, specialized PTAT entry allocation functions and corresponding macros are defined for correctly replacing common ones. For TLB miss instrumentation, various functions, which call page table entry allocation functions directly or indirectly, especially for functions handling page faults, are extended with PTAT entry allocation functions and *pid* switch logic. If *pid* of the current process equals the required workload process ID, the current process is assumed to belong to the required workload.

Similarly, for PTAT entry deallocation, the page table entry deallocation function is extended with PTAT entry deallocation functions and *pid* switch logic described above.

4.1.4 Dump Page Table Tracing with Virtual-to-Physical Mappings. We only need to add a small amount of extra information to achieve this purpose, since the original HMTT already has kernel function modifications to dump page tables. Given a hardware platform with a 16GB main memory, for example, to map a 16GB memory space requires at least 2^{22} of 4KB pages, i.e., each corresponding physical page number requires at least 23 bits, smaller than 3 bytes (24 bits). The

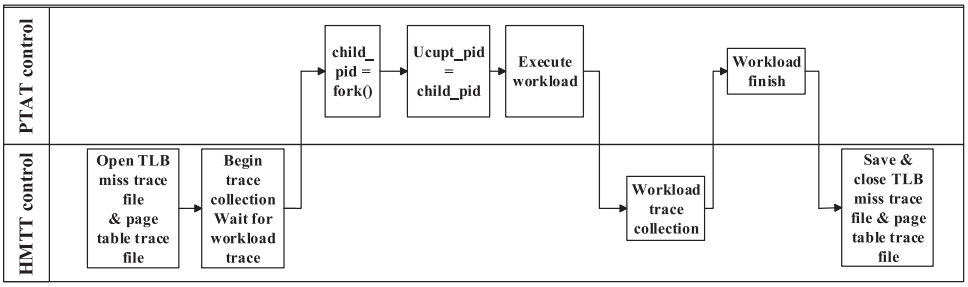


Fig. 2. The PTAT trace collecting procedure. We use a PTAT control program besides of the original HMTT control program.

page table dumping functions call a function to output a 13-byte trace entry to a binary file, including trace type (1 byte), process ID (4 bytes), physical page number (3 bytes), and virtual page number (5 bytes). Thus, we just need to add an extra trace output for 13 bytes PTE physical address after each virtual-to-NORMAL-physical ("NORMAL" means not PTE) mapping trace. Similarly, to map a 256GB memory space requires at least 4 bytes of each physical page number. Under x86_64 architecture, both common and uncached PTEs are allocated in a virtual address space (from *0xffff880000000000* to *0xffffc7ffffffff*), which space is direct mapping to all physical memory. Thus, an address bit calculation is used to calculate the physical address from the virtual address of page table entry pointer.

4.1.5 PTAT Control Module. A PTAT kernel control module is designed for passing command and workload process IDs to the kernel by control programs. Currently, only one process is set to use PTAT at one time during each workload running.

4.2 PTAT Control Program

For controlling the PTAT dumping process, besides the original HMTT control program, we use a PTAT control program. Figure 2 illustrates the PTAT trace collecting procedure. Description of the PTAT dumping process is as follows. First, the HMTT control program tells HMTT hardware to open a PTAT TLB miss trace file for tracing hardware addresses of page walks, and tells kernel module to open a page table trace file for tracing page table. Then the HMTT control program begins to wait for workload dumping. Next, the PTAT control program uses a Linux system call *fork*() to run and get the process ID of the workload process (as a child process of the control program). Before the workload process gets running, *sleep*() for 1 second and use PTAT device module function to set workload process ID. For the main process of PTAT control program, it opens the sysfs described above as a file, and write the required process id to the path. Then use *ioctl*() to pass the process id to a global variable of Linux kernel. Third, the PTAT control program calls *execvp*() to run the workload process. Then, both TLB miss traces and page table traces of the workload begin dumping. Finally, when the workload process finished running, save and close the PTAT TLB miss trace file and the page table trace file.

4.3 PTAT TLB Miss Trace Parser

The trace parser builds RPT [11] structure with dumped physical-to-virtual mapping and storage locations of PTEs. For TLB miss trace parsing, the first step is looking up RPT to find out whether a memory read trace belongs to a page table entry with virtual-to-physical mapping info. If true, the memory read trace is considered as corresponding to a TLB miss. Next, RPT updates the state

ofthe corresponding entry address in it to the missed state. Corresponding TLB miss statistics also update during the analysis process.

5 VERIFICATIONS AND EVALUATIONS

In this section, at first, we will show the experimental setup for PTAT verifications and evaluations. Next, we will testify the correctness of the PTAT method through a linear sequential per-page accessing program. Third, we will focus on evaluating the runtime overhead of collecting TLB miss traces and TLB miss counts of memory-intensive workloads using the PTAT method, compared with the TopMC [10] method. The TopMC method collects both D-TLB load misses and D-TLB store misses by reading performance counters.

5.1 Experimental Setup

For verification and evaluation, we use two different hardware platforms, Platform A and Platform B. Both Platform A and Platform B use Intel Xeon CPU E5-2620 v2 processors, which work under 2.10GHz with HyperThread enabled. The E5-2620 v2 processor has 6 cores, 12 physical threads, and 3-level caches. Each L1 instruction or data cache is 32KB and each L2 cache is 256KB. Both the L1 and the L2 are private caches. The L3 cache is 20-way 15MB and shared by all 6 cores in the processor. The cache block size is 64 bytes for all caches in the hierarchy. For Platform A with two E5-2620 v2 processors, the total capacity of the physical memory is 256GB with eight dual-ranked of DDR3-800MHz. For Platform B with one E5-2620 v2 processor, the total capacity of the physical memory is 16GB with one dual-ranked of DDR3-800MHz.

As the baseline OS environment, we use CentOS 6.6, an enterprise Linux distribution. For Linux x86-64 kernel, we choose versions 3.10.93 and 3.12.13 with the transparent huge page (THP) disabled for Platforms A and B, respectively. We use a lightweight profiling tool TopMC to read performance counters for TLB miss. TopMC works like PAPI without using system calls. We also use BadgerTrap [8] described in Section 2 for performance overhead comparison to PTAT for Platform B due to Linux kernel version restrictions. For the BadgerTrap method, we use default TLB miss statistic outputs, which do not include full TLB miss traces.

For 4KB page performance evaluation, we use both Platform A and Platform B. For Platform A, first, we have evaluated 29 benchmarks (i.e., nearly the entirety) of SPECCPU2006 benchmark suite through the PTAT method and the TopMC method, and 16 benchmarks of PARSEC benchmark suite through the TopMC method. Next, we have selected five integer benchmarks and one floating-point benchmark (*cactusADM*) from SPEC2006 as single-thread workloads, and *raytrace (RT)* and *canneal (CN)* from PARSEC as multi-thread workloads for data presentations. We select these memory-intensive workloads from corresponding benchmark suites by using the ratio of TLB miss counts divided by L1 cache replacements, with 15% as a borderline to identify whether a benchmark is memory-intensive and selected for evaluation. For Platform B, we have selected two integer benchmarks and four floating-point benchmarks from SPEC2006 as single-thread workloads, *fft* from SPLASH-2, *streamcluster*(*SC*), *x*264 from PARSEC, the OpenMP, and the serial version of HPCC *RandomAccess*(*RA*) Benchmark as multi-thread workloads. We use reference data set and one iteration for SPEC2006 workloads. For 2MB huge page performance evaluation, we use Platform B and have selected sequential version of Breadth-First Search (*BFS*) program from Graph500, and the serial version of HPCC *RandomAccess*(*RA*) Benchmark as single-thread workloads. For the *BFS* program, similar to work of Chen et al. [6], we choose scale 23 and edge factor 16 (generates a nearly 2GB graph), focus on the performance of running *BFS* step, and just store the *column* object of the *BFS* program in 2MB huge pages (allocated through Linux hugetlbfs file system). For the serial version of HPCC *RandomAccess* Benchmark, we choose 1GB, 2GB, 4GB as

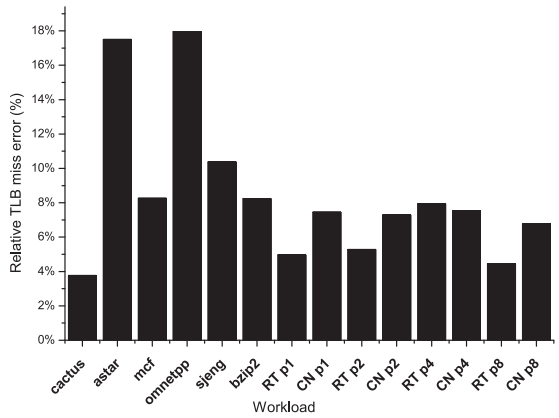


Fig. 3. The TLB miss counts comparison (Platform A, 4KB PTAT). The relative TLB miss error between the PTAT method and the TopMC method is less than 18%. For two PARSEC multi-thread workloads, the relative TLB miss error between the PTAT method and the TopMC method is less than 8%.

different scales, and store the *Table* object in 2MB huge pages. To measure an actual running time of the workloads with both PTAT and TopMC, we use Linux command *time*.

5.2 Function Correctness Verifications

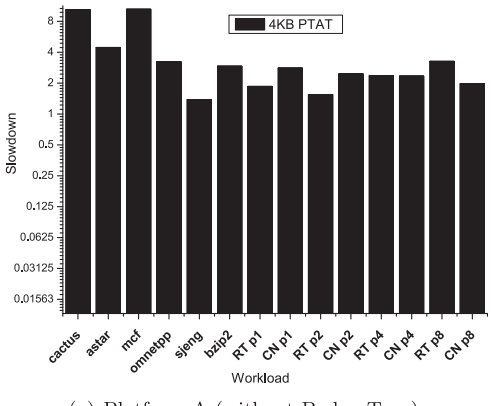
Two experiments are designed to verify the correctness of TLB miss trace collected by the PTAT method.

First, we build a linear sequential per-page accessing program. Since all memory accesses are predictable, we compare the access trace with predicted sequence one by one. This program maps a reserved and uncached memory region and reads first address in each 4KB page of the uncached region in increasing order. It also outputs the first virtual address of the uncached region. Through offline trace analysis, we get sequential reads equal to uncached page counts, and find that each accessed address of an uncached 4KB page corresponds to a previously missed page table entry address. The trace perfectly matches expected memory access and TLB miss behavior, thus partially verifies the correctness of the PTAT method. For fixed 2MB huge page (opposed to THB) TLB miss verification, we also build a similar program, which maps a reserved uncached memory region using the hugetlbfs file system. The offline trace analysis for this huge page program also matches our expected behavior of memory access and TLB misses.

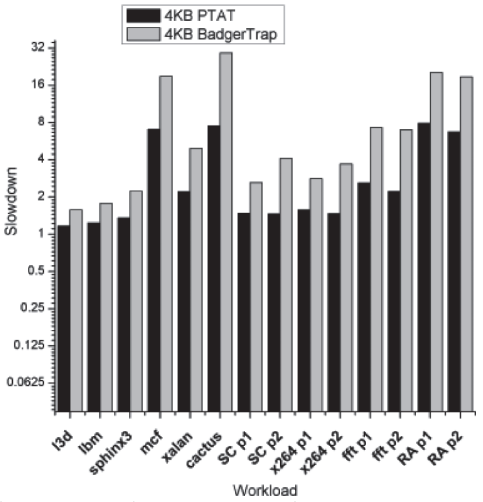
Second, we compared the statistical result of PTAT trace with performance counters for real workloads. Figure 3 illustrates the TLB miss counts comparison of the selected workloads for the PTAT method under Platform A, compared to performance counter values collected by the TopMC method. We use a metric called relative TLB miss error (*RE*), which is defined as Equation (1), where M_{PTAT} and M_{TopMC} means TLB miss count of PTAT and TopMC, respectively:

$$RE_{PTAT} = 1 - \frac{M_{PTAT}}{M_{TopMC}}. \tag{1}$$

For multi-thread workloads, $p1$, $p2$, $p4$, and $p8$ stand for running with 1, 2, 4, and 8 threads, respectively. It shows that 429.mcf and 436.*cactusADM* generate orders of magnitude more TLB misses than other workloads. For all workloads, the TLB miss count captured by the PTAT method is smaller than that captured by the TopMC method. The relative TLB miss error between



(a) Platform A (without BadgerTrap).



(b) Platform B (with BadgerTrap). The OpenMP version of the *RandomAccess* benchmark is used. For all workloads, BadgerTrap gets a higher slowdown value.

Fig. 4. The slowdown of different workloads under 4KB page configurations. For single thread workloads, 429.*mcf* and 436.*cactusADM* get the highest slowdown. For multi-thread workloads, *RandomAccess p*1 gets the highest slowdown.

the PTAT method and the TopMC method is less than 18%. For two PARSEC multi-thread workloads, the relative TLB miss error between the PTAT method and the TopMC method is less than 8%. Since the TopMC method does not distinguish TLB misses of workload processes from other processes, the performance counters of processor cores running workloads suffers from significant interference due to process scheduling, leading to higher counts than the actual case.

5.3 Performance Evaluations

We compared the performance of PTAT with that of BadgerTrap under the same workloads. It should be emphasized again that BadgerTrap does not have a trace output function by now, although it is possible theoretically with more overhead. Thus, the BadgerTrap result only means its instrumentation part of the overall overhead. The original HMTT seldom interferes with workloads [3], [11]. For each page table walk progress of normal memory access, one access to an uncached PTE (i.e., in main memory) takes up to 10 times longer latency than a cached PTE. Thus, the performance overhead of the PTAT method mainly comes from the uncached PTE accesses because of the page table walk progress. Figures 4 and 5 illustrate the relative workload running slowdown of PTAT and BadgerTrap methods, compared to the original running time while running selected workloads (lower is better). Hardware counter methods such as TopMC seldom interfere with workloads. Figure 4(a) illustrates the PTAT slowdown of different workloads under 4KB page configurations of Platform A, compared to baseline running. For single-thread workloads, 429.*mcf* and 436.*cactusADM* get the highest slowdown of 10.5 and 10.4, respectively. For multi-thread workloads, *raytrace p*8 gets the highest slowdown of 3.27 times, which decreases to 1.54 times while running with two threads. For *canneal* workload, running with one thread gets the highest slowdown of 2.8 times.

Figure 4(b) illustrates the PTAT and BadgerTrap slowdown of different workloads under 4KB page configurations of Platform B. In Figure 4(b), the OpenMP version of the *RandomAccess*

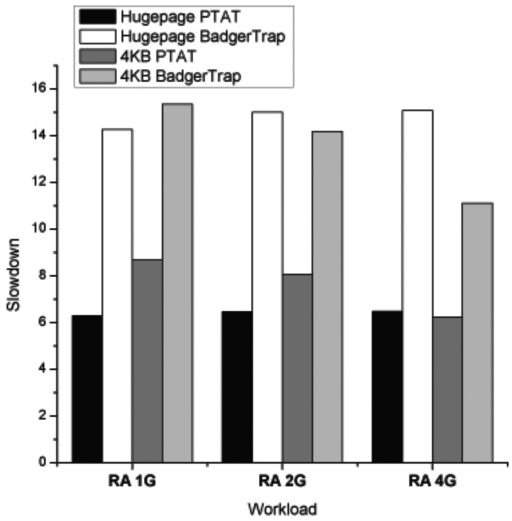


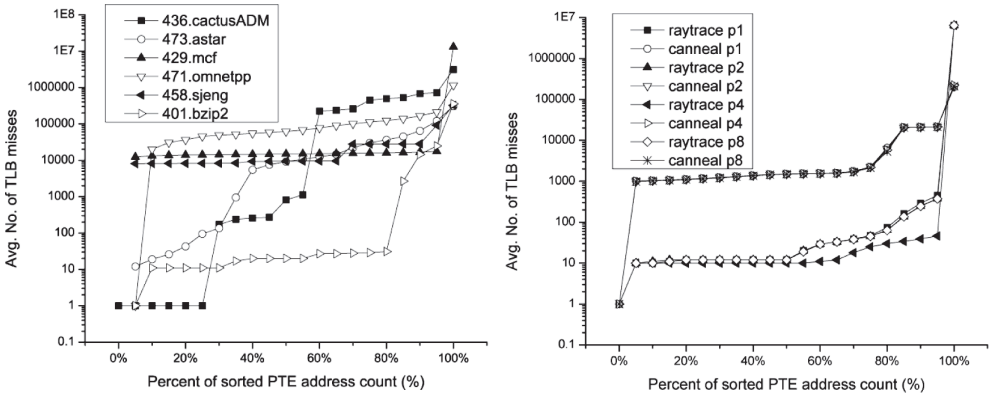
Fig. 5. The slowdown of the serial version of *RandomAccess* under 4KB page and 2MB huge page configurations. Both PTAT and BadgerTrap reach the smallest values at 4GB input data size.

benchmark is used. For all workloads, BadgerTrap gets a higher slowdown value. For single-thread workloads, 436.*cactusADM* gets the highest slowdown of the BadgerTrap method of 29.1, with a value of 7.50 of the PTAT method. 437.*leslie3d* gets the lowest PTAT slowdown for 1.16 times, with a BadgerTrap slowdown for 1.58 times. For multi-thread workloads, *RandomAccess* $p1$ gets the highest PTAT slowdown for 7.89 times, which decreases to 6.73 times while running with 2 threads. *RandomAccess* $p1$, $p2$ also reaches a slowdown of the BadgerTrap method higher than the slowdown of the PTAT method, at 20.3 and 18.6 times, respectively. Overall, the time overhead of the PTAT method is only 15–36% of BadgerTrap for memory-intensive benchmarks under 4KB page configurations. The speedup of the PTAT method over the BadgerTrap method is up to 3.88-fold. Compared with the performance counter and instrumentation methods, the PTAT method achieves the modest runtime slowdown and the best information completeness.

Figure 5 illustrates the slowdown of different workload scales and page sizes under Platform B. In Figure 5, the serial version of the *RandomAccess* benchmark is used. Under 4KB page configuration, we find that both PTAT and BadgerTrap reach the smallest values at 4GB input data size, of 6.23 and 11.1, respectively. Also, both PTAT and BadgerTrap reach the highest values at 1GB input data size, of 8.67 and 15.3, respectively. The ratio of the slowdown of the PTAT method over the BadgerTrap method is steadily from 56% to 57%. Under 2MB huge page configuration, while the input data size of *RandomAccess* grows, both PTAT and BadgerTrap generate a slightly increasing slowdown. Under 2GB and 4GB input data size, the slowdown of BadgerTrap gets even higher values than 4KB page configuration, reaches 14.0 and 14.1, respectively. The ratio of the slowdown of the PTAT method over the BadgerTrap method is steadily from 43% to 44%.

6 CASE STUDIES OF PROFILING

To demonstrate the ability for the per-page statistic, we analyze the distribution of TLB miss counts among PTE physical addresses using PTAT under Platform A and Platform B. Per-page huge page (libhugetlbfs) TLB miss distribution results are also shown.



(a) single-thread workloads.

(b) multi-thread workloads. Using 2, 4 and 8 threads get similar TLB miss distribution behaviors with which using only single thread.

Fig. 6. The average number of TLB misses (Platform A).

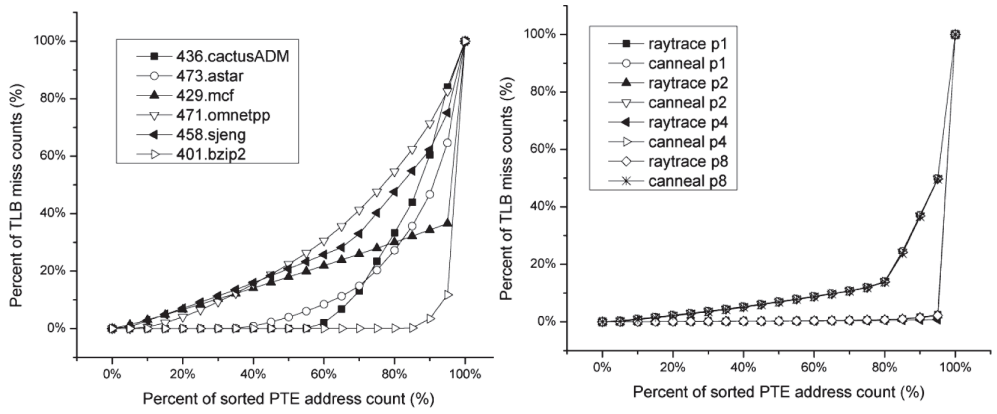
6.1 Per-Page TLB Miss Distribution

As HMTT hardware generates memory read trace at 64-byte (Platform A) or 32-byte (Platform B) granularity, reads to 8 (Platform A) or 4 (Platform B) continuous PTE address (8 bytes for each) generate the same memory reference traces. Thus, PTAT trace parser program generates average counts of TLB misses for per-page analysis in each 8 (Platform A) or 4 (Platform B) continuous allocated TLB entries rather than each TLB entry. Figures 6(a) and 6(b) illustrate the average number of TLB misses under Platform A, collected through the PTAT method for single-thread and multi-thread workloads, respectively. The *x*-axis illustrates the percentage of TLB entry addresses that are sorted by TLB misses in increasing order. For single-thread workloads, except for 436.*cactusADM* and 473.*astar*, their average number of TLB misses increases slowly until the percent of TLB entry addresses reaches 80%. For multi-thread workloads, using 2, 4, and 8 threads get similar TLB miss distribution behaviors using only single thread. The *raytrace* and *canneal* workload have a nearly constant average number of TLB misses in lower 50% and 75% of sorted TLB entry addresses, respectively.

Figures 7(a) and 7(b) illustrate cumulative distribution function (CDF) of the per-page TLB miss for single-thread and multi-thread benchmarks under Platform A, respectively. These results provide valuable insights for TLB miss optimization. The *x*-axis illustrates the percent of TLB entry addresses that are sorted by increasing TLB misses order too. The *y*-axis illustrates the percent of total TLB miss counts in previously sorted TLB entry addresses. For single-thread workloads, more than 50% of total TLB misses are generated from 25% of the TLB entry addresses. Especially, for 429.*mcf*, more than 63.5% of TLB misses are generated from the last 5% of sorted TLB entry addresses. For multi-thread workloads, more than 80% of total TLB misses are generated from 20% of the TLB entry addresses. The PTAT method helps to identify pages and corresponding data structures, which cause major address translation bottleneck. Thus, using this method for workload optimization is a well-considerable choice.

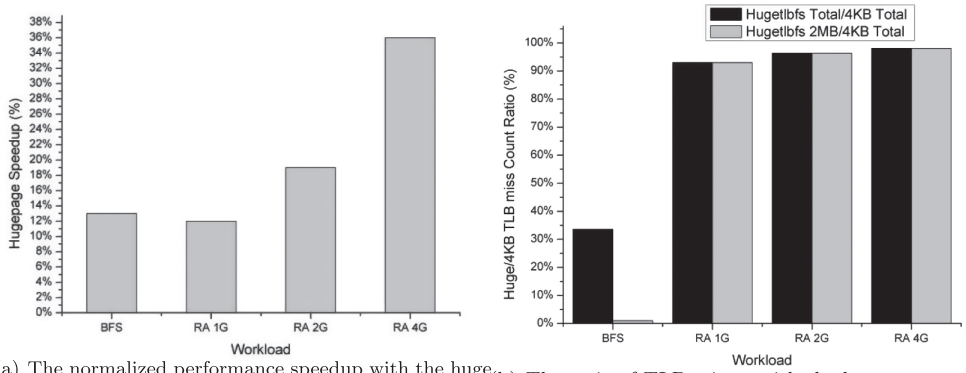
6.2 Huge Page TLB Misses

Figure 8(a) illustrates the normalized performance speedup with huge page configuration for the *BFS* program and serial version of HPCC *RandomAccess* (RA) benchmark under Platform B. Note



(a) single-thread workloads. (b) multi-thread workloads.

Fig. 7. The CDF of the per-page TLB miss (Platform A).

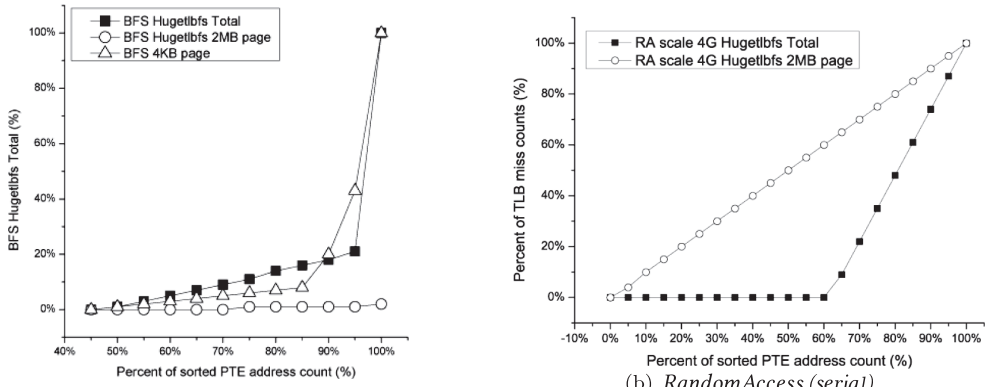


(a) The normalized performance speedup with the huge pages. (b) The ratio of TLB misses with the huge pages over those of 4KB pages for the PTAT method.

Fig. 8. Performance speedup and TLB miss comparison of *BFS* and *RandomAccess* (serial). Through putting the *column* object into 2MB huge pages, *BFS* achieves up to 66% TLB misses reduction and results in 13% performance improvement. For *RandomAccess*, about 2–7% TLB miss reduction is achieved by huge pages, which results in a maximum 36% speedup under 4GB input data size.

that the normalized performance speedup is measured under the original Linux environment without PTAT. Figure 8(b) illustrates the ratio of TLB misses of *BFS* and *RandomAccess* with the huge pages over those with 4KB pages only for the PTAT method under Platform B. Figures 9(a) and 9(b) illustrate CDF of the per-page TLB miss of *BFS* and *RandomAccess* under Platform B, respectively.

First, we analyze the evaluation results of *BFS*. The *column* object is the main cause of page memory walks in the *BFS* program, and the next main cause is the *rowstarts* and the *pred* objects [1], [6]. Through putting the *column* object into 2MB pages, the number of page memory walks caused by the *column* object reduces. Thus, it achieves up to 66% TLB miss reduction that results in 13% performance improvement. Due to inner-vertex continuous access behavior of the *column* object, the *BFS* program generates 2MB huge page TLB misses under huge page configuration for about 1% of total misses under normal 4KB page configuration. As the *column* object uses only 2MB pages under proposed huge page configuration, the main cause of TLB misses of *BFS* is changed



(a) *BFS*. (b) *RandomAccess (serial)*.

Fig. 9. The CDF of the per-page TLB miss of *BFS* and *RandomAccess* (serial). *BFS* generates fewer "hot pages" while putting the *column* object into 2MB huge pages. All huge pages of *RandomAccess* generate TLB miss count values nearly equal.

from *column* object to the *rowstarts* and the *pred* objects. Under normal 4KB page configuration, *BFS* reaches more than 90% of total TLB misses from the last 15% of the sorted TLB entry addresses. Using huge pages for *BFS*, it generates not only significant TLB miss reduction but also fewer "hot pages," which caused most (about 80%) of total TLB misses.

For *RandomAccess*, only about 2–7% TLB miss reduction is achieved by huge pages, which results in a maximum 36% speedup under 4GB input data size, however. The main cause of TLB misses under huge page configuration of *RandomAccess* is still the *Table* object in 2MB pages. As the input data size of *RandomAccess* increases, it generates a constant per-page TLB miss distribution behavior for both huge page and 4KB-only configuration. Thus, we only show per-page TLB miss distribution of 4GB input data size in Figure 9(b). All huge pages of *RandomAccess* generate TLB miss counts values nearly equal, which take about 40% of walked pages. This result is due to random access behavior of the *Table* object huge pages in the workload.

7 DISCUSSIONS

The PTAT method can be further extended to analyze various kinds of page walk statistics.

First, statistics of nested page tables for virtualized platforms can be gathered [27]. To do this, dumping nested page tables, marking hypervisor, guest machine, and guest workload process IDs are needed.

Second, this work can be extended to support collecting TLB miss traces from multiple workload processes simultaneously. However, as page table is shared among different threads of the same process, to distinguish memory accesses of a thread from those of other threads in the page walk progress is hard.

Third, the PTAT method dumps page table from the Linux kernel to get the virtual address of a TLB miss. Thus it can be used to identify whether a TLB miss corresponds to the code segment, program stack, or any program object. Detailed instrumentations to the Linux kernel are needed to do this.

Fourth, supporting THP TLB miss statistics is required as THP in Linux are widely used in modern systems for improving TLB efficiency. Proper instrumentations to huge page allocation logics are needed to do this.

Fifth, modifications of control bits in PTEs by OS can be collected for page permission profiling. These accesses can be pulled from creations of entries through extra instrumentations to kernel functions.

Finally, the PTAT method can be extended for monitoring accesses to any specific data structures of Linux kernel by moving data structures to uncached regions. It should be emphasized again that the OS does not change mapping frequently for most common applications, which means that our PTAT method seldom interferes with workloads.

8 CONCLUSIONS

This article proposes a novel tool PTAT for detailed TLB miss tracing and profiling. The PTAT method works by putting new allocated Linux user-space PTEs of workload processes into a reserved uncached memory region. TLB misses of monitored workload processes will cause DRAM read operations to the PTAT region, which are captured by hardware monitors and dumped into the offline storage for further analysis.

Compared with dedicated memory access sequence and performance counters, the PTAT method presents accurate results. Thus, the correctness of the PTAT method is well-verified. Experimental results show that the runtime overhead of PTAT is significantly lower than the previous software-based BadgerTrap method, especially under memory-intensive workloads. Through various detailed TLB and memory corresponding statistics, it is shown that, in the case studies, workload or system software optimizations for memory access can be promoted by the tracing and profiling results of PTAT.

REFERENCES

[1] D. A. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, W. Mann, and Theresa Meuse. 2011. The Graph 500 List. Retrieved from http://graph500.org.

[2] Will Cohen, Suravee Suthikulpanit, Will Deacon, Gilles Allard, Daniel Hansel, and Robert Richter. 2017. Oprofile. Retrieved from http://oprofle.sourceforge.net.

[3] 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. *Meas. Model. Comput. Syst.* 36, 1 (2008), 229–240.

[4] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*. 41–46.

[5] Nathan Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The GEM5 simulator. *ACM SIGARCH Computer Archit. News* 39, 2 (2011), 1–7.

[6] 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 *2012 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'12)*. IEEE, 46–57.

[7] Licheng Chen, Zhipeng Wei, Zehan Cui, Mingyu Chen, Haiyang Pan, and Yungang Bao. 2014. CMD: classification-based memory deduplication through page access characteristics. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 65–76.

[8] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2014. BadgerTrap: A tool to instrument x86-64 TLB misses. *ACM Sigarch Comput. Archit. News* 42, 2 (2014), 20–23.

[9] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Comput. Archit. News* 34, 4 (2006), 1–17.

[10] Yongbing Huang. 2011. TopMC: An Approach for Understanding Architectural Characteristics (Technical Report). Retrieved from http://asg.ict.ac.cn/projects/topmc.

[11] Yongbing Huang, Licheng Chen, Zehan Cui, Yuan Ruan, Yungang Bao, Mingyu Chen, and Ninghui Sun. 2014. HMTT: A hybrid hardware/software tracing system for bridging the DRAM access trace's semantic gap. *ACM Trans. Archit. Code Optim.* 11, 1 (2014), 7.

[12] Yongbing Huang, Zehan Cui, Licheng Chen, Wenli Zhang, Yungang Bao, and Mingyu Chen. 2012. HaLock: hardware-assisted lock contention detection in multithreaded applications. In *21st International Conference on Parallel Architectures and Compilation Techniques*. ACM, 253–262.

[13] Intel. 2013. *Intel-64 and IA-32 Architectures Software Developer's Manual*, Vol. 3A: System Programming Guide, Part 1, 64 (2013), (3–11–20)–(3–11–33).
[14] Intel. 2017. Intel VTune Amplifier 2017. Retrieved from http://software.intel.com/en-us/intel-vtune.
[15] Bruce L. Jacob and Trevor N. Mudge. 1998. A look at several memory management units, TLB-refill mechanisms, and page table organizations. In *ACM SIGOPS Operating Systems Review*, Vol. 32. ACM, 295–306.
[16] Teledyne LeCroy. 2017. Kibra 480 analyzer. Retrieved from http://teledynelecroy.com/protocolanalyzer.
[17] Yuhang Liu and Xian-He Sun. 2015. LPM: Concurrency-driven layered performance matching. In *2015 44th International Conference on Parallel Processing (ICPP'15)*. IEEE, 879–888.
[18] Yuhang Liu and Xian-He Sun. 2015. Reevaluating data stall time with the consideration of data access concurrency. *J. Comput. Sci. Technol.* 30, 2 (2015), 227–245.
[19] Yuhang Liu and Xian-He Sun. 2017. Evaluating the combined effect of memory capacity and concurrency for many-core chip design. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 2, 2 (2017), 9.
[20] Peter S. Magnusson, Mattias Christensson, J. Eskilson, Daniel Forsgren, G. Hallberg, J. Hogberg, F. Larsson, Andreas Moestedt, and B. Werner. 2002. SIMICS: A full system simulation platform. *IEEE Comput.* 35, 2 (2002), 50–58.
[21] Shirley V. Moore. 2002. A comparison of counting and sampling modes of using performance monitoring hardware. In *International Conference on Computational Science*. Springer, 904–912.
[22] Nobuyuki Ohba, Seiji Munetoh, Atsuya Okazaki, and Yasunao Katayama. 2014. Non-intrusive scalable memory access tracer. In *International Conference on Quantitative Evaluation of Systems*. Springer, 245–248.
[23] Dhinakaran Pandiyan, Shin-Ying Lee, and Carole-Jean Wu. 2013. Performance, energy characterizations and architectural implications of an emerging mobile platform benchmark suite-MobileBench. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 133–142.
[24] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. 2011. MARSS: A full system simulator for multicore x86 CPUs. In *Proceedings of the 48th Design Automation Conference*. ACM, 1050–1055.
[25] Steven J. Plimpton, Ron Brightwell, Courtenay Vaughan, Keith Underwood, and Mike Davis. 2006. A simple synchronous distributed-memory algorithm for the HPCC randomaccess benchmark. In *2006 IEEE International Conference on Cluster Computing*. IEEE, 1–7.
[26] Green Hills Software. 2017. Supertrace probe. Retrieved from http://www.ghs.com/products/supertraceprobe.html.
[27] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. 2005. Intel virtualization technology. *Computer* 38, 5 (2005), 48–56.
[28] Petrie Wong, Ziqiang Feng, Wenjian Xu, Eric Lo, and Ben Kao. 2015. TLB misses: The missing issue of adaptive radix tree? In *Proceedings of the 11th International Workshop on Data Management on New Hardware*. ACM, 6.
[29] Lixin Zhang, Zhen Fang, Mike Parker, Binu K. Mathew, Lambert Schaelicke, John B. Carter, Wilson C. Hsieh, and Sally A. McKee. 2001. The impulse memory controller. *IEEE Trans. Comput.* 50, 11 (2001), 1117–1132.

Received July 2017; revised January 2018; accepted January 2018