A Lightweight Hybrid Hardware/Software Approach for Object-Relative Memory Profiling

Licheng Chen[†‡], Zehan Cui[†‡], Yungang Bao[†], Mingyu Chen[†], Yongbing Huang[†‡], Guangming Tan[†]
[†]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
[‡]Graduate School of Chinese Academy of Sciences
{chenlicheng, cuizehan, baoyg, cmy, huangyongbing, tgm}@ict.ac.cn

***Abstract*—Memory profiling is the process of collecting memory address traces during the execution of a program, then analyzing and characterizing the memory behavior of the program offline. With the trend that there will be more and more cores integrated in a processor chip, the "Memory Wall" problem will become more serious in the chip multiprocessor (CMP) system. Thus accurate and effective memory profiling is becoming one of the keys to identify the source of memory system bottlenecks. A large body of work has been contributed to memory profiling, however, most adopts instrumentation, simulator which suffers heavy overhead, or hardware performance counter which is lack of detail trace information. Furthermore, correlating the raw memory address traces with object-relative information allows us to separate regular pattern for certain object from the irregular mixed, thus helps the optimization. In this paper, we propose a lightweight hybrid hardware/software approach for object-relative memory profiling. We monitor physical memory addresses through hardware snooping with negligible overhead; meanwhile we dump Linux kernel page tables of processes, as well as object-relative memory allocation information. Our approach supports not only to collect applications' full memory traces with detail object relative information, but also to identify hardware-generated memory accesses such as page memory walks due to TLB miss at object level. The experimental results on real system show that our approach is highly accurate (the largest error is 2.04%) and low overhead (the average overhead is 1.60%). Furthermore, we profile two multi-thread applications in detail, and successfully identity hot TLB-miss objects. With object-targeted optimization, we can improve applications' performance by nearly 6.86%.**

Keywords- hybrid; object; memory profiling; page memory walks; full memory traces

I. INTRODUCTION

Program profiling is an important technology for collecting a variety of information (such as control flow, memory access address) during the execution of applications, and it provides insight into the hotspot of resource usage and helps to identify performance bottlenecks. Thus, program profiling is widely used in guiding performance optimization, compilers optimization and system architecture design. With the trend that there will be more and more cores integrated in a processor chip, the memory system has been the main performance bottleneck in a chip multiprocessor (CMP) system (known as the "Memory Wall" problem), which is mainly due to two factors: high memory latency and relative lower available memory bandwidth for each core [28]. Accurate and effective memory profiling is one of the keys to identify the source of memory system bottlenecks in CMP systems, and it can be used to quantify locality [9][39], optimize prefetching [10], improve hardware cache partitioning performance [18], drive memory simulator, analyze data dependency , and optimize data layout [30] etc.

Memory behavior profiling can be done through many different ways, including: compiler-driven, dynamic instrumentation, simulation, and hardware performance counter. Compiler-driven instruments code for collecting memory trace during the compile time, thus it is a relative lightweight way. But it needs to recompile and relink programs, which is impossible for applications without source code. Dynamic instrumentation adopts Just-In-Time (JIT) compile technology to instrument code at runtime, and it can collect memory traces with rich information. However its main drawback is heavy overhead, which makes it unsuitable for running applications and collecting full traces for a long time. As shown in Soft-OLP [18], the overhead of object-level memory profiler with dynamic instrumentation (Pin) is nearly 30 to 80 times even with 10% sampling. Simulation is another common way for memory behavior profiling with rich information, however simulation has the accuracy problem, and the overhead of accurate simulation is significantly heavy. Hardware performance counter is a useful tool for measuring memory-related events. But the performance counter can provide only some statistics information, rather than detailed memory traces.

Table I summarizes these ways. It is noteworthy that all the above ways just collect the memory traces caused by explicit data accesses (or data cache misses), they are unable to distinguish implicit **page memory walks** (or page walks) which are the memory accesses for page tables due to TLB miss on real systems. Since the TLB miss can raise an impact on the overall system performance ranges from 5-14% for nominally sized applications [4], it has been becoming more and more important to profile the page memory walk behavior on real CMP system. Previous work often drove a TLB simulator with collected memory reference traces to study the behavior of TLB and tried to optimize TLB performance, however this simulation approach had the accuracy problem.

Wu et al. [40] presented an object-relative translation and decomposition method for effective memory profiling. The raw memory address trace was translated into object-relative format: (instruction-id, group, object, offset, time-stamp). The object-

TABLE I. SUMMARY OF DIFFERENT MEMORY PROFILING APPROACHES

	Accurate	Detailed	Low overhead	Page walks+
Instrument	√	√	×	×
Simulator	*	√	×	×
Performance Counter	√	×	√	*
Compiler	√	√	√	×
Hybrid H/S	√	√	√	√

Note: √-Yes, ×-No, *-Maybe

+Here we mean monitor memory accesses for page tables (due to TLB miss) on the real system, rather than with a TLB simulator. **Hybrid H/S** represents the hybrid hardware/software approach we present in this paper

relative memory profiling enables decomposition of a memory access stream to separate regular and interesting information form irregular memory accesses and to provide compiler with useful information for optimization. However, they adopted dynamic instrumentation (with pin tool) for memory profiling, which suffered heavy overhead, and their memory profiling can only collect explicit data memory accesses, without monitoring and identifying implicit page memory walks.

In this paper, we present a lightweight hybrid hardware/software approach for object-relative memory profiling. Instead of using heavy-overhead dynamic instrumentation, we adopt a hardware which is based on memory bus snooping technology to monitor memory transaction signal and dump physical memory reference traces with negligible overhead. The hardware is able to monitor full-system memory traces [3], including operating system, DMA [32], page memory walks, and data accesses etc. Meanwhile, we modify the Linux kernel to dump processes' page tables, and propose a synchronization mechanism between the page table traces and the physical memory traces. To support the object-relative memory profiling, we further provide a software toolkit to monitor object allocation information during the execution of a program. With our hybrid hardware/software approach, we can get the memory requests for data accesses, as well as identify page memory walks at object level on real systems by outputting memory trace with abundant semantic information as following:

<**paddr**, **r/w**, **time-stamp**, **pid**, **vaddr**, **object**, **page-walk**>

For each memory access trace, we can get rich information which is valuable for optimization, including: physical address (*paddr*), read or write (*r/w*), time stamp of the request (*time-stamp*), the id of the process who sends the memory request (*pid*), virtual address (*vaddr*), object id (*object*), and whether it is a page memory walk (*page-walk*).

In summary, we have made the following contributions in this paper:

- We propose a lightweight hybrid hardware/software approach for object-relative memory profiling with rich semantic information. We adopt a snooping-based hardware cooperative with lightweight kernel modification and a software toolkit to achieve this goal. The experimental results show that the approach is highly accurate and with low overhead.
- Besides the ordinary memory requests issued by data accesses, our approach can also capture and identify page memory walks at object level on real systems, which is valuable for locating TLB performance bottlenecks. To the best of our knowledge, this approach is the first to be able to distinguish page memory walks at object level on real systems.
- With our approach, we have performed the object-relative memory profiling on two multi-thread applications on a real system in detail, and show that we can successfully identify the memory and TLB performance bottlenecks at object level. And with object-targeted optimization, we can improve their performance by nearly 6.86%.

The rest of the paper is organized as follows. Section II gives a brief introduction to Object-Relative Memory Profiling, while Section III describes the implementation of our hybrid hardware/software approach in detail. Section IV presents the experimental results on a real system: it first discusses the validation to show its accuracy; and then it evaluates the overhead to show its effectiveness; at last, it gives two case studies. Section V presents an overview of related work, and Section VI gives a conclusion of the paper.

II. OBJECT-RELATIVE MEMORY PROFILING

An object is a group of data stored as a unit, for instance, an array with each element having the same type or a structure with elements might have different types. Thus for programmers, the object contains the high-level semantic information of data. Object-Relative Memory Profiling was first proposed by Wu et al. [40]. They converted the raw memory address into object-relative form which could be used to separate regular memory access pattern for each object from the mixed irregular memory traces.

As illustrated in [40], figure 1 shows the conceptual object-relative memory trace separation. The original memory traces are mixed with different processes, thus it might appear as irregular memory pattern. In the next blocks, the memory traces are separated by each process, it appears a little more regular, but still mixed with multiple objects. In the last blocks, the memory traces are further separated by objects, then the per-object traces might be quite regular. With the regular memory pattern for each object, many optimizations can be done at object-level, such as: guide prefetcher at object-level; optimize data-layout for different type of objects.

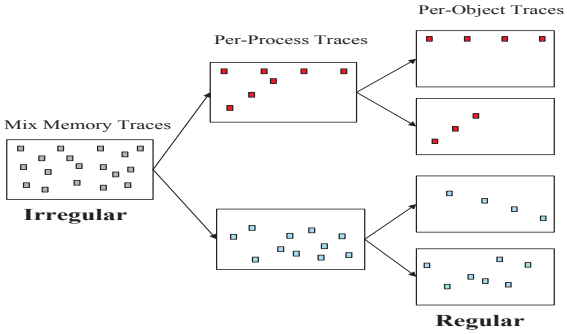

Figure 1. The process of object relative memory profiling to identify regular memory access pattern at object level [40].

III. IMPLEMENTATION

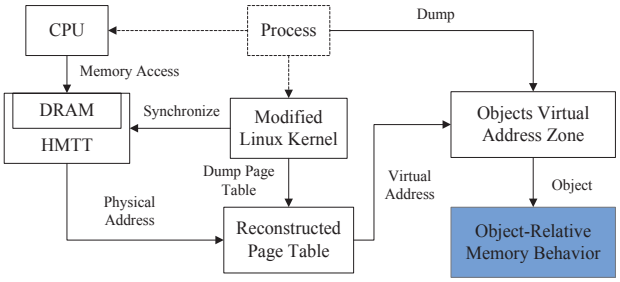

Figure 2. The framework of ORMBP.

In this section, we first give a brief introduction to the framework of ORMBP (Object-Relative Memory Behavior Profiling), and then we introduce the detailed implementation of the main components in subsections.

Figure 2 shows the framework of ORMBP. The ORMBP adopts a hybrid hardware/software approach and it mainly consists of three components. In terms of hardware, the HMTT [3] card monitors the memory access requests to the DRAM system and dumps all physical memory address traces. In terms of software, we modify the Linux kernel to dump page table traces, which will be used to reconstruct reverse page table (RPT). The virtual address of a memory trace could be extracted from the RPT with the physical address captured by HMTT. Additionally, we provide a software toolkit for dumping object-relative virtual address zone during the execution of the process. With this information we can identify each memory request trace belonging to which object, thus obtain the object-relative memory behavior of the process.

A. HMTT

HMTT [3] is a DDRx SDRAM compatible memory trace monitoring system, which is able to track full-system physical/virtual memory reference traces (including OS, VMMs, libraries, and applications). It acts as a DIMM adaptor, which is directly plugged into a DIMM slot of a motherboard and a DRAM DIMM is plugged onto the HMTT card. This allows HMTT to monitor memory transaction signals on the transfer wires, reconstruct memory references, and send the memory trace out to another receiver PC. Since the transfer wires are very short, very little electrical interference is introduced to the system.

By now, the third version of the HMTT has been released[1], which supports DDR3-800 UDIMM/RDIMM. It adopts a high speed PCIE1.0 cable to transfer memory traces out to a receiver PC, the transfer bandwidth can achieve up to 8Gbps. The physical memory trace has the following format: <*seq_no*, *duration*, *r/w*, *paddr*>. Where *seq_no* is the sequence number of the PCIE packets, which is used to check whether packets are losing during transferring; *duration* is the HMTT hardware clock cycles between this memory reference trace and the previous trace, the HMTT clock is fixed at 400MHZ, so we can get accurate issue-time for each memory trace; *r/w* is a bit indicating whether it is a read (when 1) or a write (when 0) memory request; *paddr* is the physical cache block address of the memory request.

[1] The homepage of HMTT is: http://asg.ict.ac.cn/hmtt/

B. Dump Page Table Trace

Figure 3. The splitting of the x86-64 linear address.

Linux kernel has supported four-level radix tree page table structure since 2.6.11, they are Page Global Directory (PGD), Page Upper Directory (PUD), Page Middle Directory (PMD) and Page Table (PT) [6]. Thus a linear address is split into five parts. By now, in Intel x86-64 architecture, the ordinary page size is 4KB, and the linear address splitting is (9, 9, 9, 9, 12), as shown in figure 3.

Each page directory entry contains the physical page frame number for the next lower-level page directory and each page table entry contains the physical page frame number for the data page (and some other page attributes). Thus the physical page frames in the system can be split into two categories: **the data pages** and **the page table pages**. The data page is used for real data storage of a process, and the page table page is used for page table storage of a process. When a last level cache (LLC) miss due to a load/store operation occurs, it will send a memory request to a data page (known as normal data access). And when a last level Translation Lookaside Buffer (TLB) miss occurs (assume it also miss in cache), it might need at most 4 memory requests for the page table pages (known as page memory walks) to get the physical page frame number for the request virtual address. When the HMTT monitors a memory access request that is for a page table page, it can distinguish that this memory request is a page memory walk due to TLB miss. And other requests for the data pages are identified as normal data access LLC miss.

Figure 4 shows the overall structure of DPTC (Dump Page Table Component). We first modified Linux kernel to monitor each page table update operation and dump the page table info into the page table buffer through the interface provided by the Dump Module. The Dump Module is implemented as a kernel module which provides mainly two functionalities: (1) it provides the interface for the modified kernel to dump page table info into the page table buffer and meantime to send a synchronous memory access to HMTT card when a page table update happens (we will discuss the synchronization with HMTT later); (2) it provides the interface to the Control Component for controlling starting and stopping of dumping page table traces. The Control Component is a user level process which can control starting and stopping of dumping page table traces flexibly by sending commands to the Dump Module. With this control mechanism, we can just dump page

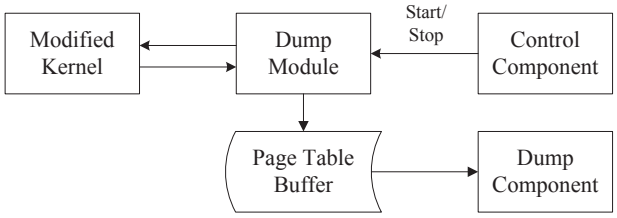

Figure 4. The overall structure of DPTC.

table traces for the region we are interested in and significantly reduce the amount of traces. The Dump Component is also a user level process to dump the page table traces from the page table buffer into a file periodically. By providing an independent process, the runtime overhead can be reduced greatly.

To dump the page tables of processes, the kernel first goes through the whole original page table of each process (or just the processes we are interested in), and dump each page entry (including page directory entry and page table entry) to the Page Table Buffer. Since the page table would change dynamically during the lifetime of a process, for example allocate some new pages at some time and then free them later. In response, the kernel would set some new page table entries for the new-allocated pages and then clear them from the page table. Thus the kernel needs to monitor all the page table updates and dump each page table update to the buffer. This can be done by overwriting the page table update functions (*native_set_pud*, *native_pud_clear* etc.). For page directory updates, it only needs to dump the physical page frame number and the process id (*pid*); for page table entry updates, besides the above information, it also needs to dump the corresponding virtual address, which will be used to reconstruct the reverse page table (RPT).

The reverse page table is reconstructed according to the page table traces dumped. Figure 5 shows the reconstructed reverse page table, it has N buckets, where N is the available physical page frame number in the system. Since a physical page frame might be shared by multiple processes (such as shared memory segments), each bucket is organized as a link list with each list entry containing the corresponding virtual address and the pid of each process. The reverse page table uses physical page frame number for indexing. Given a physical address captured by HMTT and a specified process id, one can extract the corresponding virtual address from the reverse page table in two steps: (1) calculate the physical page number from the physical address, use it as an index to get the bucket; (2) access each list entry of the bucket until finding the specified process id. To improve the searching performance in the reverse page table, we have implemented each bucket with an AVL tree (a self-balancing binary search tree) instead of a link list.

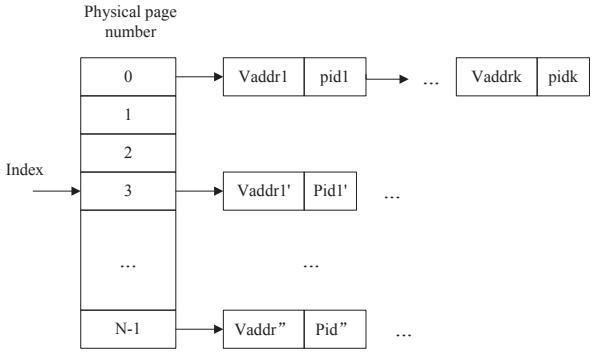

Figure 5. The reconstructed reverse page table.

C. *Synchronizing the HMTT Traces with Kernel Page Table Traces*

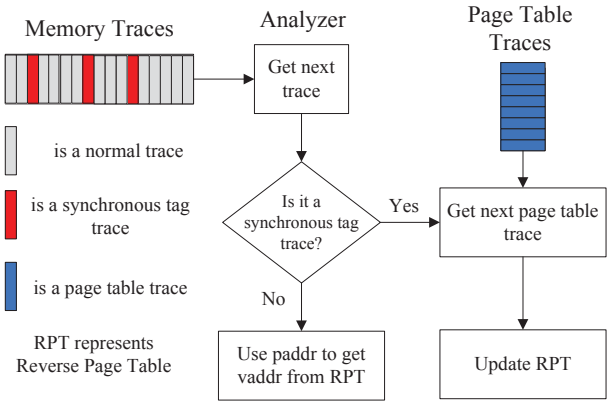

Figure 6. Synchronization HMTT memory traces with page table traces during trace analyzing (paddr represents physical address and vaddr represents virtual address).

Because HMTT has a fixed frequency clock and each memory trace has a duration field, it is appropriate to use this clock to synchronize the HMTT traces with kernel page table traces. At each page table update, the Dump Module will send an uncacheable memory access request to the HMTT, which we call a synchronous tag trace. We reserve some physical memory (nearly 32MB) for synchronous memory access, and then *ioremap* it into the kernel space to make it accessible with *nocache* flag set. This guarantees that the reserved memory space could only be accessed by the Dump Module for synchronous tag accesses and prevents interfering by others (such as kernel threads and other processes).

Figure 6 shows the synchronization of HMTT memory traces with page table traces during trace analyzing. The HMTT memory traces contain both normal traces and synchronous tag traces. Each time, the analyzer gets the next trace form HMTT memory traces, it first needs to check whether it is a synchronous tag trace or not. If so the analyzer gets the next page table trace from the kernel page table traces and update the reconstructed reverse page table (i.e, set an entry or clear an entry), this makes the reverse page table keep in correct state. If not, it is just a normal memory trace, the analyzer will use the physical address of the trace to extract (or query) the corresponding virtual address from the reverse page table.

D. *Dump Objects Virtual Address Zone*

In a program, there are two different types of objects: static objects whose space is automatically managed in the data segment, and dynamic objects whose space is manually allocated in the heap. For static objects, we can get the entry address and the size of them from the symbol table of the execution file (it is an ELF format file in Linux system). For dynamic objects, we develop a software toolkit which uses the *LD_PRELOAD* environment variable in Linux system to overlay the *malloc()* and *free()* functions to record the entry address and size of each dynamic object. The overlay operation might introduce some overhead to the application. We will evaluate the overhead in section IV. As like dumping page

table, we need to synchronize the object address zone information with the HMTT traces, this is also done by sending a synchronous tag trace at each interested object allocation, thus we can get the time-stamp information for each object allocation.

Furthermore, if the source code of the program is provided, we can associate the object information with the main variables in the program, which is valuable for programmers to identify the performance bottleneck in variables and to optimize these variables (that is what we will do in case studies). Even without source code, it is easy to associate the dynamic object with calling address for further analysis by backtracking the call stack.

It should be noted that in our approach the object is not limited to user space. In fact with careful annotation, memory access to kernel object can also be identified without difficulty. Page memory walks and DMA buffers [32] are such examples. Considering both kernel and user space memory behavior may help to profile those applications using kernel service intensively.

The memory trace can be extended to the following format:

<**paddr**, **r/w**, **time-stamp**, **pid, vaddr**, **object**, **page-walk**>

The latter four fields are semantic information for the program. The *Pid* field is the id of the process issuing this memory request; *vaddr* field is the virtual address in the process virtual space for the memory request; *object* field indicates the object issuing the memory access request; *page-walk* is a flag indicates whether the memory request is a page memory walk due to TLB miss or not. Thus with the object-relative memory trace, we can:

1) Separate the memory traces of a process into objects, and then analyze the memory access pattern for each object.

2) Analyze the percentage of memory requests for each object, and find the hot objects which contribute the most of the memory requests. These provide valuable information for program optimization and compiler optimization.

3) Analyze the percentage of page memory walks for each object, and find the hot objects for page memory walks. These can be used to optimize TLB performance.

IV. Experiments and Case Studies

A. Experiment Setup

We carried out our experiments with two 2.00GHz Intel Xeon E5504 processors. Each E5504 processor has 4 physical cores and we have disabled the Hyper-Thread. There are 3-level caches in each processor, the L1 instruction and data caches are 32KB each and the L2 cache is 256KB, both the L1 and L2 are private in each core. The L3 cache is 16-way 4MB and shared by all four cores in each processor. The cache line size is 64B for all caches in the hierarchy. There are 2-level private TLB in each core, the L1 DTLB has 64 entries for 4KB pages and 32 entries for 2MB pages (huge-page or huge-tlb), and the second-level TLB has 512 entries for 4KB pages. The total capacity of physical memory is 4GB with one dual-ranked DDR3-800 RDIMM, and the peak memory bandwidth is 6.4GB/s. We reserved 0.25GB memory space for HMTT configuration space and page table buffer, thus the actual system available memory is 3.75GB. The operating system is 64-bit CentOS5.3 for x86_64, the Linux kernel is 2.6.32.18 which is modified for dumping page tables. The compiler is gcc 4.1.2. The C language library is glibc 2.5.

As case studies, we choose a custom hybrid MPI/pthread implemented BFS of Graph500-1.2 benchmarks [1] and the canneal program from PARSEC-2.1 benchmarks [5]. The BFS default implements 64 times of Breadth-First Search from different randomly generated root vertexes on a Recursive MATrix (R-MAT) scale-free graph. Since we just perform the object-relative memory profiling for the BFS on one node, we run the BFS with one process and multiple threads. The canneal program we use is pthread implementations and with native input. Both the programs in our experiments are compiled with -O3 optimization. We run each program three times with different number of threads respectively.

B. Validation

We used a micro-benchmark to validate our hybrid hardware/software approach for object relative memory profiling. The benchmark allocates 5 arrays (they are *a0*, *a1*, *a2*, *a3* and *a4*), the size of each array is 256MB. After initialize it with memset (to make sure that the physical pages are all allocated for them), we traverse each array with a step of 64 bytes and different read/write rate access pattern:

- a0: all read accesses, forward
- a1: 3/4 read and 1/4 write accesses, forward
- a2: 2/4 read and 2/4 write accesses, forward
- a3: 1/4 read and 3/4 write accesses, backward
- a4: all write accesses, backward

We set the access step as 64B which is the size of a cache line, to make sure that each access is not created in the same cache line as the previous access, thus each access would cause a cache miss and perform a memory request to the DRAM. Since the size of each array is 256MB, the number of memory requests for each array is 256MB/64B=4M=4,194,304. We respectively create a thread for each array traverse with *pthread* in Linux with different read/write rate access pattern, for example, for a1, we performed 3M read accesses to the first 3/4 elements and then 1M write accesses to the last 1/4 elements. We ran the five threads simultaneously in one process. Further, to distinguish each array access pattern, we forward traversed on a0, a1, a2 and backward traversed on a3, a4.

It is noteworthy that the L3 cache adopts write-back policy, so when a memory write access requests a line not in the L3 cache, it first sends a **read request** to the memory to get the line back into the L3 cache, then writes data to the line, and writes back the data to the memory when the line is evicted out (this is done by sending a **write request** to the memory). Since we make each write access miss in the L3 cache, each write access will cause one read memory request and one write memory request. So for all a0~a4, each of the arrays will send 4M read requests and different numbers of write requests.

TABLE II. THE OBJECT RELATIVE MEMORY PROFILING RESULTS OF THE MICRO-BENCHMARK

Obj	Read	Write	Rate	Per	Error
a0	4,194,370	0	4:0	4:0	0%
a1	4,194,310	1,048,576	4:1	4:1	0%
a2	4,194,369	2,096,927	4:2	4:2	0%
a3	4,194,303	3,087,379	4:2.94	4:3	2.04%
a4	4,194,436	4,149,586	4:3.96	4:4	1.01%

Table II shows the object relative memory profiling results of the micro-benchmark. The *Read/Write* column represents the number of read/write memory requests of each array. The *Rate* column represents the rate of read/write requests, the *Per* column represents the perfect rate of read/write requests, and the *Error* column shows the error of the object-relative memory profiling. We can see that, each array has nearly 4M read memory requests, which is the same with our analysis above. And from a0 to a4, each array performs nearly 0, 1M, 2M, 3M, 4M write requests respectively, and they all conform with the access pattern. Finally, the largest error of the profiling is 2.04%. It shows our object-relative memory profiling has highly accuracy. The possible reasons causing the error include: invalid hardware prefetch and DRAM refresh etc.

Figure 7 shows the virtual memory traces of the object *a0* and *a4* of the micro-benchmark. We can see that the virtual memory traces of the object *a0* increase and the virtual memory traces of the object *a4* decrease, because we forward traverse on object a0 and backward traverse on object a4. This test shows that our approach can get accurate complete virtual memory traces for major objects of a program.

We also performed object-relative memory profiling on serial version SpMV (Sparse Matrix-Vector multiplication), a program to multiply a sparse matrix (in CSR format) with a dense vector. The non-zero elements in the matrix are stored in array *ax*, and the vector is stored in array *xhost*. Figure 8 and figure 9 illustrate that with our object relative memory profiling, we can decompose the regular access pattern of *ax* object from the irregular access pattern of *xhost* object. The virtual memory address traces appear periodicity because we ran the multiply operation multiple times. This test illustrates that different objects may have different memory access patterns that can be investigated by our approach.

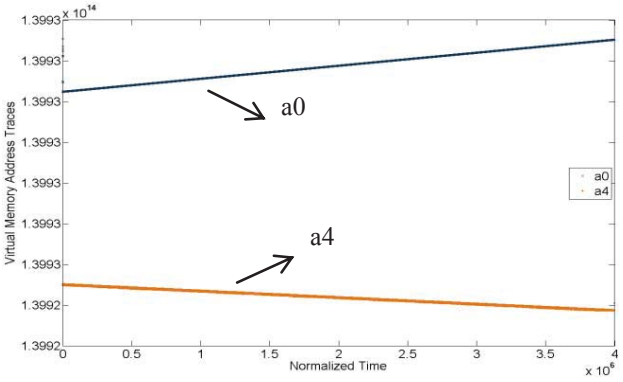

Figure 7. The virtual memory traces of the object a0 and a4 in the micro-benchmark.

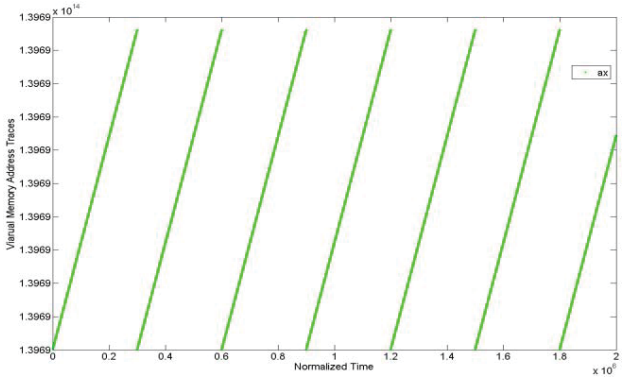

Figure 8. The regular access pattern of the *ax* object in SpMV.

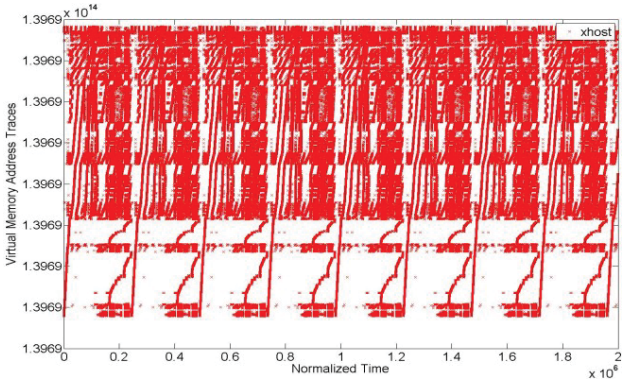

Figure 9. The irregular access pattern of the xhost object in SpMV.

C. Overhead

In this subsection, we measure the overhead of object-relative memory profiling. We show that with our hybrid hardware/software approach, the overhead is quite lightweight. The overhead mainly includes two parts: dump page table in kernel and dump object-relative memory allocation information. In this subsection, we run all the programs with 8-thread to measure the overhead.

Figure 10 shows the overhead of the object relative memory profiling, the *original*, *+dump_pt* and *+dump_obj* each respectively represents the original run time of the program, the run time of the program just with dumping page table in kernel, and the run time of the program with dumping page table and dumping object relative memory allocation information[2]. Since the main memory accesses of a multi-thread program are on a few large objects, we only need to monitor these objects relative memory allocation information. In our experiments, we choose to only monitor objects which are larger than 4KB in memory profiling. We can see that the average overhead of dumping page table in kernel is about 0.66% and the average overhead of dumping object relative memory allocation information is about 1.60%. The largest overhead of dumping object relative memory information is 5.00% for the dedup, this is because the number of object allocations of dedup is large. Actually there are total 1,240,324 dynamic memory allocations with size larger than 4KB on the heap during its execution.

[2] Since there is negligible overhead of monitoring the physical memory traces with HMTT (hardware), we do not show it here

TABLE III. THE PERCENTAGE OF MEMORY REQUESTS FOR MAIN OBJECTS IN BFS AS THE NUMBER OF THREADS INCREASING

Threads	column_r	column_w	visited_r	visited_w	rowstarts_r	rowstarts_w	pred_r	pred_w	total
1	65.71%	0.00%	5.21%	0.87%	7.79%	0.00%	8.43%	8.44%	96.46%
2	64.96%	0.00%	5.36%	1.32%	7.72%	0.00%	8.35%	8.35%	96.06%
4	55.19%	0.00%	16.01%	4.54%	6.57%	0.00%	7.10%	7.06%	96.46%
32	55.13%	0.00%	17.11%	5.28%	6.58%	0.00%	7.10%	7.06%	98.24%
128	54.22%	0.00%	18.38%	5.27%	6.47%	0.00%	6.98%	6.93%	98.26%

(*_r represents read memory requests and *_w represents write memory requests)

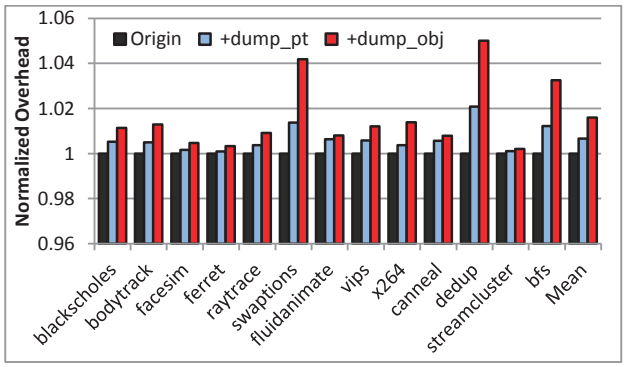

Figure 10. The overhead of the object relative memory profiling.

As shown above, if the number of object memory allocations (>=4KB) is large, the overhead would increase. To further reduce overhead, we could set a larger threshold or we could choose to only monitor the top 10% largest objects of some complex programs.

Dumping page table and object memory allocations information during the running of a program would introduce some cache/memory interference, but the interference is very low. Because in our experiments, we find that the size of page table traces and object memory allocation traces is quite small relative to the size of the normal memory access traces. For example, running 8-thread canneal with native input set, we get 200GB memory access traces with only 6.2MB page table traces and 26.8KB object allocation traces; running 8-thread streamcluster with native input set, we get 104GB memory access traces with 6.7MB page table traces and 27.4KB object allocation traces.

D. Case Studies

1. BFS in Graph500

Graph 500 [1] is a set of large-scale benchmarks for data-intensive applications, especially graph algorithms for analytics workloads. Breadth-First Search (BFS) is the most common and important operation in graph algorithms, which is the basis of many other graph operations. Graph 500 adopts synthetic kronecker graphs whose degrees follow power law distributions, which means that most of the vertices has a small degree (the number of edges associated with the vertex) and only a small portion of the vertices has huge degrees. Thus the graph is represented as compressed sparse row (CSR) format in order to save space. In a CSR format graph, the main data structures (objects) are *column* and *rowstarts*. The *column* object stores the adjacency array for all the vertices one by one and the *rowstarts* object stores the start index for each vertex's adjacency array in the column object. In each BFS, the *visited* object is a bitmap for each vertex indicates whether the vertex has been visited or not, if visited, then there is no need to revisit the vertex's adjacency array. The *visited* object improves the BFS performance significantly and needs to be accessed multiple times for each vertex. The *oldq* and *newq* objects are both simple FIFO queue structures, which store the vertices to be extended at this level and the next level. The *pred* object is used to store the parent vertex for each vertex during a BFS. The scale of the graph we choose is 23 and the edgefactor is 16 (default), which mean that the number of the vertices in the graph is 2^{23} and the mean degree of each vertex is 16. The total size of the CSR graph is nearly 2GB. The *visited* object is 1MB, which is smaller than the LLC capacity (4MB).

The BFS program mainly contains 4 steps: construct a graph, run BFS for each root (source vertex), validate, and compute the performance information. Only the running BFS step is counting for the performance, so we just perform the object relative memory profiling for this step.

Table III shows the object relative memory profiling results of the BFS with the number of threads increasing. We can see that the *column* object contributes the largest portion of memory requests, it has a percentage of 65.71% for 1-thread and 54.22% for 128-thread, and it is read-only, the write memory requests is always 0%. The percentage of memory requests of the *column* object decreases with the number of threads increasing, but the percentage of *visited* object increases from 6.08% for 1-thread to 23.65% for 16-thread, we will explain the reason for it latter. The *total* column represents the percentage of the total memory requests of the main objects relative to the total memory requests of the whole BFS program. We can see that we successfully capture the main objects contributing to the memory requests, the percentage is larger than 96%.

Figure 11 shows the normalized memory requests (including read and write) number of each main object in BFS against increasing the number of threads. We can see that except for the *visited* object, the numbers of memory requests of all the other main objects are nearly constant (the largest increasing rate is 0.59%). But for the *visited* object, the number of memory requests increases by 79.04% for 2-thread, 547.81% for 4-thread, and 662.27% for 128-thread (normalized to 1-thread). That is because the *visited* object needs to be accessed every time when a vertex is extended, to check whether the vertex has been visited, so the number to be accessed during

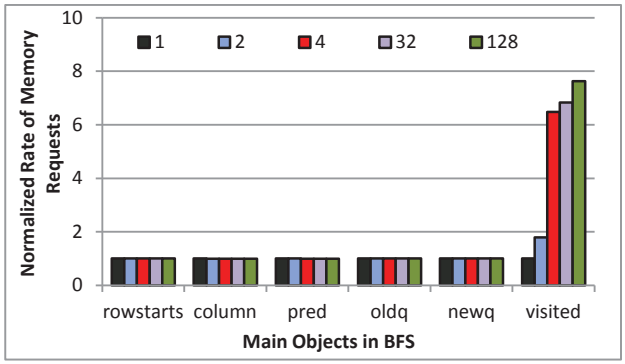

Figure 11. The normalized memory requests of each main object in the BFS against increasing the number of threads, where the baseline is 1-thread.

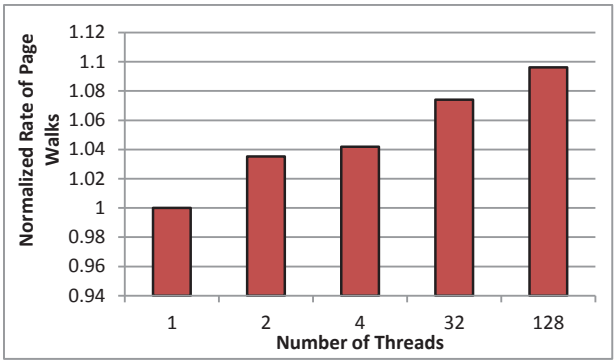

Figure 13. The normalized rate of the number of page memory walks due to TLB miss as the number of threads increasing, the baseline is 1-thread.

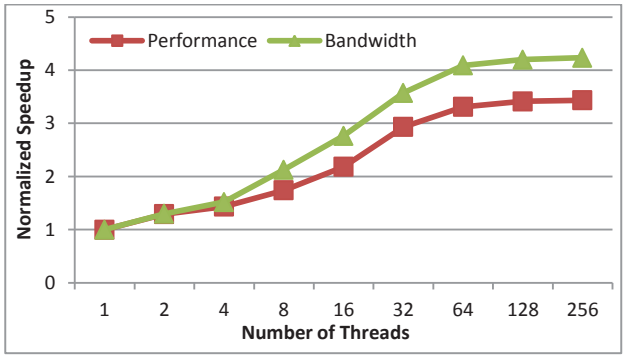

Figure 12. The normalized speedup of the performance and the memory bandwidth against the number of threads, where the baseline is 1-thread.

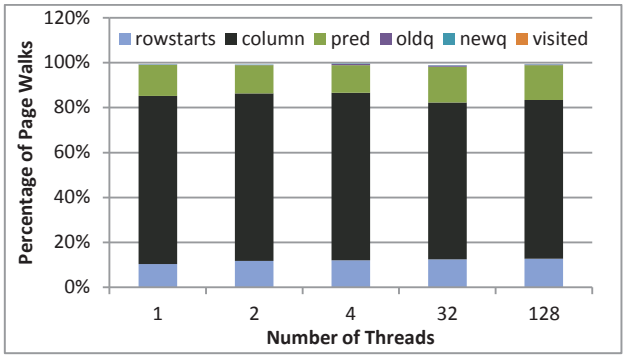

Figure 14. The percentage of page memory walks for main objects in BFS as the number of threads increasing.

each BFS depends on the degree of the vertex. Since the *visited* object (1MB) is less than the L3 cache (4MB), for 1-thread when the L3 cache contentionis minor, the *visited* object has quite a good cache locality (only 6.08%). But with the number of threads increasing, the access pattern of the *visited* object becomes more random (each thread needs to access it) and the shared L3 cache contention becomes more serious among these threads, which results in more L3 cache misses of the *visited* object during the BFS. For other objects, with the filtering effect of the *visited* flag, each element is actually accessed only once, so the access pattern of these objects are affected slightly by the increasing L3 cache contention as the number of threads increasing.

Figure 12 shows the normalized speedup of the performance (in Traversed Edges Per Second, TEPS[1]) and the memory bandwidth against the number of threads. We can see that the performance and the memory bandwidth both increases as the number of threads varied from 1 to 256, and it has 3 stages with different characteristics. The first stage is from 1-thread to 4-thread, the normalized performance increases nearly at the same rate as the memory bandwidth, so the performance linearly benefits from the increasing of the memory bandwidth. The average normalized speedup is about 1.2 as the number of threads doubled. The second stage is from 4-thread to 64-thread, the average normalized speedup of the memory bandwidth is about 1.28 as the number of threads doubled, and the average speedup is about 1.23 of the performance. Thus in this stage, the performance does not fully benefit from the increasing memory bandwidth, that is because the shared L3 cache contention increases as the number of threads increasing and causing more L3 cache misses. According to figure 11, the *visited* object suffers the most of the increasing L3 cache miss due to the more serious L3 cache contention, which could significantly affect the performance. The last stage is from 64-thread to 256-thread, the normalized speedup of the memory bandwidth is increasing very little (only 0.018), that is because BFS has an irregular memory access pattern and it has achieved the memory bandwidth limitation at this stage. For 256-thread in our experiments, the memory bandwidth achieved 4.66GB/s, which is 72.81% of the peak memory bandwidth (6.4GB/s). Because of the random distribution of nodes in the CSR graph, lock contention among threads and the synchronous overhead at each search level, this should be the maximum memory bandwidth it could achieve. In this stage, the performance also increases little (only 0.019) due to the little increasing memory bandwidth.

And for more than 1024-thread (it is not shown in figure 13), the performance will decrease due to the serious cache contention. So increasing the number of threads on one hand can improve memory bandwidth, but on the other hand will make the L3 cache contention more serious, especially for the *visited* object. There is a tradeoff to choose the proper number of threads. To reduce the cache contention on *visited* object, we can adopts page coloring technology [12][17] and protect the *visited* object with enough colors.

As figure 13 shown, the number of page memory walks due to TLB miss also increases with the number of threads

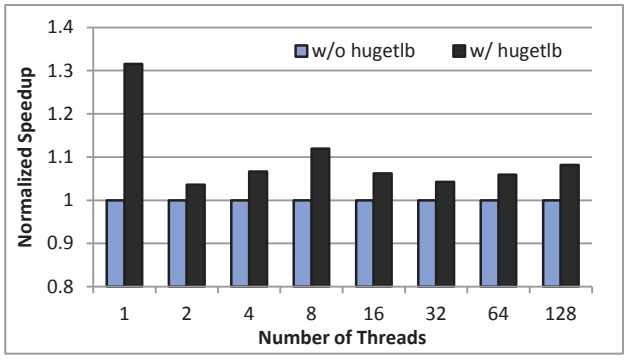

Figure 15. The normalized performance speedup with huge pages (hugetlb) as the number of threads varied from 1 to 128.

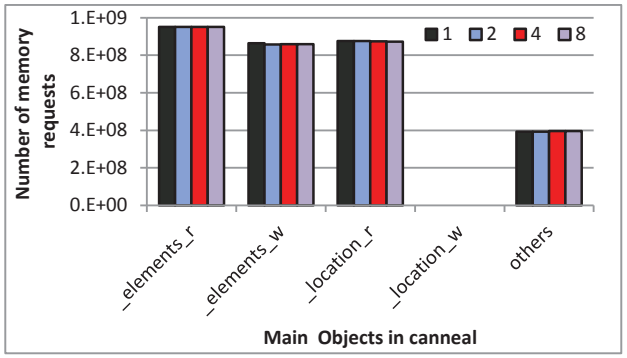

Figure 16. The number of memory requests for the objects in canneal as the number of threads varied from 1 to 8.

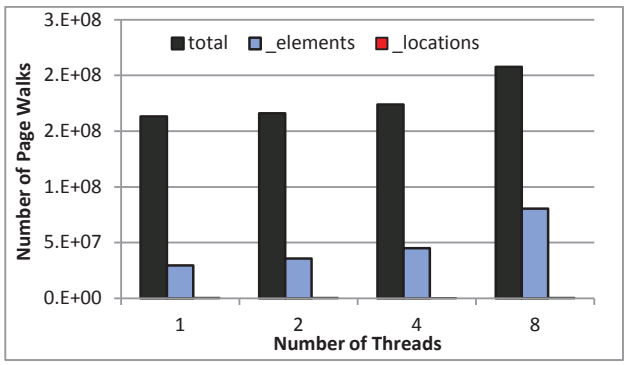

Figure 17. The number of page memory walks for each object in canneal as the number of threads varied form one to eight. The total indicates the total number of the program.

increasing, it increase 3.5% for 2-thread and 9.6% for 128-thread. Figure 14 shows the percentage of page memory walks for each main object. We can see that the percentage for each main object nearly maintains constant among different number of threads. The *visited*, *newq* and *oldq* objects exhibit nearly no page memory walks (nearly 0%), that is because the visited object is quite small (1MB) and it can reside in the TLB; the *newq* and *oldq* objects are accessed with a nearly contiguous pattern which would cause little TLB miss. The other three objects contribute the most of the page memory walks; the average percentages are nearly 11.92% for *rowstarts* object, 72.85% for *column* object, and 14.02% for *pred* object. The *column* object suffers the most of page memory walks, because the *column* object is the largest data structure in the BFS program (nearly 2GB), and it is accessed in a inter-vertex random and inner-vertex continuous manner.

To reduce the number of page memory walks caused by the column object, we can put the column object into huge pages (2MB) to reduce the TLB miss [3]. Figure 15 shows the normalized performance speedup with the huge pages (or hugetlb). It can achieve up to 31.59% performance improvement for 1-thread, but much lower for multiple threads, 8.18% for 128-thread. That is because with the number of threads increasing, the *column* object is accessed more randomly, which would result in more TLB contention. But on the other side, for BFS, increasing number of threads could achieve better load-balance, and decrease the level synchronous overhead among threads. Thus we can see the different normalized performance speedup under different number of threads with huge page.

2. Canneal

Canneal benchmark from PARSEC uses cache-aware simulated annealing (SA) to minimize the routing cost of a chip design [5]. In the experiments, we ran the pthreads parallelization version benchmark with native input, and performed the object-relative memory profiling in the Region-Of-Interest (ROI).

There are two main data objects which contribute most of the memory accesses. The _elements object stores the actual elements, each of which contains a name and all the fanin and fanout elements in the netlist. The _locations object stores the actual locations, and each location is a two-dimensional coordinates. Each time it randomly chooses two netlist elements and tries to swap them with a cost-related probability to minimize routing cost. Thus the _elements object needs to be updated (written) frequently (when a swap happens), and the _locations object only needs to be read to calculate the new cost.

Figure 16 shows the number of memory requests for each main object as the number of threads is varied from one to eight. The *_elements* object has mostly memory write requests and the *_locations* object is read-only, there are no write requests on it. Further, we can see an interesting phenomenon that all the objects has nearly the same number of the memory read and write requests in different number of threads (1 to 8). And the average percentage of memory read/write requests for main objects are: 30.87% for *_elements* read requests, 27.92% for *_elements* write requests, 28.39% for *_locations* read requests, 0% for *_locations* write requests, and 12.82% for others memory requests.

Figure 17 shows the number of page memory walks for each object as the number of threads increasing. The total number of the page memory walks for the program increases form 1.58E08 for 1-thread to 2.15E08 for 8-thread, meanwhile, the number of page memory walks for *_elements* object is increasing from 2.96E07 for 1-thread to 7.95E07, and the *_locations* object has almost no page memory walks. Thus in

[3] Since the number of TLB entries is limited for huge pages (32 in Nehalem), we just put the *column* object in huge pages to avoid contention for it among multiple objects

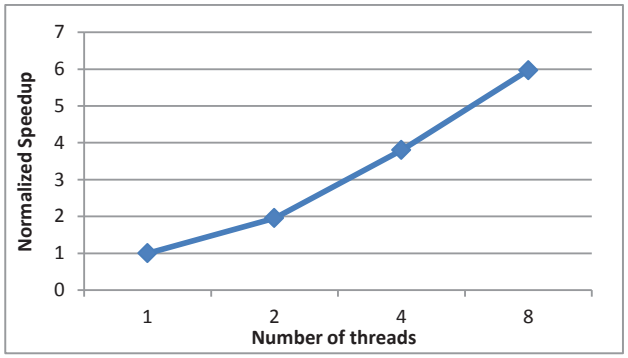

Figure 18. The normalized performance speedup as the number of threads varied form one to eight.

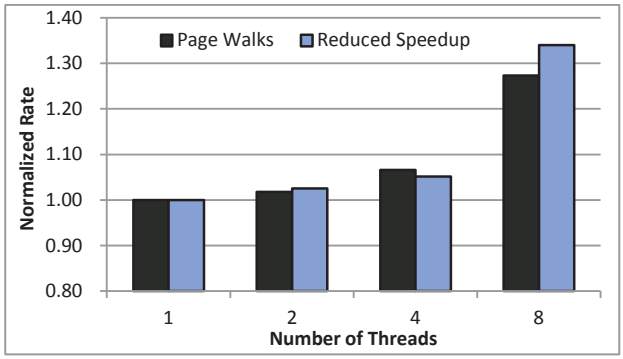

Figure 19. The normalized rate page memory walks and the reduced speedup as the number of thread varied from 1 to 8.

canneal, the *_elements* objects is the main reason for the increasing of the page memory walks as the number of threads increasing. For 8-thread, the percentage of the page memory walks is up to 25.84% for *_elements* object.

Figure 18 shows the normalized performance speedup as the number of threads increasing. The normalized speedup is 1.95 for 2-thread, 3.80 for 4-thread, and 5.97 for 8-thread. Here we define reduced speedup as the rate of the peak speedup (the number of the threads) relative to the actual speedup, thus the higher the reduced speedup means the poorer scalability. For instance, for 2-thread, the reduced speedup is 2/1.95=1.03.

Figure 19 shows the normalized rate of the number of page memory walks and the reduced speedup, we can see that the reduced speedup has a strong correlation with the number of page memory walks as the number of threads increasing. For 8-thread, the normalized rate of the page walks is 1.27 and the reduced speedup is 1.34. According to figure 16, the number of the memory requests is almost no change, thus here we can conclude that the main reason for the reduced speedup is the increasing page walks. Furthermore according to figure 17, the *_elements* object is the real source.

Thus we put the _elements object into huge pages to optimize its TLB performance. As figure 20 shown, with huge page, the normalized performance speedup is 1.07 for 1-thread, and 1.055 for 8-thread. The speedup is reduced due the more serious TLB contention as the number of threads increasing.

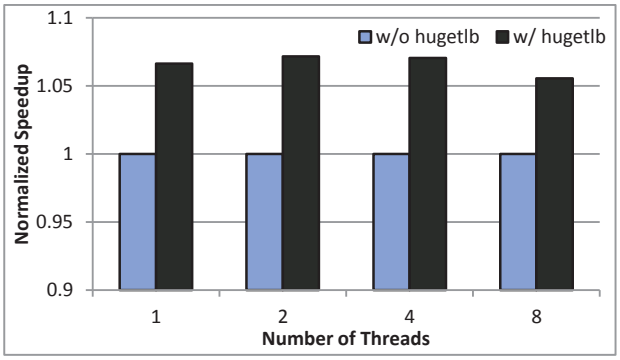

Figure 20. The normalized performance speedup with huge pages as the number of threads varied from one to eight of the dedup program, the baseline is without huge pages.

V. Related Work

There are several areas of effort related to memory trace profiling: binary rewriting, Instrumentation, Simulation, and hardware performance counter etc.

Binary rewriting: The METRIC [20] system exploits dynamic binary rewriting for partial memory reference tracing. It also compresses the traces and employs offline cache simulation to find out the memory performance bottlenecks. Ben-Asher et al. [2] employ source code instrumentation with the LLVM compiler framework to collect memory traces, and the memory traces are used to guide automatic data structure partitioning to increase memory parallelism. Weinberg et al. [39] gather memory access characteristics using a binary rewriting tool and then use the traces to quantify the spatial locality and temporal locality for HPC applications.

Performance Counter: Itzkowitz et al. [14] exploits UltraSPARC-III hardware counters for memory behavior profiling, however the hardware counters can just provide event count without the memory address and instruction. Thus they employ some extensions of the Sun ONE Studio compilers further more to provide per-instruction details of memory accesses, data aggregated and sorted by object structure types and elements. Buck et al. [8] presents a tool named Cache Scope on the Itanium 2 which exploits performance counter to sample cache miss address and then performs the mapping of addresses to objects for variables by using the debug information. Eranian [13] argues that the performance counters are the key hardware resource to locate and understand memory subsystem performance with low overhead. DProf [27] is a data-oriented profiler which can attribute cache misses to data types instead of code. DProf uses performance monitoring hardware (AMD instruction-based sampling hardware and x86 debug registers) to acquire memory address references, which is as the basis of categorizing all types of cache misses. Oprofile [16] is a system-wide profiler for Linux systems at low overhead, it samples hardware performance counters of the CPU to enable profiling of a wide variety of interesting statistics and reports the cost of each function.

Instrumentation: Pin [19], DynamoRIO [7], Valgrind [24] and ATOM [31] are widely used binary instrumentation tools, and they are often used for memory profiling, the main

drawback of these popular instrumentation tools is heavy overhead. SIGMA [11] is a data collection framework that uses software instrumentation to gather memory addresses during the running of a process, it also provides a number of simulation and analysis tools to provide detailed information on variables and functions. To reduce instrumentation overhead, Ephemeral instrumentation [35] uses statistical sampling, shadow profiling [23] and SuperPin [36] execute instrumented code in parallel with a program's execution to perform instrumentation sampling for multi-core system. Zhang and Gupta [41] exploit statically instrumenting to collect whole execution traces (including memory address traces) with an efficient compression mechanism. Weinberg et al. [38] use binary instrumentation to reduce memory tracing overheads without significant loss of accuracy with basic block sampling techniques. Roy et al. [29] present a hybrid static and dynamic binary rewriting instrumentation, the instrumented code is written into the PIC in active mode and then executed out of the PIC in passive mode to reduce overhead. Marathe et al. [21] present two hybrid hardware performance counters with software instrumentation techniques to determine cache coherence bottlenecks in shared-memory applications.

Simulation: MemSpy [22] is a tool for locating memory system bottlenecks with detailed statistics on low-level memory system events, which uses a software memory simulation in monitoring the memory system behavior of programs on both data- and code-oriented information. Weidendorfer et al.[37] use runtime instrumentation and cache simulation to analyze memory access behavior. SIMT [33] is an execution-driven simulator focus on the memory performance and contains mainly facilities for simulating the memory system.

Torrellas et al. [34] present a similar hybrid hardware/software approach for characterizing cache performance of multiprocessor OS. However their hardware monitor relied on MIPS buses which were proprietary and the software implementations were totally different from ours. Actually, our work is more portable than their approach which depends on MIPS' architecture. For example, their approach leveraged MIPS' software-managed TLB to track physical-virtual address mapping. But this approach is not suited for contemporary prevalent x86 platforms which use page walks to refill TLB. Therefore, our Linux-based software adopts monitoring page-faults to track physical-virtual address mapping.

Wu et al. [40] present an object-relative translation for effective memory profiling instead of raw memory addresses, which enables decomposition of a memory access stream to separate regular and interesting information from the irregular. However in their work, the raw memory profiling is implemented by instrumentation and it will suffer heavy overhead. Our hybrid hardware/software approach can significantly reduce the memory profiling overhead and get object-relative information for each memory trace.

Actually, detailed object-level memory profiling is valuable for performance optimization, such as it can be used to optimize object-level cache partition through well known page coloring technology in Soft-OLP [18]. However, since they adopted dynamic instrumentation (with Pin) for memory profiler, the overhead was heavy: the slowdown was 50 to 80 times even with 10% sampling. With our hybrid hardware/software memory profiler, we could also get full detailed and accurate shared cache misses on a set of objects for a given cache size directly without cache model estimation [18], and the overhead is lightweight (no more than 6%).

VI. CONCLUSIONS

In this paper, we have proposed a hybrid hardware/software approach for object relative memory profiling, which is able to not only profile the object level memory traces of data access, but also identify the page memory walks due to TLB miss at object level. It adopts hardware snooping technology, modifies the kernel to support dump page table, and dumps object virtual address range during the running of a program. We present some case studies to show that with our approach, we can effectively identify the memory and TLB performance at object level, which is valuable for optimization.

ACKNOWLEDGMENT

The authors thank the anonymous reviewers for their constructive suggestions. They thank Huiwei Lv from ASG, ICT for providing the pthread-based Graph500 BFS benchmark. This research is supported by the National Natural Science Foundation of China (NSFC) under grant numbers 60925009, 60921002, 60903046, 60803030, 61033009 and the National Basic Research Program of China (973 Program) under a grant number 2011CB302502, and by IBM SUR Awards program.

REFERENCES

[1] The Graph 500 List, 2011. URL http://www.graph500.org/.

[2] Yosi Ben-Asher and Nadav Rotem. 2010. Automatic memory partitioning: increasing memory parallelism via data structure partitioning. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis* (CODES/ISSS '10). ACM, New York, NY, USA, 155-162.

[3] Yungang Bao, Mingyu Chen, Yuan Ruan, et al. HMTT: a platform independent full-system memory trace monitoring system. in *ACM SIGMETRICS*, 2008.

[4] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems* (ASPLOS XIII).

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[6] Daniel Bovet , Marco Cesati, Understanding The Linux Kernel, Oreilly & Associates Inc, 2005

[7] Derek L. Bruening. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA. AAI0807735.

[8] Bryan R. Buck and Jeffrey K. Hollingsworth. 2004. Data Centric Cache Measurement on the Intel ltanium 2 Processor. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing* (SC '04). IEEE Computer Society, Washington, DC, USA, 58-.

[9] T.M. Chilimbi, "Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality," ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 191–202, Snowbird, Utah, June 2001.

[10] T.M. Chilimbi and M. Hirzel, "Dynamic Hot Data Stream Prefetching for General-Purpose Programs," ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 199-209, 2002.

[11] Luiz DeRose, K. Ekanadham, Jeffrey K. Hollingsworth, and Simone Sbaraglia. 2002. SIGMA: a simulator infrastructure to guide memory analysis. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing* (Supercomputing '02).

[12] Xiaoning Ding, Kaibo Wang, and Xiaodong Zhang. 2011. ULCC: a user-level facility for optimizing shared cache performance on multicores. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming* (PPoPP '11).

[13] Stéphane Eranian. 2008. What can performance counters do for memory subsystem analysis?. In Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08) (MSPC '08).

[14] Marty Itzkowitz, Brian J. N. Wylie, Christopher Aoki, and Nicolai Kosche. 2003. Memory Profiling using Hardware Counters. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing* (SC '03). ACM, New York, NY, USA, 17-.

[15] Weidendorfer, J., Kowarschik, M., and Trinitis, C. 2004. A Tool Suite for Simulation Based Analysis of Memory Access Behavior. In Int'l Conf. on Computational Science ICCS.

[16] J. Levon et al. Oprofile, August 2011. URL: http://oprofile.sourceforge.net/.

[17] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In HPCA '08, pages 367--378, Salt Lake City, UT, 2008.

[18] Qingda Lu, Jiang Lin, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. 2009. Soft-OLP: Improving Hardware Cache Performance through Software-Controlled Object-Level Partitioning. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques* (PACT '09).

[19] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (PLDI '05).

[20] Jaydeep Marathe, Frank Mueller, Tushar Mohan, Sally A. Mckee, Bronis R. De Supinski, and Andy Yoo. 2007. METRIC: Memory tracing via dynamic binary rewriting to identify cache inefficiencies. *ACM Trans. Program. Lang. Syst.* 29, 2, Article 12 (April 2007).

[21] Jaydeep Marathe, Frank Mueller, and Bronis R. de Supinski. 2006. Analysis of cache-coherence bottlenecks with hybrid hardware/software techniques. *ACM Trans. Archit. Code Optim.* 3, 4 (December 2006), 390-423.

[22] Margaret Martonosi, Anoop Gupta, and Thomas Anderson. 1992. MemSpy: analyzing memory system bottlenecks in programs. In *Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems* (SIGMETRICS '92/PERFORMANCE '92), Blaine D. Gaither (Ed.). ACM, New York, NY, USA, 1-12.

[23] Tipp Moseley , Alex Shye , Vijay Janapa Reddi , Dirk Grunwald , Ramesh Peri, Shadow Profiling: Hiding Instrumentation Costs with Parallelism, *Proceedings of the International Symposium on Code Generation and Optimization*, p.198-208, March 11-14, 2007

[24] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (PLDI '07). ACM, New York, NY, USA, 89-100.

[25] Nicholas Nethercote and Julian Seward. 2007. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments* (VEE '07). ACM, New York, NY, USA, 65-74.

[26] Jeffrey Odom, Jeffrey K. Hollingsworth, Luiz DeRose, Kattamuri Ekanadham, and Simone Sbaraglia. 2005. Using Dynamic Tracing Sampling to Measure Long Running Programs. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing* (SC '05). IEEE Computer Society, Washington, DC, USA, 59-.

[27] Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris. 2010. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European conference on Computer systems* (EuroSys '10). ACM, New York, NY, USA, 335-348.

[28] Brian M. Rogers , Anil Krishna , Gordon B. Bell , Ken Vu , Xiaowei Jiang , Yan Solihin, Scaling the bandwidth wall: challenges in and avenues for CMP scaling, *Proceedings of the 36th annual international symposium on Computer architecture*, June 20-24, 2009, Austin, TX, USA

[29] Amitabha Roy, Steven Hand, and Tim Harris. 2011. Hybrid binary rewriting for memory access instrumentation. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (VEE '11). ACM, New York, NY, USA, 227-238.

[30] S. Rubin, R. Bodik, and T. Chilimbi, "An Efficient Profile-Analysis Framework for Data Layout Optimizations," The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Portland, Oregon, Jan. 2002.

[31] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. *In Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205. ACM New York, NY, USA, 1994.

[32] Dan Tang, Yungang Bao, Weiwu Hu, Mingyu Chen, DMA Cache: Using On-Chip Storage to Architecturally Separate I/O Data from CPU Data for Improving I/O Performance, in *the 16th IEEE International Symposium on High-Performance Computer Architecture* (HPCA-16), 2010.

[33] Jie Tao, Martin Schulz, and Wolfgang Karl. 2003. A Simulation Tool for Evaluating Shared Memory Systems. In *Proceedings of the 36th annual symposium on Simulation* (ANSS '03). IEEE Computer Society, Washington, DC, USA, 335-.

[34] Josep Torrellas , Anoop Gupta , John Hennessy, Characterizing the caching and synchronization performance of a multiprocessor operating system, *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, p.162-174, October 12-15, 1992, Boston, Massachusetts, United States

[35] O. Traub, S. Schechter, and M. Smith. Ephemeral instrumentation for lightweight program profiling, 2000.

[36] Steven Wallace and Kim Hazelwood. 2007. SuperPin: Parallelizing Dynamic Instrumentation for Real-Time Performance. In *Proceedings of the International Symposium on Code Generation and Optimization* (CGO '07).

[37] Josef Weidendorfer, Markus Kowarschik and Carsten Trinitis. A Tool Suite for Simulation Based Analysis of Memory Access Behavior. *Proceedings of the 4th International Conference on Computational Science* (ICCS 2004), Krakow, Poland, June 2004.

[38] Jonathan Weinberg and Allan Edward Snavely. 2008. Accurate memory signatures and synthetic address traces for HPC applications. In *Proceedings of the 22nd annual international conference on Supercomputing* (ICS '08). ACM, New York, NY, USA, 36-45.

[39] Jonathan Weinberg, Michael O. McCracken, Erich Strohmaier, and Allan Snavely. 2005. Quantifying Locality In The Memory Access Patterns of HPC Applications. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing* (SC '05).

[40] Qiang Wu , Artem Pyatakov , Alexey Spiridonov , Easwaran Raman , Douglas W. Clark , David I. August, Exposing Memory Access Regularities Using Object-Relative Memory Profiling, P*roceedings of the international symposium on Code generation and optimization*: feedback-directed and runtime optimization, p.315, March 20-24, 2004, Palo Alto, California

[41] Xiangyu Zhang , Rajiv Gupta, Whole Execution Traces, Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture, p.105-116, December 04-08, 2004, Portland, Oregon