2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)

Profiling Dynamic Data Access Patterns with Bounded Overhead and Accuracy

SeongJae Park*, Yunjae Lee†, Yoonhee Kim‡ and Heon Y. Yeom§

* † § Department of Computer Science and Engineering, *Seoul National University*
‡Department of Computer Science, *Sookmyung Women's University*
Seoul, South Korea

Email: *sj38.park@gmail.com, †yjlee@dcslab.snu.ac.kr, ‡yulan@sookmyung.ac.kr, §yeom@snu.ac.kr

***Abstract*—One common characteristic of modern workloads such as cloud, big data, and machine learning is memory intensiveness. In detail, such workloads tend to have a huge working set and low locality. Especially, the size of working sets is rapidly growing so that cannot be fully accommodated by a DRAM based main memory. Worse yet, the cloud computing systems, which has been pervasive since few decades ago, are continuously reducing the size of DRAM per CPU and encouraging memory overcommitment. Consequently, efficient and effective out-of-core memory management is becoming more important.**

Though a number of memory management mechanisms for such situations have proposed, manual analysis and optimization are still required for optimal performance of each workload due to the wide variety of data access patterns. However, existing tools for memory access analysis are not appropriate to be used here because those are not designed for extraction of the dynamic data access pattern of modern workloads. When those tools are used for the purpose, those incur unacceptably high overheads for unnecessarily accurate analysis results.

To mitigate this situation, we introduce a tool that is designed for the purpose. Basically, the tool employs a memory access tracking technique based on page table entry access bit, which incurs only minimal overhead. It also provides a technique for an effective tradeoff between profiling overheads and accuracy of the output by dynamically adjusting number of tracking regions. By adopting the technique, this tool can control the level of overheads and output accuracy in bounded range that user specified regardless of the size of target workloads. The overhead can be lowered even enough to be used for online target workloads while still providing useful quality of the extracted data access pattern.

The main contributions of this paper are: 1) introduce of the data access patterns profiler tool designed for modern memory-intensive workloads, and 2) empirical memory access pattern analysis of various realistic workloads.

***Index Terms*—data access pattern, memory-intensive workloads, profiler, performance, optimization**

I. Introduction

For the last decades, data-intensive workloads including cloud, big data, and machine learning workloads have widely spread around the world. Such workloads commonly tend to have a low locality and a huge working set [8]. In many cases, those huge working sets are unable to be fully accommodated in DRAM-based main memory because the capacity improvement speed of DRAM devices has been slower than that of working set size increases. The difference between tendencies of main memory per CPU ratio on virtual machines and host physical machines also gives evidence of this trend. Since a decade ago, the ratio has continuously increased on virtual machines (indicates an increase of the size of working sets) while it consistently decreased on physical machines [11] (indicates a decrease of the DRAM capacity). Furthermore, cloud providers recommend to overcommit RAM [6].

Fortunately, both the speed and capacity of storage devices have continuously improved for the last decades as the size of working sets has increased. Nowadays, high-end storage devices show DRAM-comparable speed [3] and HDD-comparable capacity [5], though those are still slower and smaller than DRAMs and HDDs, respectively. These trends on software and hardware indicate that hierarchical memory architectures configured with heterogeneous memory devices (DRAM-based main memory and high-end storage based auxiliary memory) will be widely adopted in the near future to solve the capacity problem. Nevertheless, because such memory devices are still slower than DRAMs, placement strategy of each data item on each layer memory becomes important for the performance of given workloads.

Traditionally, locality-based data access pattern speculation mechanisms have been widely used. Least-Recently-Used (LRU) is one such mechanism most widely adopted and other improved mechanisms also proposed [7], [9], [10]. However, such locality-based speculations cannot provide high accuracy for modern workloads because those workloads normally show low locality as aforementioned [8]. Especially, such estimation becomes even harder if the given workload has complex data access pattern. Thus, knowing data access patterns of given workloads is essential to make ideal placement decision because if the patterns are known, systems can place data items to be frequently used in faster memory devices while placing items to be rarely used in slower memory devices.

It can be helpful to utilize a memory access monitoring tool to extract the data access pattern for optimization of the data placement for such huge and complex workloads. There are many memory access introspection tools [2], [4], [13] that could be used for the analysis of the data access pattern. However, existing memory access introspection tools focus on general and precise monitoring of every memory access. Their precise and general monitoring outputs can be used for a wide range of purpose including correctness check, rather than only data access pattern detection. This precise and general monitoring incurs overhead which is unnecessary and even unacceptably high for data access pattern detection. Because the overhead becomes even higher when it is applied to the modern workloads having huge working set and high complexity, such tools cannot be used for lightweight data access pattern analysis.

Therefore we propose a memory access monitoring technique that is optimized for fast and accurate data access pattern tracing. It provides minimal tracing overhead by utilizing page table entry access bit manipulation based access monitoring instead of binary instrumentation, which is widely used by existing tools. One main property of this technique is its support of tradeoff between quality of the traced output and the tracing overhead. This is achieved by splitting the entire address space of given workload into small number of memory regions and sampling pages in each region instead of fully monitoring every page. Furthermore, it continuously splits and merges memory regions so that only useful regions exists while keeping the number of memory regions in a bounded range. Owing to this property, our technique can keep the tracing overhead and the quality of the output in a bounded range regardless of the size of a given working set.

For evaluation of the technique, we implemented a tool that traces the data access pattern of given workloads utilizing the technique and call it "Daptrace". We apply the tool to a few realistic workloads and provide visualizations of those tracing outputs, which clearly shows the data access patterns of those workloads.

II. DAPTRACE: DATA ACCESS PATTERN TRACER

The main procedure of our data access pattern tracer can be simplified as a loop which runs while a user wants to keep tracing. Simplified pseudo code for the loop is illustrated in Listing 1. Following sections will describe each major parts in detail.

A. Region-Based Sampling of Memory Accesses

One widely used memory access monitoring technique is based on the program instrumentation [4]. It hooks every memory access instruction in a given workload via binary reverse engineering or source code modification and records every memory access in the hook while it runs. Though this technique ensures the precise record of entire accesses so that could be used for correctness check of each and every access, it incurs excessively high overhead for time and space. For example, our evaluation for the adoption of a widely-used instrumentation based memory access monitoring tool [4] to a realistic workload [1], which has few hundred megabytes working set and about 10 minutes runtime, consumed more than 24 hours of monitoring time and more than 500 gigabytes of storage space. Because the overhead becomes even greater as the scale of a given workload grows, it cannot be applied

```
while (target_workload_running()) {
    for_each_region(r) {
        track_region(r);
        if (last_touched(r) &&
                touched(r))
                split(r);
        n = r->next
        if (last_untouched(r) &&
                untouched(r) &&
                last_untouched(n) &&
                untouched(n))
                merge(r, n);
    }
    if (now() - last_aggr
                >= aggr_interval) {
        extract_track();
        last_aggr = now();
    }
    sleep(sample_interval);
}
```

Listing 1. Main loop of our data access pattern tracing technique

to modern workloads that commonly equips huge workloads and complex execution.

Though only instrumentation based monitoring technique can provide 100% precise tracking of every memory access, such detailed tracking is not necessary for data access pattern tracing. Moreover, because we are focusing on accesses to DRAM and lower layer memory only, memory accesses that complete with CPU cache rather than touching DRAM are not needed to be tracked. In other words, instrumentation-based memory access monitoring incurs unacceptably high overhead for unnecessarily high-quality output.

Another widely used memory access monitoring technique relies on page table access bit manipulation. Every page table entry has additional bits representing the status of corresponding page frame including permissions, present, dirty, and access. When the CPU accesses a page frame in the DRAM, it automatically sets the access bit of the page table entry for the frame. Because the CPU only sets the bit but never reset, the page table access bit manipulation based memory access tracking technique periodically checks and reset the bit for target page frames. Though this technique cannot track every memory access in detail because of the page frame sized tracking granularity and miss of accesses during the check interval, it is sufficiently detailed to extract the data access pattern of the data-intensive workloads. Also, it incurs a significantly smaller overhead compared to the instrumentation-based technique.

That said, the overhead should be still carefully controlled because the monitoring overhead grows as the number of pages to be tracked increases. If the number of pages to be tracked can increase without any limit, the time required for monitoring can exceed the time for each data access pattern. Such a situation will result in an output of unacceptably low quality. Worse yet, if the monitoring occurs too frequently, monitoring task might interfere with the target workloads so that the performance of the target workload degrades. For the reason, we divide the address space of given workload into multiple regions constructed with page frames having similar access frequencies. Then, we track accesses to each region by tracking a page inside the region which is randomly selected, with the access bit manipulation technique. Before the beginning of the main loop (Listing 1), we randomly select the page for each region and unset its access bit. After that, we check the access bit and increment the number of accesses to the region in the current aggregation interval inside the `update_nr_accesses()` function. After that, we again randomly select a page in the region to be checked at next sampling and unset its access bit.

In this way, we bound the trace overhead and the quality of the traced output to an upper limit. Keeping the balance between the number of regions and the access frequency similarity of pages in each region is the key. To make the balance, we dynamically merge and split regions as described in the following section.

B. Dynamic Identification of Effective Memory Regions

We guarantee the monitoring overhead and quality of traced results to be in a bounded range by keeping the number of regions in a bounded range while also maintaining each region to be constructed with page frames with similar access frequencies. For this, we continuously split and merge regions (lines 4–12 of Listing 1) based on the continuous activeness of each region.

Before the start of tracing, users can set a corresponding attribute, maximum number of the memory regions to manage. When the tracing starts, we first construct initial memory regions based on the memory mapping of given workload. If a region has detected to be accessed for two consecutive monitoring, it splits the region into two regions because continuously accessed region may have different access patterns inside. Else, if no access to two adjacent regions is perceived for two consecutive monitoring, it determines the regions as having no frequent access and just merges the two regions into one region. If the total number of regions reaches the maximum number of regions, which users set at initial time, it stops splitting more.

C. Implementation

We implement the Daptrace on the top of the Linux v4.20 kernel as a set of two components which resides in the kernel space and the user space, respectively. The overall architecture of our work is illustrated in Figure 1 depicts.

The relationship between the kernel space component and the user space componen is similar to that of policy and mechanism.

Kernel's obligation is implementation of the mechanism. Almost every main functions are implemented in the kernel space component, which is implemented as a form of an independent kernel module. It implements core functions including setting of the process id of the tracing target workload, initialization and cleaning of memory regions, check of the access to each region (The merge and split of regions will be executed inside this function), and extraction of currently aggregated sampling results. It implements dedicated system calls for these functions and export the interface to the user space component.

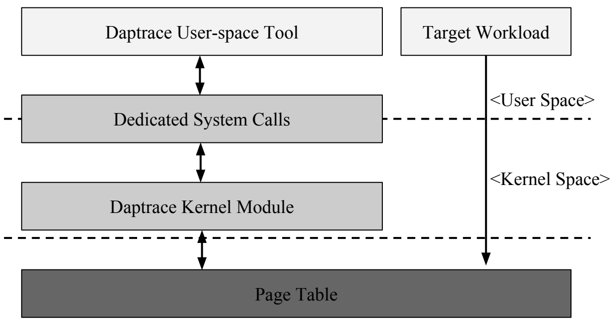

Fig. 1. Implementation of the Daptrace.

On the contrary, the userspace works as a policy maker. It also works as an intermediate layer between a user and the kernel component. It receives a target workload, transforms the user input to a form that the kernel can easily understand and sends the request to the kernel module via the dedicated system calls. After the workload finishes, it stops tracing and extracts the remaining tracing output in the kernel buffer via the dedicated system calls. It also provides a visualization feature that converts the traced data access patterns into a graph showing the data access pattern in a human-friendly way.

III. Evaluation

A. Evaluation Setup

Our evaluation of the Daptrace aims to answer to following three questions:

1) How much overhead does it incurs?
2) How precisely and human-readably does the Daptrace extracts data access patterns of realistic workloads?

To answer these questions, we use a server utilizing an Intel Xeon E7-8837 processor, 128 GB DRAM, and an Intel Optane SSD as a swap device. The server operates on the Ubuntu 18.04 LTS Server version and the Linux kernel v4.20.

Also, we set the sampling interval, the aggregation interval, and the maximum number of regions to 100 milliseconds, 1 second, 1, and 256, respectively. The values are chosen for minimum sampling overhead based on our experience.

B. Capturing Overhead

To know the overhead of the Daptrace itself, we measure the runtime of each workload when it runs alone and when it runs along with the data access pattern tracing. Nonetheless, under our default configuration which is described in Section III-A, every workload shows only neglectable differences in the runtime. This workload-independent and near-zero overhead comes from the fact that the Daptrace controls the total number of memory monitoring operations per each step in bounded range, which is adjustable with the attributes of the intervals and the bounded range. Thus, we could conclude that the Daptrace can control the overhead in a bounded range regardless of the size and characteristics of given workloads.

C. *Visualized Data Access Patterns*

Figure 2 shows the visualized data access pattern of each workload. Each region in each graph represents when and how frequently corresponding memory region has been accessed. Y-axis represents the virtual address space of each workload while X-axis represents the time from the start to the end of the workload runs. Darker shade means more frequent access and vice versa.

Roughly speaking, the virtual address space of each process in the Linux can be divided into three big regions: the uppermost (higher address) big region including stack which grows downward, the lowermost big region including heap which grows upward, and the middle big region including memory mapped regions. Because the gaps between these three big regions are usually large enough to dominate the visualized space and no access will be made into the gap, our visualization simply omits the gap area. In other words, each graph has three X-axis lines dividing the uppermost, middle, and lowermost big regions.

Based on these visualized results, we can easily divide the execution of each workload into multiple phases each containing similar data pattern. In the case of 433.milc, for example, would be able to be divided into 9 phases. In even-numbered phases, a memory region in the uppermost part becomes hot, while it is not so much in other phases. For another example, 403.gcc also could be divided into multiple phases depending on the hotness of the lower memory region and upper memory region. There are also a few workloads showing no dynamic change. For example, 470.lbm shows almost uniform access to the entire middle big memory region.

We also analyze source code of a few workloads including 433.milc, 470.lbm, and 429.mcf to confirm whether the visualized results make consistency with the source code. After the comparison of the results from the source code and the visualized results, we confirm that the visualzied data access pattern fits well with the source code. We do not describe the comparison in detail due to the page limit.

IV. Related Works

Wang *et al* [13] found that it is almost impossible to monitor memory accesses of a huge workload with conventional tools such as PIN [4], which slows a workload up to 2000x. They reduced the number of accesses needs to be profiled in a dynamic analysis by pruning predictable memory access patterns which is obtained by static analysis. The technique reduced slowdown from PIN by 61x on average. However, the tool requires a source code to profile the memory access pattern of a workload, it can not be applied directly to executable binaries. Also, it only supports programs written in C.

A miniature cache simulation technique [12] evaluated various data replacement policies on hundreds of real-world storage block traces with up to 200x smaller spaces and up to 10x faster runtime. The technique reduced space/time overhead of simulation by efficient sampling on data access stream and cache size. The authors of the work also showed that high accuracy can be achieved with sampling-based simulation. The work proposed a great idea that reduced the overhead of simulation, but the domain of the work is limited to cache simulation. Moreover, the down-scaling of a cache could be more efficient if hotness information of regions in the cache is combined.

V. Conclusion

A few kinds of workloads including cloud, big data, and machine learning, which has been prevalent during the last decades, commonly have memory intensive characteristics and huge working sets. Because the evolvement trend of hardware and software for such widely spreading workloads implicates requirements of an efficient hierarchical memory system, which is constructed with fast but small DRAM and slow but large storages, efficient and effective data placement will become a key for optimal system performance. For optimal data placement, knowing the dynamic data access pattern of a given workload is essential. Though existing memory monitoring tools could be used for data access pattern extraction, those incur unacceptably high overhead mainly because those are not developed for the purpose.

In this paper, we introduced a data access pattern tracing tool, namely Daptrace, which is developed to meet such requirements. The tool traces the dynamic data access pattern of a given workload and provides visualization of the data access pattern in human-friendly format with minimal overhead and high quality. Moreover, it can provide a knop to keep the overhead and quality of the tracing in a bounded range.

Our evaluations that conducted with eleven realistic workloads conclude that the tool imposes almost no overhead and provides a high quality which is sufficient to be run online and be adopted for performance optimization, regardless of the size and characteristics of target workloads. We demonstrated the correctness quality of the traces by manually analyzing their source code and confirmed the consistent results.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (NRF-2015M3C4A7065646 and NRF-2016M3C4A7952587).

References

[1] 433.milc, SPEC CPU2006 Benchmark Description. https://www.spec.org/cpu2006/Docs/433.milc.html.

[2] About valgrind. http://valgrind.org/info/about.html.

[3] Intels new Optane SSDs are superfast and can even work as extra RAM. https://www.theverge.com/circuitbreaker/2017/10/31/16582018/intel-optane-p900-ssd-fast-dram-nand-flash-memory-desktop-computer.

[4] Pin - a dynamic binary instrumentation tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[5] Samsung unveils worlds largest SSD with whopping 30TB of storage. https://www.theverge.com/circuitbreaker/2018/2/20/17031256/worlds-largest-ssd-drive-samsung-30-terabyte-pm1643.

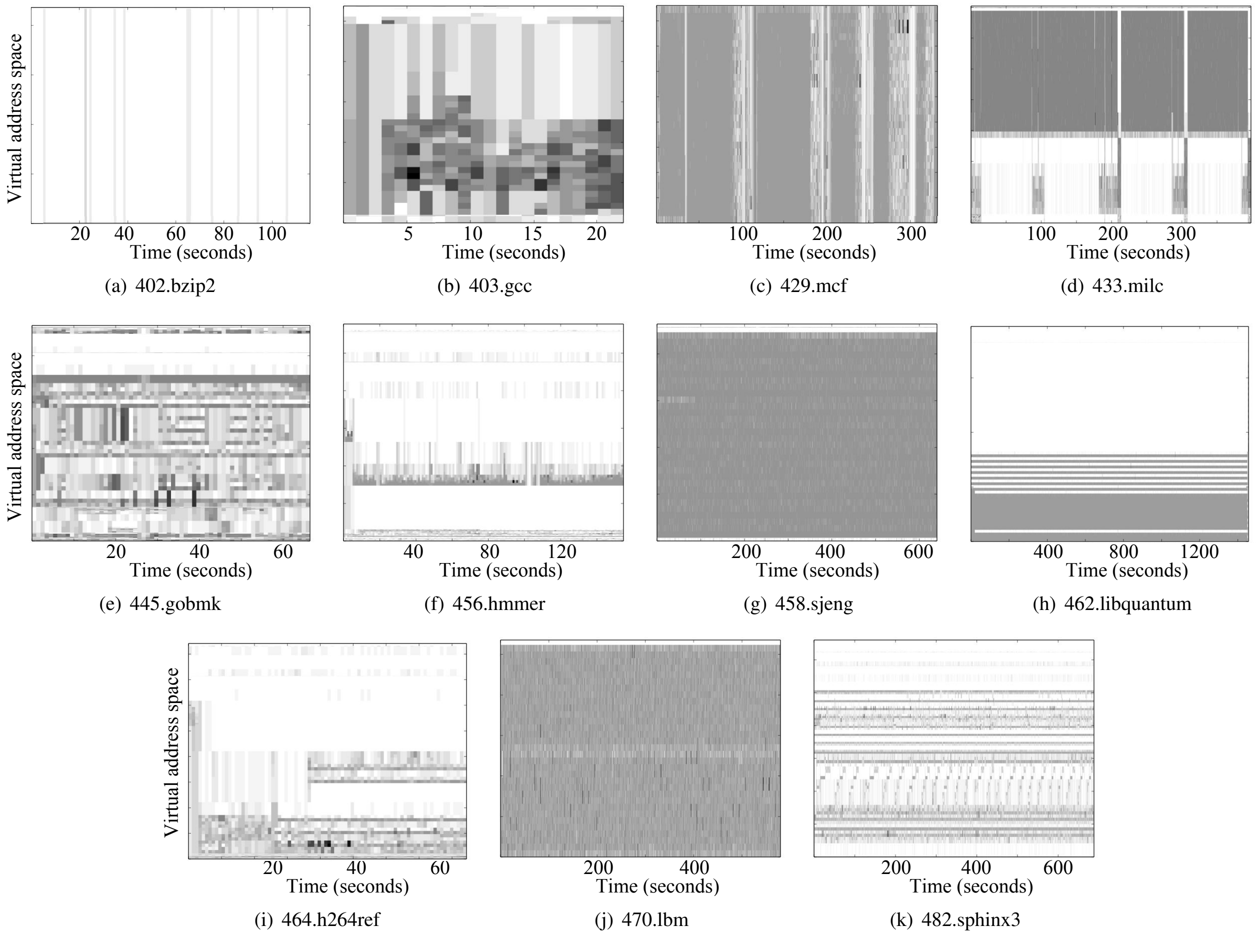

Fig. 2. Visualized data access patterns of SPEC CPU 2006 workloads.

[6] Overcommitting CPU and RAM. https://docs.openstack.org/arch-design/design-compute/design-compute-overcommit.html, 2018.

[7] Sorav Bansal and Dharmendra S Modha. Car: Clock with adaptive replacement. In *FAST*, volume 4, pages 187–200, 2004.

[8] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds. In *Proceedings of the international conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '12*, volume 47, page 37, New York, New York, USA, 2012. ACM Press.

[9] Zhan-sheng Li, Da-wei Liu, and Hui-juan Bi. Crfp: a novel adaptive replacement policy combined the lru and lfu policies. In *2008 IEEE 8th International Conference on Computer and Information Technology Workshops*, pages 72–79. IEEE, 2008.

[10] Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[11] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to zombieland: practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the Thirteenth EuroSys Conference*, page 16. ACM, 2018.

[12] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 487–498, Santa Clara, CA, 2017. USENIX Association.

[13] Haojie Wang, Jidong Zhai, Xiongchao Tang, Bowen Yu, Xiaosong Ma, and Wenguang Chen. Spindle: Informed memory access monitoring. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 561–574, Boston, MA, 2018. USENIX Association.