Profiling Dynamic Data Access Patterns with Controlled Overhead and Quality

SeongJae Park
Seoul National University
sj38.park@gmail.com

Yunjae Lee
Seoul National University
yjlee@dcslab.snu.ac.kr

Heon Y. Yeom
Seoul National University
yeom@snu.ac.kr

Abstract

Modern workloads tend to have huge working sets and low locality. Despite this trend, the capacity of DRAM has not been increased enough to accommodate such huge working sets. Therefore, memory management mechanisms optimized for such modern workloads are widely required today. For such optimizations, knowing the data access pattern of given workloads is essential. However, manually extracting such patterns from huge and complex workloads is exhaustive. Worse yet, existing memory access analysis tools incur unacceptably high overheads for unnecessarily detailed analysis results.

To mitigate this situation, we introduce a tool that is designed for data access pattern tracing. Two core mechanisms in this tool, a region-based sampling and an adaptive region adjustment, allow users to limit the tracing overhead in a bounded range regardless of the size and complexity of target workloads, while preserving the quality of results. Our empirical evaluations that conducted with 20 realistic workloads show the high quality, low overhead, and a potential use case of this tool.

CCS Concepts • **Software and its engineering → Memory management**; *System administration*; Monitors;

Keywords data access pattern, memory management, memory-intensive workloads, profiler, performance, optimization

ACM Reference format:
SeongJae Park, Yunjae Lee, and Heon Y. Yeom. 2019. Profiling Dynamic Data Access Patterns with Controlled Overhead and Quality. In *Proceedings of Middleware '19: International Middleware Conference Industrial Track, Davis, CA, USA, December 9–13, 2019 (Middleware '19),* 7 pages.
DOI: 10.1145/3366626.3368125

1 Introduction

Over the last decade, modern computing workloads that can be easily encountered in various areas including cloud, big data, and machine learning differentiate them from traditional workloads by tending to have huge working sets and low locality [14, 20, 22]. These huge working sets can even be unable to be fully accommodated in DRAM-based main memory as the growth speed of DRAM capacity in a single machine has been surpassed by that of the working set size [12, 33]. Fortunately, both the speed and capacity of storage devices have been continuously improved for decades. Nowadays, manufacturers of high-end storage devices that are equipped with modern technologies such as flash and/or phase-change memory (PCM) announce that those devices have DRAM-comparable speed [3] and hard disk drive (HDD) comparable capacity [5], though such devices are comparable but obviously slower than DRAM and smaller than HDD. This gap is not expected to be entirely disappeared in the near future, either. To mitigate this problem, systems will normally equip heterogeneous memory (utilizing DRAM-based main memory and high-end storage based auxiliary memory) and strongly require optimal memory management mechanisms [20, 32, 36].

Such optimal memory management should be able to place each data object in the proper memory device in proper time so that the performance penalty from the slow auxiliary memory device is concealed. To develop such an optimal mechanism, data access patterns of performance-sensitive workloads are required. A dedicated tool should be used for the task because manually extracting the patterns from huge and complex workloads is exhaustive. Various tools [2, 6, 30, 38] that are designed for general and precise memory access introspection can also be used for this situation. Nevertheless, those tools incur high overheads for the general and precise introspection, even though only a few parts of the introspection results are required for the data access pattern extraction. Worse yet, the overhead arbitrarily increases as the size and complexity of the target workload expand.

We introduce a data access pattern tracing tool, namely *Daptrace*, which is designed to mitigate this situation. It basically utilizes the page table access bit manipulation based access monitoring, which is also adapted to a number of other schemes, because it incurs much smaller overhead compared to workload instrumentation based schemes.

One main property of our technique that differentiates it from other page table access bit based schemes is its support of the bounded trace overhead. For this property, we split the virtual memory space of a given process into sub-regions and monitor only sampled pages in each memory region instead of every page. Users can, thus, control the upper-bound of tracing overhead by limiting the maximum number of regions. Furthermore, it minimizes the errors caused by the sampling and dynamic change of the data access pattern by adaptively restructuring the regions so that each region covers only memory regions with unique data access patterns that are different from adjacent regions. In this way, our technique keeps the worst-case tracing overhead under a user-defined bound regardless of the size of target workloads, while preserving its output quality as high as possible.

We conduct evaluations of this tool with various realistic workloads that have been categorized in a wide area, including supercomputing, scientific application, and deep neural network training. To show its preciseness of output, we visualize the data access pattern of each workload and compare the visualized pattern illustrating workflow of the workloads with manual code review

Middleware '19, Davis, CA, USA

DOI: 10.1145/3366626.3368125

analysis. We also measure its tracing overhead, which is bounded and low enough (3,159.61 times lower than a pervasive scheme) so that even could be applied online with target workloads. For further proof of its quality and potential usefulness, we apply Daptrace to optimize data-intensive workloads for a memory pressure situation on a heterogeneous memory system. The results show consistent performance improvement (1.65x on average and up to 2.55x) with negligible overhead (0.9% on average and up to 8.2%).

2 Background

The ratio of memory capacity to the number of CPU cores, which has continuously increased on virtual machines and decreased on physical machines for the last decade [33], indicates the increase in the working set size and a relative decrease in DRAM capacity. As a consequence, the optimized use of heterogeneous memory is becoming more important. Locality-based mechanisms [13, 28, 31] have been conventionally used for this area. Due to huge working sets, low locality [22] and complex data access pattern of modern data-intensive workloads, however, actual access pattern based mechanisms are required. Existing memory access monitoring techniques, which can be used for extractions of the patterns, are classified into two categories.

2.1 Workload Instrumentation

Workload instrumentation based access monitoring techniques, which are widely used for memory access analysis, install a hook in its target workload via a binary reversing [24, 30] or a compile-time optimization [15, 35]. Once the hook is installed, the workload is forced to execute the hook function before and after every memory access of the workload. The hook function then records every related data point about each access, including the type of access, access target address, the value that is loaded from or written to. Though this technique ensures that every piece of information regarding accesses is recorded so that they can be used for various purposes including bug detection, correctness check, and performance optimization, it inherently incurs excessively high overhead. To show the overhead, we measured CPU time and storage space overhead of a widely used instrumentation based memory access monitoring tool [30] with a realistic workload [1], which has about 500 megabytes of working sets and about 10 minutes of runtime. The instrumented run of the workload consumed more than 24 hours of CPU time and more than 500 gigabytes of storage space.

2.2 Page Table Access Bit Manipulation

Other widespread memory access monitoring techniques rely on page table access bit manipulation [27, 37]. Every page table entry has one additional bit representing the access status of the corresponding page frame, called access bit. This bit is set when the corresponding page frame is accessed by CPUs and can be cleared by system software. Therefore, system software can track accesses to specific page frames by repeatedly clearing and monitoring those bits with a user-defined time interval. Unlike workload instrumentation, however, this technique cannot track every memory access in detail because this scheme provides only a presence of accesses to a given page frame.

That said, this restrictive information is sufficient in many cases. Furthermore, because this technique does not directly disturb the target workload, it incurs a smaller overhead compared to the workload instrumentation based techniques. That said, it is still required to be carefully utilized, because the monitoring overhead can arbitrarily grow as the number of pages to be tracked increases. If the number of pages is too large, the time for monitoring of target pages can even exceed the user-defined time interval between monitorings so that result in unacceptably low quality of output. Moreover, if the monitoring intensively consumes CPU times or memory space, the monitoring task itself can degrade the performance of the target workloads by competing for CPUs or caches.

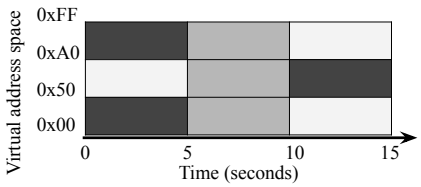

Figure 1. An example data access pattern.

3 DAPTRACE: Data Access Pattern Tracer

3.1 Definition of Data Access Pattern

The term data access pattern tends to be differently interpreted based on the individuals and involved contexts. Some works use the term to represent the static hotness of data objects while some others use it to represent sequentiality or randomness of accesses to each data object [20, 36]. In Daptrace, we define the data access pattern as access frequencies of data objects that dynamically change. Figure 1 is showing an example data access pattern of a possible workload. This example assumes the target workload has three objects that mapped on the `0x00-0x4f`, `0x50-0x9f`, and `0xa0-0xff` regions of its virtual address space. During the first 5 seconds, the workload accesses the first and the third objects frequently, but it only rarely accesses the second object. After that, it accesses three objects with the same frequency for 5 seconds. Finally, the second object is frequently accessed while the first and the third objects are rarely accessed for the final 5 seconds.

3.2 Overall Workflow

The overall workflow of Daptrace can be described as a big loop repeating access monitorings and aggregations of the monitoring results with different periods. Simplified pseudo-code for the overall workflow is illustrated by Listing 1. The loop continues as long as the user wants to (line 1). For each iteration of the loop, it first checks whether each memory region containing a data object has been accessed since the last access check (lines 2-3). If accessed, it records this access event by incrementing an internal counter representing the number of accesses to the region. Section 3.3 provides more detail about this process. After that, if the user pre-defined time interval between aggregations has been surpassed since the last aggregation (lines 4-5), it adjusts the regions (lines 6 and 8) to maintain the best quality of trace output under the dynamic access patterns. Section 3.4 illustrates this task in detail. Between the region adjustment operations, it passes the currently aggregated trace data to users, resets the internal access counters to prepare for the next aggregation (line 7), and saves current timestamp as the last aggregation time (line 9). Before starting the next iteration of this loop, a little loop spanning lines 11-13 waits for the user-defined time interval between each access monitoring passes.

```
1  while (tracing_on) {
2      for_each_region(r)
3          chk_update_nr_accesses(r);
4      if (now() - last_aggr
5                  >= aggr_interval) {
6          merge_regions();
7          flush_result_buffer();
8          split_regions();
9          last_aggr = now();
10     }
11     while (now() - last_access_track
12                 < track_interval)
13         yield_cpu();
14     last_access_track = now();
15 }
```

Listing 1. The main loop of the data access pattern tracing.

Finally, it records the current timestamp as the time that the last access monitoring completed (line 14).

3.3 Region-based Sampling of Memory Accesses

Workload instrumentation based memory access monitoring provides 100% tracking of every access. That said, such detailed information is not necessary for the data access pattern extraction that aims to be used for performance-centric memory management optimizations. Also, because this area is focusing on accesses to DRAM and lower-layer memory devices, accesses that complete within the CPU cache do not need to be tracked at all. Contrarily, the page table access bit tracking based access monitoring significantly reduces the tracing overhead and focuses only on DRAM-touching accesses. Nevertheless, if no careful control is provided, it can arbitrarily increase the overhead or unduly decrease the output quality.

Our scheme is based on the page table access bit tracking, but it tracks accesses to each data object instead of each page frame for this reason. One widely used scheme for data object identification is allocation-oriented [2, 20, 36]. In this scheme, each memory region allocated with allocation operations such as the `malloc()` library function is identified as an object. This scheme works well if the program accesses each object with a consistent data access pattern. However, there are programs dividing a big allocated region into multiple small objects or composing an object with a number of small allocated regions. These programming patterns are common and even encouraged for performance [21] and reusability [16]. To cover such corner cases, we define the data object in a data access pattern oriented way: a data object is a memory region that the program is accessing every page frame in the region with a similar access frequency. This definition not only avoids such corner cases but also reduces access monitoring overhead. By the definition, if a page frame of a data object is accessed during an interval, other page frames of the object would probably be accessed as well. Thus, we can track each object by monitoring only one page in its region instead of every page in the region.

This region-based memory access sampling is defined by the `chk_update_nr_accesses()` function we shown in Listing 1. Simplified pseudo-code for the function is in Listing 2. It first checks whether the page frame to be sampled is already decided (line 2). If not (this would be the case for regions that just created), it randomly picks a page in this region (line 7), clears the page table access bit for the page (line 8), and returns. When this function is called again

```
1 void chk_update_nr_accesses(region *r) {
2     if (!r->sample_target)
3         goto next;
4     if (accessed(r->sample_target))
5         r->nr_accesses++;
6 next:
7     r->sample_target = random_pick(r->pages);
8     set_unaccessed(r->sample_target);
9 }
```

Listing 2. The region based sampling of memory accesses.

after the user-defined tracking interval (lines 11-13 of Listing 1), it will execute line 4 because we have set the `->sample_target` before. Line 4 reads the bit again to check whether the page has been accessed after we cleared its access bit (line 8) from the last invocation of this function. If so, it updates the access number counter of the region (line 5). Then, it again randomly picks and clears access bit of the page for the next invocation of this function (lines 7-8) and returns.

The logic indicates that the tracing overhead and the accuracy of the traced output become greater as the number of regions increases and the time interval between sampling iterations decreases. Daptrace thus allows users to control the upper bound of the tracing overhead and the quality of the output by setting the upper bound of the number of regions and the time interval.

Nevertheless, this sampling technique assumes each region is representing an object as defined above. If the implementation fails to adhere to the rule, the quality of the output cannot be guaranteed. To construct the optimal regions while keeping the minimal tracing overhead, we utilize an adaptive region adjustment scheme, which is described in the following section.

3.4 Dynamic Identification of Effective Memory Regions

To ensure each memory region to accurately cover the memory area for each data object while keeping the upper bound of the number of regions, Daptrace applies an adaptive region adjustment algorithm, which is somewhat analogous to the random forest algorithm [29].

When the tracing starts, Daptrace first traverses the virtual memory area trees [17] of the target processes to see how virtual memory spaces of processes are constructed. Then, it creates initial memory regions so that the regions can cover whole data objects. If the virtual memory spaces of the processes are modified later, Daptrace updates the regions according to the modification as soon as it perceives. After constructing the initial regions, Daptrace adaptively adjusts the regions in each aggregation step (lines 6-8 of Listing 1). It first merges adjacent regions having similar access frequencies (`merge_regions()`). We judge two access frequencies as similar if the difference between those is less than 10% of their average. Once the merging task completes, it records the traced results of the current step and resets the access count of each region (`flush_result_buffer()`). Finally, if the current number of regions is smaller than a half of the upper bound, it splits each region into two regions with a random size ratio. An example sequence of this process is depicted in Figure 2.

This example assumes that the workload has one hot, small data object (depicted as a black square) and two cold, large data objects (depicted as two gray squares) adjacent to the hot one. It

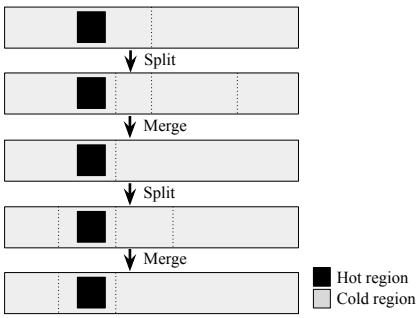

Figure 2. An example of the dynamic region identification.

also assumes the user has set the maximum number of the regions to four.

Initially, the regions are set as shown in the first row, which contains two same-sized memory regions. After an aggregation interval passes, the merge procedure checks the access frequencies of the two regions, which are probably different because the left region contains the hot data object. Thus, the two regions are not merged. Then, the trace results are recorded and the access frequencies are reset to zero. Finally, each region is split into two randomly sized regions; the second row shows this state. After one more aggregation interval passes, the merge process again checks the number of accesses to each region and merges the rightmost three regions because they have similar access frequencies. However, the leftmost region will not be merged with the others due to its high access frequency. As of this step, the state corresponds with the third row. Again, the traced results are flushed, the access frequency of each region is reset, and each region is split into two randomly sized regions, so the state becomes the fourth row. After another aggregation interval, the rightmost two regions are merged again while the leftmost two regions are not, leading to the fifth row.

In short, we repeatedly modify the regions in random manner (`split_regions()`), wait for an aggregation interval, evaluate the modification, and revert parts of modifications that are judged as unnecessary (`merge_regions()`). In this way, our scheme adaptively constructs proper regions while maintaining the minimal number of regions in the bounded range.

4 Evaluation

4.1 Evaluation Setup

Server configuration: The server we use for evaluations utilizes an Intel Xeon E7-8837 processor and a swap-based simple heterogeneous memory using 128 GB DRAM as its main memory and an Intel Optane SSD as its swap device. The Ubuntu 18.04 LTS Server and the Linux kernel v5.0 are installed on the server.

Target workloads: We selected 20 realistic workloads for various areas including super-computing, scientific application, and deep neural network training. The set of workloads includes 5 workloads in SPEC CPU 2006 (429.mcf, 433.milc, 462.libquantum, 470.lbm, and 458.sjeng), 4 workloads in NAS Parallel benchmark (bt, cg, ep, and sp), 5 workloads in PARSEC 3.0 (ferret, vips, fluidanimate, streamcluster, and freqmine), 5 workloads in SPLASH-2X (water_nsquared, fft, volrend, lu_ncb, and radix), and the Tensorflow benchmark for CIFAR-10 image [25] classification training.

Daptrace implementation and configuration: We implemented Daptrace on the Linux v5.0 kernel as a set of two components that each resides in the kernel space and the user space. The user space module interacts with users and controls the kernel space module while the kernel space module processes the main logic of Daptrace. The source code of Daptrace is available at Github (https://github.com/daptrace) under the GPL v3 open source license.

For every evaluation, we configure the sampling interval, the aggregation interval, and the maximum number of regions as 1 millisecond, 100 milliseconds, and 1,000 regions, respectively. The values set up upon our experimental experiences. We will discuss the optimized parameter setup in Section 5.

4.2 Visualized Data Access Pattern

Figure 3 shows visualized data access patterns of the 20 realistic workloads. The format of the visualization is similar to that of Figure 1. The Y-axis represents the virtual address space of each workload while the X-axis represents the time from the start to the end of the workload's execution. A darker data plot in the layer indicates higher access frequency and the bar in the right side shows a mapping between the color and the number of accesses per 100 milliseconds.

We measured the runtime of each workload when it runs alone and when it runs with the access pattern tracing to show the interference from Daptrace. However, those runtimes were almost the same owing to Daptrace's low overhead.

Based on these visualizations, we can easily divide the execution of each workload into multiple phases, each containing a different data access pattern. For example, the execution of *433.milc* can be divided into 9 phases. In even-numbered phases, a memory region in the uppermost part of the middle big region becomes especially hot, while it is not so much in other phases. For another example, *fft* is also clearly divided into 7 phases depending on the hotness of its memory regions. There are also a few workloads showing no dynamic access pattern change. For instance, *470.lbm* and *streamcluster* show static data access patterns which access a part of the middle big memory region only. The visualized data access patterns also reveal sequentiality, randomness, and skewness of accesses. *freqmine*, *water_nsquared*, and *lu_ncb* show clear sequential access patterns to a portion of its memory regions, while many other workloads tend to show skewed or uniformly random access to memory regions.

We further analyze the source code of a few workloads including *433.milc* and compare the results with the visualized patterns to prove the correctness of the Daptrace output. In the case of the *433.milc*, its `main()` function executes a big loop that repeats 4 times. This is the reason for the data access pattern that is repeated four times in the visualization. The code review also figures out that a 312 MB-sized object allocated in the `make_lattice()` function is intensively accessed during the execution. Our further investigation confirms that the big region that is the most frequently accessed in the visualized pattern matches to the object. We also investigate and compare other workloads in this way and get similar results, but omit the details due to the page limit.

With these results, we conclude that Daptrace extracts a highly accurate data access pattern as we intended with the dynamic identification of effective memory regions, while inducing almost zero interference to target workloads.

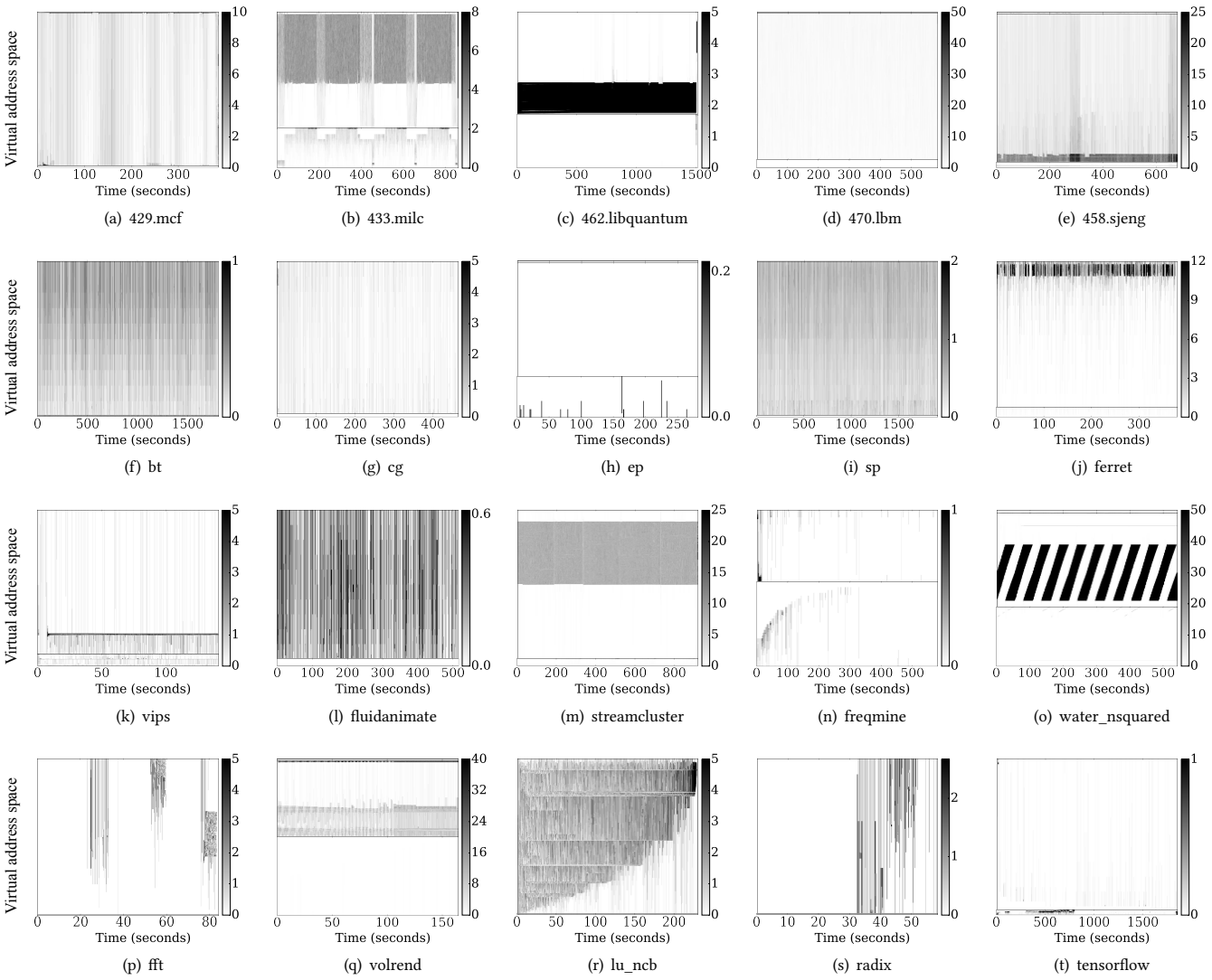

Figure 3. Visualized data access patterns.

4.3 Tracing Overhead

To show how much our region-based sampling technique reduces the tracing overhead, we measure the average number of access bits to be checked for each monitoring step, which is in proportion to the overhead. For this measurement, we use 9 data-intensive workloads from the 20 workloads and compare the numbers with that from the normal page table access bit based scheme, which is described in Section 2.2. Figure 4 shows the results in logscale.

As expected, the region-based scheme keeps the number under 1,000 as we intended. The number is not only keeping the bound but minimal (only 6 and 7 bits for 470.lbm and sp, respectively, and in total average 132.88 bits) due to its adaptive region adjustment algorithm. Contrarily, the normal scheme always shows higher overhead which increases in proportion to the size of the workload. Compared to the normal page table access bit based scheme, our region-based scheme achieves at least 24.92x (volrend), in average 3,159.61x, and up to 94,242.42x (fft) overhead reduction. Upon these results, we conclude that Daptrace controls the overhead in a bounded range regardless of the size and complexity of given workloads as we intended with the region-based sampling.

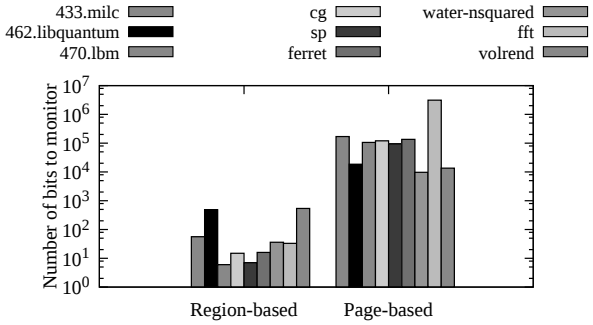

Figure 4. Tracing overhead of the schemes.

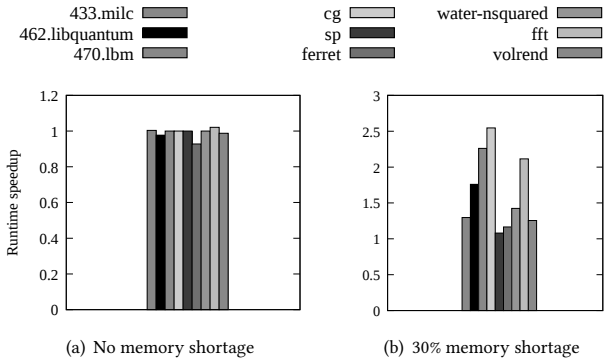

Figure 5. Runtime speedup of manually optimized workloads.

4.4 Manual Optimizations for Memory Pressure

Memory pressure is inevitable in many cases [27, 33], while the underlying system cannot provide optimal data placement because it cannot know the data access pattern of running workloads. For this reason, operating systems provide special primitives [7, 9] that allow programmers to suggest the preferred location for each data object. To show the potential usefulness of Daptrace for this kind of situations, we optimize the 9 workloads we used in Section 4.3 with the primitives using the Daptrace-provided data access pattern.

We shrink the available memory for each workload to be only 70% of its performance-effective working set size using cgroups [8]. This harsh memory pressure is possible in the real world where cloud providers are recommended to overcommit DRAM by 1.5 times [4]. For the optimization, we simply pick a few hot regions of a given workload and apply the `mlock()` system call [9] to force the regions to be locked in the DRAM. We measure the runtime speedup of the optimization for each workload with two situations: with and without the memory pressure. The measurement of the speedup without the memory pressure shows the overhead of the optimization itself. Figure 5 shows the measurement results.

These optimizations for each workload consistently incur only modest overhead and improve the performance. The overhead is 0.9% on average and about 8.2% overhead is observed in the worst case (*ferret*). Meanwhile, the performance improvement is 1.65x in average and 2.55x in the best case (*cg*). Note that our optimization is intentionally simple; the performance would be much more improved by spending more efforts to the optimization, but it is not within the scope of this paper. In short, the results show the high potential of Daptrace to greatly aid the optimization of memory management under memory pressure.

5 Future works

Though the default Daptrace parameters we used in this paper, which is set upon our experimental experiences, worked well for the 20 randomly selected realistic workloads, someone would lack such experiences. We will extend Daptrace with a machine learning based inference model that automatically finds best parameters for given workloads. Because machine learning algorithms are well-known for their success at finding answers to complex problems, utilizing those for best parameter searching would make sense.

We have shown a potential benefit of Daptrace for the optimization of workloads for memory pressure situation. Because Daptrace is lightweight enough to be even run online with target workloads, we believe that it could be used in almost every system area that traditionally relies on data access pattern estimations. Selection of candidates for transparent huge pages [10, 26, 34], compaction [18], or reclamation [23] could serve as good examples. We will expand and mature the use of Daptrace with these use cases.

6 Related Works

Wang *et al.* [38] reduced the overhead in workload instrumentation based monitoring by pruning predictable memory access patterns which were obtained by static analysis. Nonetheless, it is not designed for the data access pattern but for the conventional general access monitoring. Also, it requires source code recompilation and supports only C, C++, and Fortran.

A miniature cache simulation technique [37] reduced space/-time overhead of simulations by utilizing an efficient sampling and showed that sampling-based simulation can achieve high accuracy. That said, the domain of the work is limited to in-memory cache simulation and lacks elaborated sampling region identification.

Various memory management schemes have been proposed [20, 32, 36] for the heterogeneous memory system. Such schemes profile the data access pattern of data objects and calculate their priorities. Because most of these approaches use the workload instrumentation based profiling which incurs large overhead, we believe Daptrace could be alternatively used.

Lagar-Cavilla *et al.* [19, 27] introduced a proactive page reclaim scheme. They utilize the page table access bit based monitoring to identify cold pages that have not been accessed for at least 2 minutes. Simultaneously, their machine learning based algorithm decides which cold pages are better to be swapped out to `zram` [11] block devices and reclaims those to minimize the memory pressure. Nevertheless, their access monitoring of every page frame incurs high overhead so that one dedicated CPU needs to be fully utilized.

7 Conclusion

Memory-intensive modern workloads have been widespread over a couple of decades. The evolvement trend of hardware and software for heterogeneous memory system implicates a strong request for efficient and effective memory management. For such mechanisms, knowing the dynamic data access patterns of given workloads is essential. Though existing memory monitoring tools could be used for the purpose, those incur unacceptably high overhead for unnecessarily detailed results.

We introduced a data access pattern tracing tool, called Daptrace, which traces the data access patterns of given workloads in high quality with minimal and controllable overhead. Our evaluations conducted with 20 realistic workloads conclude that Daptrace provides high-quality trace outputs with minimal overhead, regardless of the size and complexity of the target workloads. Additionally, we demonstrated a potential use of Daptrace for manual optimization of workloads for a memory pressure situation, which achieved consistent speedups with only modest overhead.

Acknowledgments

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

References

[1] 2011. 433.milc, SPEC CPU2006 Benchmark Description. (2011). https://www.spec.org/cpu2006/Docs/433.milc.html.

[2] 2015. Extrae user guide. (2015). https://tools.bsc.es/sites/default/files/documentation/extrae-3.2.1-user-guide.pdf.

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

[4] 2018. Overcommitting CPU and RAM. (2018). https://docs.openstack.org/arch-design/design-compute/design-compute-overcommit.html.

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

[6] 2019. About Valgrind. (2019). http://valgrind.org/info/about.html.

[7] 2019. madvise(2) - Linux manual page. (2019). http://man7.org/linux/man-pages/man2/madvise.2.html.

[8] 2019. Memory Resource Controller. (2019). https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt.

[9] 2019. mlock(2) - Linux manual page. (2019). http://man7.org/linux/man-pages/man2/mlock.2.html.

[10] 2019. Transparent Hugepage Support. (2019). https://www.kernel.org/doc/Documentation/vm/transhuge.txt.

[11] 2019. zram: Compressed RAM based block devices. (2019). https://www.kernel.org/doc/Documentation/blockdev/zram.txt.

[12] Neha Agarwal and Thomas F Wenisch. 2017. Thermostat: Application-transparent page management for two-tiered main memory. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 631–644.

[13] Sorav Bansal and Dharmendra S Modha. 2004. CAR: Clock with Adaptive Replacement. In *3rd USENIX Conference on File and Storage Technologies (FAST)*, Vol. 4. 187–200.

[14] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D Hill, and Michael M Swift. 2013. Efficient virtual memory for big memory servers. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 237–248.

[15] Pohua P Chang, Scott A Mahlke, and Wen-Mei W Hwu. 1991. Using profile information to assist classic code optimizations. *Software: Practice and Experience* 21, 12 (1991), 1301–1321.

[16] Siobhán Clarke and Robert J Walker. 2001. Composition patterns: An approach to designing reusable aspects. In *Proceedings of the 23rd international conference on Software engineering*. IEEE Computer Society, 5–14.

[17] Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. 2012. Scalable address spaces using RCU balanced trees. *ACM SIGPLAN Notices* 47, 4 (2012), 199–210.

[18] Jonathan Corbet. 2017. Proactive compaction. (2017). https://lwn.net/Articles/717656/.

[19] Jonathan Corbet. 2019. Proactively reclaiming idle memory. (2019). https://lwn.net/Articles/787611/.

[20] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*. ACM, 15.

[21] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the bsdcan conference, ottawa, canada*.

[22] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Vol. 47. ACM Press, New York, New York, USA, 37. DOI: https://doi.org/10.1145/2150976.2150982

[23] Mel Gorman. 2004. *Page Frame Reclamation*. Prentice Hall Upper Saddle River. https://www.kernel.org/doc/gorman/html/understand/understand013.html.

[24] Aamer Jaleel. 2007. Memory characterization of workloads using instrumentation-driven simulation. (2007). http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf.

[25] Alex Krizhevsky and Geoffrey Hinton. 2009. *Learning multiple layers of features from tiny images*. Technical Report. Citeseer.

[26] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. 2016. Coordinated and efficient huge page management with ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 705–721.

[27] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, NY, USA, 317–330. DOI: https://doi.org/10.1145/3297858.3304053

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

[29] Andy Liaw, Matthew Wiener, and others. 2002. Classification and regression by randomForest. *R news* 2, 3 (2002), 18–22.

[30] 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 *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[31] Nimrod Megiddo and Dharmendra S Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *2nd USENIX Conference on File and Storage Technologies (FAST)*, Vol. 3. 115–130.

[32] Gaku Nakagawa and Shuichi Oikawa. 2017. Data Placement Based on Data Semantics for NVDIMM/DRAM Hybrid Memory Architecture. *CLOUD COMPUTING 2017* (2017), 109.

[33] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. 2018. Welcome to zombieland: practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*. ACM, 16.

[34] SeongJae Park, Minchan Kim, and Heon Y Yeom. 2018. GCMA: Guaranteed Contiguous Memory Allocator. *IEEE Trans. Comput.* 68, 3 (2018), 390–401.

[35] Karl Pettis and Robert C Hansen. 1990. Profile guided code positioning. In *ACM SIGPLAN Notices*, Vol. 25. ACM, 16–27.

[36] Harald Servat, Antonio J Peña, Germán Llort, Estanislao Mercadal, Hans-Christian Hoppe, and Jesús Labarta. 2017. Automating the application data placement in hybrid memory systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 126–136.

[37] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. 2017. Cache Modeling and Optimization using Miniature Simulations. In *2017 USENIX Annual Technical Conference (ATC)*. USENIX Association, Santa Clara, CA, 487–498. https://www.usenix.org/conference/atc17/technical-sessions/presentation/waldspurger

[38] Haojie Wang, Jidong Zhai, Xiongchao Tang, Bowen Yu, Xiaosong Ma, and Wenguang Chen. 2018. Spindle: Informed Memory Access Monitoring. In *2018 USENIX Annual Technical Conference (ATC)*. USENIX Association, Boston, MA, 561–574. https://www.usenix.org/conference/atc18/presentation/wang-haojie