Lightweight memory tracing for hot data identification

Yunjae Lee, Yoonhee Kim & Heon Y. Yeom

Cluster Computing
The Journal of Networks, Software Tools and Applications

ISSN 1386-7857

Cluster Comput
DOI 10.1007/s10586-020-03130-1

Springer

Your article is protected by copyright and all rights are held exclusively by Springer Science+Business Media, LLC, part of Springer Nature. This e-offprint is for personal use only and shall not be self-archived in electronic repositories. If you wish to self-archive your article, please use the accepted manuscript version for posting on your own website. You may further deposit the accepted manuscript version in any repository, provided it is only made publicly available 12 months after official publication or later and provided acknowledgement is given to the original source of publication and a link is inserted to the published article on Springer's website. The link must be accompanied by the following text: "The final publication is available at link.springer.com”.

Springer

Author's personal copy

Cluster Computing
https://doi.org/10.1007/s10586-020-03130-1

Lightweight memory tracing for hot data identification

Yunjae Lee[1] · Yoonhee Kim[2] · Heon Y. Yeom[1]

Received: 2 December 2019 / Revised: 16 March 2020 / Accepted: 13 May 2020
© Springer Science+Business Media, LLC, part of Springer Nature 2020

Abstract

The low capacity of main memory has become a critical issue in the performance of systems. Several memory schemes, utilizing multiple classes of memory devices, are used to mitigate the problem; hiding the small capacity by placing data in proper memory devices based on the hotness of the data. Memory tracers can provide such hotness information, but existing tracing tools incur extremely high overhead and the overhead increases as the problem size of a workload grows. In this paper, we propose Daptrace built for tracing memory access with bounded and light overhead. The two main techniques, region-based sampling and adaptive region construction, are utilized to maintain a low overhead regardless of the program size. For evaluation, we trace a wide range of 20 workloads and compared with baseline. The results show that Daptrace has a very small amount of runtime overhead and storage space overhead (1.95% and 5.38 MB on average) while maintaining the tracing quality regardless of the working set size of a workload. Also, a case study on out-of-core memory management exhibits a high potential of Daptrace for optimal data management. From the evaluation results, we can conclude that Daptrace shows great performance on identifying hot memory objects.

Keywords Memory tracing · Hot data identification · Performance · Optimization · Memory management

1 Introduction

Modern computing workloads are characterized by huge working sets and low localities [1–3]. However, the growth rate of memory devices such as DRAM has not followed this trend, and it is evident in the number of data centers and servers that suffer from a lack of memory these days. Moreover, as the evolution of CPUs leads to an increasing number of cores in a system, the size of memory that a single CPU core can utilize decreases [4]. Thus, main memory is becoming the bottleneck of system performance.

To overcome the low memory capacity, alternatives such as heterogeneous memory are used. They utilize two or more types of memory: small high-performance memory and large low-performance memory. Also, modern high-end storage devices such as a solid-state drive and phase change memory [5] reveal the opportunity for out-of-core management. These alternatives give an illusion of large high-performance memory by keeping the hot data in fast memory. However, it is hard to provide the illusion without proper data management, and the identification of the hot objects in a workload plays an important role in this case.

One may trace the memory accesses of a workload for hot object identification. Unfortunately, however, it would take a large amount of time, because memory tracing commonly incurs excessively high overheads. For example, a popular memory tracing tool [6] results in a 1000 × slowdown of workloads [7]. Moreover, an extremely large amount of trace data is generated by this technique. The main reason for these high overheads is that these kinds of tools exploit workload instrumentation technique, which provides rich information for various fields such as performance modeling [8] and bug detection [9]. However, the rich information is too excessive for checking the hotness of data objects; we do not have to pay a large amount of tracing cost to gather such information.

✉ Yoonhee Kim
yulan@sookmyung.ac.kr

Yunjae Lee
lyj7694@gmail.com

Heon Y. Yeom
yeom@snu.ac.kr

[1] Seoul National University, Seoul, South Korea

[2] Sookmyung Women's University, Seoul, South Korea

Published online: 04 June 2020

Author's personal copy

Fortunately, a large portion of the high overheads can be alleviated by access bit tracking technique. The technique does not directly interfere with the execution of a program, unlike the workload instrumentation technique. Nevertheless, because the technique fundamentally tracks accesses by checking whether each page of a program is accessed, the tracing overhead increases as program size grows. The problem becomes worse when tracing modern workloads that have extremely large working sets [1]. Moreover, the tracing quality may degrade as the number of pages to be tracked increases.

Thus, this paper presents Daptrace; a lightweight memory access tracer for the identification of hot objects that has low overhead regardless of program size. To keep the overhead in a range, it exploits the access bit tracking technique with a spatial sampling technique. The sampling technique makes it possible to draw the overall memory access of a program by inspecting a small number of accesses. Also, Daptrace can even be applied to workloads already running on a system because it does not require any workload modification.

We trace a wide range of workloads with Daptrace: high-performance computing, machine learning, and scientific computing workloads. Results show that Daptrace incurs only 1.95% runtime overhead for tracing while keeping the tracing quality to provide the hotness information. Only a small amount of storage space (5.38 MB on average) is occupied for recording the trace data. We also trace the workloads without the sampling technique, and we observe that the sampling technique reduces the runtime overhead by 14.2 $\times$. In addition, we conduct a case study on out-of-core memory management using Daptrace. We optimize 9 workloads according to the data access pattern of workloads, and performance improvement of 1.58 $\times$ is achieved under high memory pressure. The experimental result exhibits the high potential of Daptrace, that Daptrace can be effective for such optimized data management.

The rest of this paper is organized as follows: Sect. 2 provides a background of memory access tracing; Sect. 3 introduces the Daptrace with its design in detail; Sect. 4 shows the evaluation of Daptrace with a wide range of workloads and studies its example use case; Sect. 5 discusses past works related to this paper and Sect. 6 concludes this paper.

2 Background

Memory access tracing tools provide a variety of information on memory accesses, including the hotness of memory areas. They can be classified into two categories based on the core techniques: workload instrumentation and access bit tracking.

2.1 Workload instrumentation

Workload instrumentation is a widespread memory access tracing technique [6, 10]. It uses compile-time optimization [11, 12] or binary reversing scheme [13] to install a hook in a target workload. When the workload runs, it executes the hook function before and after every execution of memory reference instructions (e.g., `load` and `store`), and the hook function records the information of the memory reference. The information may include the accessed memory address, type of access, and the value loaded or written. This technique provides rich information on every memory access, so it can be applied in various fields such as correctness check, bug detection and performance optimization. Since the technique hooks every memory access, however, it inherently takes a large amount of time for tracing. Worse yet, to record the information of every memory access, extremely large storage space is needed.

We trace a realistic workload [14] with a popular instrumentation-based memory access monitoring tool [6] to show the overhead. The workload has a working set of a few hundred megabytes and takes 10 min to complete. The execution time exceeds 24 h with tracing and the size of the trace result is larger than 500 GB. It is not even able to finish the tracing; the trace result requires more storage space than available in our experiment.

2.2 Access bit tracking

Access bit tracking is a technique that is widely used for memory access monitoring [15, 16]. It exploits the *Accessed bit* in a page table entry, which shows whether the mapped page has been accessed or not. The bit is set when the corresponding mapping is used for address translation during a translation lookaside buffer (TLB) walk or a page table walk. The bit can be cleared by the software, therefore, it is possible to track memory accesses by periodically clearing and checking the bit. Unlike the workload instrumentation technique, this technique does not gather information such as access type, the value loaded or written, or fine-grained memory address accessed. This technique tracks only whether a page is accessed or not. To trace how frequently a page is accessed, it requires a brief period of monitoring.

Since this technique does not directly interfere with the execution of a program, it incurs a smaller overhead compared to the instrumentation technique. Nevertheless, as the program utilizes more memory, the overhead becomes higher and the tracing quality degrades sharply

Author's personal copy

because it requires a longer time to monitor whole pages which leads to infrequent access monitoring. Thus, it is challenging to maintain tracing quality with various kinds of workloads.

3 Daptrace: a lightweight memory tracer

3.1 An example of a memory trace

Figure 1 shows an example of a memory access trace of a workload. The workload has three objects mapped on the virtual address space: `0x00-0x4f`, `0x50-0x9f` and `0xa0-0xff`. The first and third objects are frequently accessed during the first 5 s. In other words, those two regions are hot for 5 s at the start. Immediately after, the three regions are accessed equally for 5 s. The workload then runs for another 5 s before it finishes, and the second object is frequently accessed during this period. From now on, we use the term "access frequency" as "observed access frequency".

3.2 Overall workflow

Fundamentally, Daptrace exploits access bit tracking technique due to its low overhead. In addition, to deal with workloads with various sizes, Daptrace uses a spatial sampling method to bound the overhead of tracing, which we call *region-based sampling*. The region-based sampling is described in the Sect. 3.3 in detail.

The overall workflow of the Daptrace can be described as a loop that repeatedly monitors accesses and aggregates monitoring results with different periods. A simplified pseudo-code for the workflow is illustrated by Listing 1. The loop iterates until a target workload finishes execution or a user explicitly stops the tracing (line 1). In each iteration of the loop, it is first checked whether each memory region has been accessed since the last check (lines 3–4). If a region is accessed, an access counter corresponding to the region is incremented. Subsequently, Daptrace checks if the time spent has passed *aggregation interval* (lines 7–8). If it has, Daptrace aggregates the monitoring result, resets access counters, and adjusts memory regions for the next aggregation (lines 9–11). The adjustment is called *adaptive region construction*, and it is illustrated in Sect. 3.4 in detail. Before continuing the next iteration of the loop, Daptrace waits for a short period to ensure *monitoring interval* is passed until the next monitoring (lines 16–18).

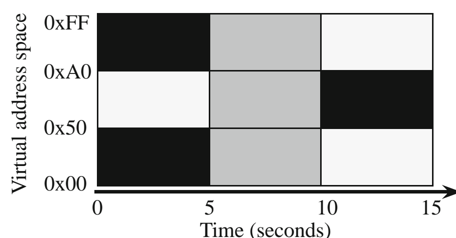


Fig. 1 An example data access pattern

```
1  while (!need_stop()) {
2       /* access monitoring */
3       for_each_region(r, rlist)
4               check_access(r);
5
6       /* region adjustment */
7       if (now() - last_aggr
8                       >= aggr_interval) {
9               merge_regions(rlist);
10              aggregate_results(rlist);
11              split_regions(rlist);
12              last_aggr = now();
13      }
14
15      /* wait until the next monitoring */
16      while (now() - last_monitor
17                      < monitor_interval)
18              yield_cpu();
19      last_monitor = now();
20 }
```

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

3.3 Region-based memory access sampling

Access bit tracking technique incurs significantly less overhead for memory access tracing, compared to the workload instrumentation technique. Daptrace uses this technique, but it has an inherent problem. The overhead and tracing quality can become worse as the working set of a workload grows.

Daptrace solves the problem with a spatial sampling method. Instead of monitoring all the pages, it monitors only a small portion of pages, each representing a continuous memory region. If Daptrace observes that a page is accessed, it considers that the whole region represented by the page is accessed. This indicates that a memory region should be constructed with pages having similar access frequency.

We might construct the regions by memory allocation information, which is a widely used scheme for the identification of data objects [3, 10, 17]. 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 an object is accessed uniformly. However, programs do not always access allocated objects in a uniform manner, and there are programs, even, that divide an allocated region into small objects or compose an object with small allocated regions. The latter type of programs is common and even encouraged for better performance [18] and reusability [19]. Thus, it is not a good idea to construct the memory regions in an allocation-

Author's personal copy

oriented manner. Instead, we use an adaptive method to construct the memory regions; Daptrace tries to find out the best arrangement of memory regions at runtime. The trace information is used for the adaptive construction. Section 3.4 explains how Daptrace constructs the regions adaptively.

The region-based memory access sampling is defined by the `check_access()` function called in the main loop in Listing 1. A simplified pseudo-code for the function is in Listing 2. It first checks whether a region has a sampled page or not (line 2). If so, it checks the access bit of the sampled page and updates the access counter of the region (lines 4–5). Otherwise, which is the case for regions just created, it skips the access check. Next, it randomly samples a page in the region (line 7) and clears the access bit for the page (line 8).

```
1 void check_access(region *r) {
2     if (!r->sampled_page)
3         goto next;
4     if (accessed(r->sampled_page))
5         r->nr_accesses++;
6 next:
7     r->sampled_page = rand_pick(r->pages);
8     clear_accessed(r->sampled_page);
9 }
```

Listing 2 The region based sampling of memory accesses.

This sampling technique assumes each region is constructed properly as mentioned above. If the memory regions are poorly constructed, Daptrace fails to generate accurate trace results. To construct the memory regions properly, we utilize an adaptive region construction scheme, which is described in the following section.

3.4 Adaptive construction of memory regions

Daptrace requires two conditions to meet with construction of memory regions. First, pages in a region should have similar access frequencies; which makes sampling-based access monitoring possible. Second, the number of memory regions must not exceed *max_nr_regions*, which helps limit tracing overheads. To satisfy these conditions, Daptrace exploits an adaptive region construction algorithm, which is similar to the random forest algorithm [20].

At the initialization step, Daptrace splits a virtual address space into *min_nr_regions* regions. However, it is not a good idea to use the entire address space, because a virtual address space is usually fragmented and only its portion is utilized. Figure 2 illustrates a general virtual address space. The *code* and *heap* sections are placed in the low address, while the *stack* section lies in the high address. The *mmaped* section, which includes file-backed

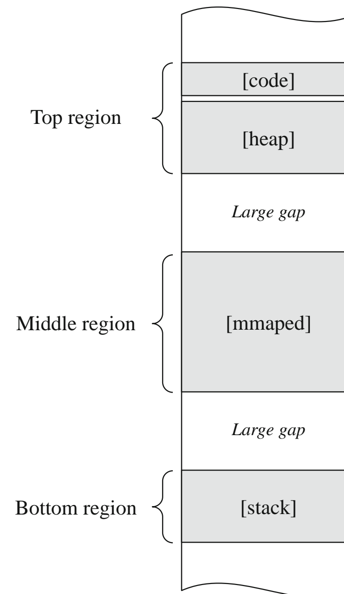


Fig. 2 A general memory address space

pages and anonymous pages, is located in the middle address. One can observe that there are two large gaps adjacent to the mmaped section. Thus, we construct three regions; one with the code and the heap sections, another with the mmaped section, and the other with the stack section. Then, the Daptrace splits the middle region to make sure that *min_nr_regions* regions are constructed before starting the main loop in Fig. 1.

After the initialization, the Daptrace starts tracing memory accesses. It counts accesses to each region repeatedly during each aggregation interval. When the time exceeds the interval, it first compares each region with adjacent regions and merges them if they have similar access frequencies in `merge_regions()`. The weighted average of access counts of the two regions becomes the access count of the merged region. After that, Daptrace records the counts and resets the counters in `aggregate_results()`. Then, each region is split into two regions with a random ratio in `split_regions()`. When these steps are done, each region has access count of zero which means it is ready for the next aggregation interval.

The random ratio split makes it possible to construct regions properly with sampling-based access monitoring, even the accesses are focused on small memory areas. Such adjustment is performed during each aggregation. During the adjustment, the number of regions is kept in between *min_nr_regions* and *max_nr_regions*. The *max_nr_regions* makes Daptrace have an upper-bounded overhead, and the *min_nr_regions* helps Daptrace follow the change of data access pattern of a program quickly.

Author's personal copy

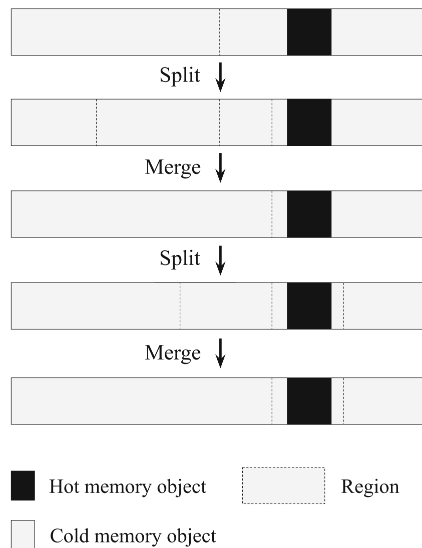


Fig. 3 An example of the adaptive region construction

Figure 3 shows an example of this process. Two gaps are not illustrated for simplicity. In the example, there is a small object accessed frequently (depicted as a black box) and two objects accessed less frequently (depicted as two gray boxes). In the first row, there are two regions that are not yet well constructed. After splitting both regions, four regions now exist. Another aggregation interval has passed, the regions are checked to be merged. Daptrace observes similar access frequencies for the three regions on the left. However, it observes a higher access frequency for the rightmost region due to the small hot object. Thus, only the three regions in the left are merged. After a cycle of splitting and merging, the regions are constructed properly as shown in the fifth row.

As the proper regions are constructed as time passes, the virtual address space also changes. The major cause of the change is the dynamic allocation or deallocation of memory objects. Whether the regions are constructed properly or not, they depend on the initial regions constructed at the initialization step, which might be outdated. Thus, Daptrace updates the regions with *update interval*.

To show how Daptrace actually adjusts regions, we conducted an experiment with an example program by tracing the program using Daptrace. The example program allocates eight memory objects with the same size and accesses the objects one by one in ascending order. Thus, Fig. 4a would be the actual data access pattern of the program. Figure 4b is the trace result that Daptrace generates by tracing the program. From the figure, we can see that accesses are observed not accurately at the start of access to each object. This is due to the change of access phase which leads to the reconstruction of regions. As shown in Fig. 3, a few merge/split steps are needed to find the proper region boundaries. However, we observe that Daptrace quickly adapts to the phase change so only a small amount of error occurs from the reconstruction. Once after regions are properly constructed, they keep the boundaries that separate hot memory objects from cold memory objects. This is because the hot regions are not likely to be merged with adjacent cold regions due to the difference of access frequencies observed.

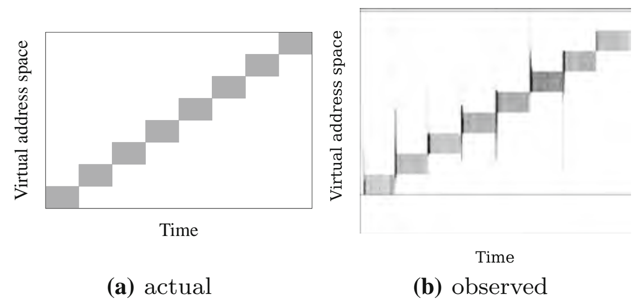


Fig. 4 An actual data access pattern of an example program (**a**) and the trace of the program (**b**). The example program has eight memory objects and accesses the objects in ascending order

In short, Daptrace constructs proper regions with an adaptive algorithm, which splits and selectively merges regions repeatedly. The number of regions is maintained in a range to limit the tracing overhead. However, because the result of a split is random, it might take a long time to construct proper regions. This is discussed with evaluations in the next section.

4 Evaluation

In this section, we evaluate the accuracy and overhead of Daptrace. To measure the accuracy, we compare the traces from Daptrace with baseline traces, which are extracted by general access bit tracking technique without sampling method, described in Sect. 2.2. In the case of overhead, we measure the size of trace results and the tracing time of Daptrace and the baseline. In addition, we conduct a case study on out-of-core memory management to show the effectiveness of Daptrace in such areas that require identification of data hotness.

4.1 Evaluation setup

System configuration The machine we used for evaluations has an Intel Xeon E7-8837 processor and 128 GB of DRAM as the main memory. It utilizes an Intel Optane SSD as a fast swap device, and has the Ubuntu 18.04 LTS Server, with the Linux kernel v5.0 installed.

Workloads We chose 20 realistic workloads in wide areas including high-performance supercomputing,

Author's personal copy

machine learning, and scientific computing. The workloads include 429.mcf, 433.milc, 458.sjeng, 462.libquantum and 470.lbm in SPEC CPU 2006 benchmark (SPEC); bt, cg, ep and sp in NAS parallel benchmarks (NAS); ferret, vips, × 264, streamcluster and freqmine in PARSEC benchmark suite (PARSEC); water_nsquared, fft, volrend, lu_ncb and raytrace in SPLASH-2x benchmarks (SPLASH); finally, a Tensorflow benchmark for training the CIFAR-10 dataset classification [21, 22].

We used input types of test, train and ref in SPEC CPU 2006 benchmark; A, B and C in NAS parallel benchmarks; simmedium, simlarge and native in both PARSEC benchmark suite and SPLASH-2x benchmarks. We classified the input types into three classes by the size: class A, B and C. Class A consists of the smallest input types, class B consists of medium-size input types, and class C consists of the largest input types in the benchmarks. Table 1 shows the classification of the input types. The class C input types are used in Sect. 4.2, and the class A, B and C input types are used in Sect. 4.3.

Implementation of Daptrace We implemented Daptrace as a loadable kernel module on the Linux kernel v5.0. The module creates a tracer thread that runs in parallel with the target workload. Some userspace tools were developed for visualization and investigation of Daptrace results.

Daptrace has three parameters that control the interval of the access monitoring, aggregation of monitoring results and update of regions: *monitoring interval*, *aggregation interval* and *update interval*. Also, it has two parameters that bound the number of regions: *min_nr_regions* and *max_nr_regions*. To utilize meaningful access information, the *monitoring interval* should be much smaller than the *aggregation interval*. Additionally, the *monitoring interval* should be larger than the time required for monitoring the *max_nr_regions* regions. If the *aggregation interval* is too large, it loses the ability to follow the dynamic access pattern of workloads quickly. The *update interval* can be relatively large because most memory allocations occur at the start of a workload and the virtual address space changes less frequently during runtime. The *max_nr_regions* should be small to avoid high overhead under any circumstances. The *min_nr_regions*, however, has a relatively minor effect on the overhead, it should have a proper value to help Daptrace follow the change of a program behavior quickly. The values were chosen empirically with experiments considering these conditions, which are listed in Table 2.

Table 1 Classification of input types by size

Benchmark	Class A	Class B	Class C
SPEC	Test	Train	Ref
NAS	A	B	C
PARSEC	Simmedium	Simlarge	Native
SPLASH	Simmedium	Simlarge	Native

Table 2 Values of parameters

Parameter	Value
Monitoring interval	1 ms
Aggregation interval	100 ms
Update interval	1000 ms
Min_nr_regions	10
Max_nr_regions	1000

4.2 Accuracy of daptrace

To evaluate the accuracy of Daptrace, we applied it to the selected 20 workloads and compared the results to those of baseline. We extracted the baseline traces of the workloads using a general access bit tracking technique. It shares the same internal structures with Daptrace; the only difference is that it does not utilize a sampling technique, so it monitors every page periodically. To reduce the size of the trace result, the baseline tracer divides virtual memory space into 1000 regions uniformly, except two large gaps depicted in Fig. 2, and aggregates the per-page statistics into per-region statistics. The parameters in Table 2 are also applied to the baseline tracer.

Figures 5 and 6 show the results of Daptrace and the baseline. Two traces are shown per workloads: one from Daptrace on the left and baseline trace on the right. In each trace, the x-axis represents time and the y-axis represents the virtual memory space of a workload. Memory areas are allocated discontinuously in the virtual memory space as depicted in Fig. 2, thus the two large gaps in the space were omitted and discontinuous areas are separated by two horizontal lines. The bar on the right side shows the mapping between access frequency (number of accesses per second) and darkness. The darker area indicates a higher access frequency to the area in each plot. Note that the access frequency is not very accurate in the baseline; the baseline shows the access differences between objects, but the access frequency becomes inaccurate as the memory size of a program grows, unlike Daptrace. For instance, the virtual memory size of fft (Fig. 6f) is approximately 12 GB, and the access frequency observed is even lower than that of Daptrace because a single monitoring time exceeds the monitoring interval (1 ms), which results in infrequent monitoring. However, it is enough to see which objects are hot and cold, thus we discuss the accuracy of Daptrace based on the baseline.

Author's personal copy

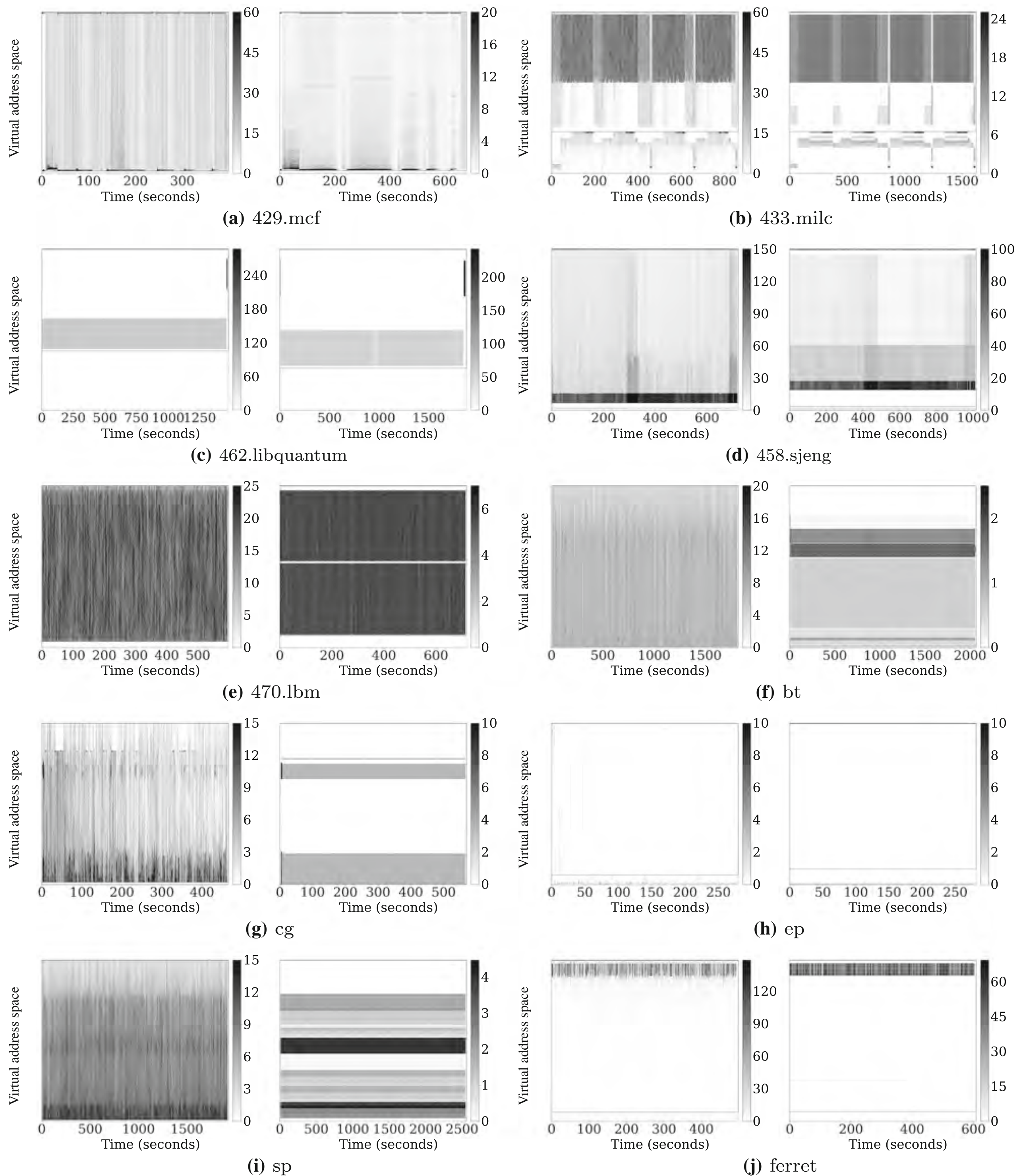


Fig. 5 Visualized trace of workloads

From Figs. 5 and 6, we can see that Daptrace is capable of identifying consistently hot memory areas. It may appear that hot areas are not properly identified in three cases: bt (Fig. 5f), cg (Fig. 5g) and sp (Fig. 5i). In these cases, some areas are not distinguished clearly. However, their access frequencies are less than 10 per second, which means that

Author's personal copy

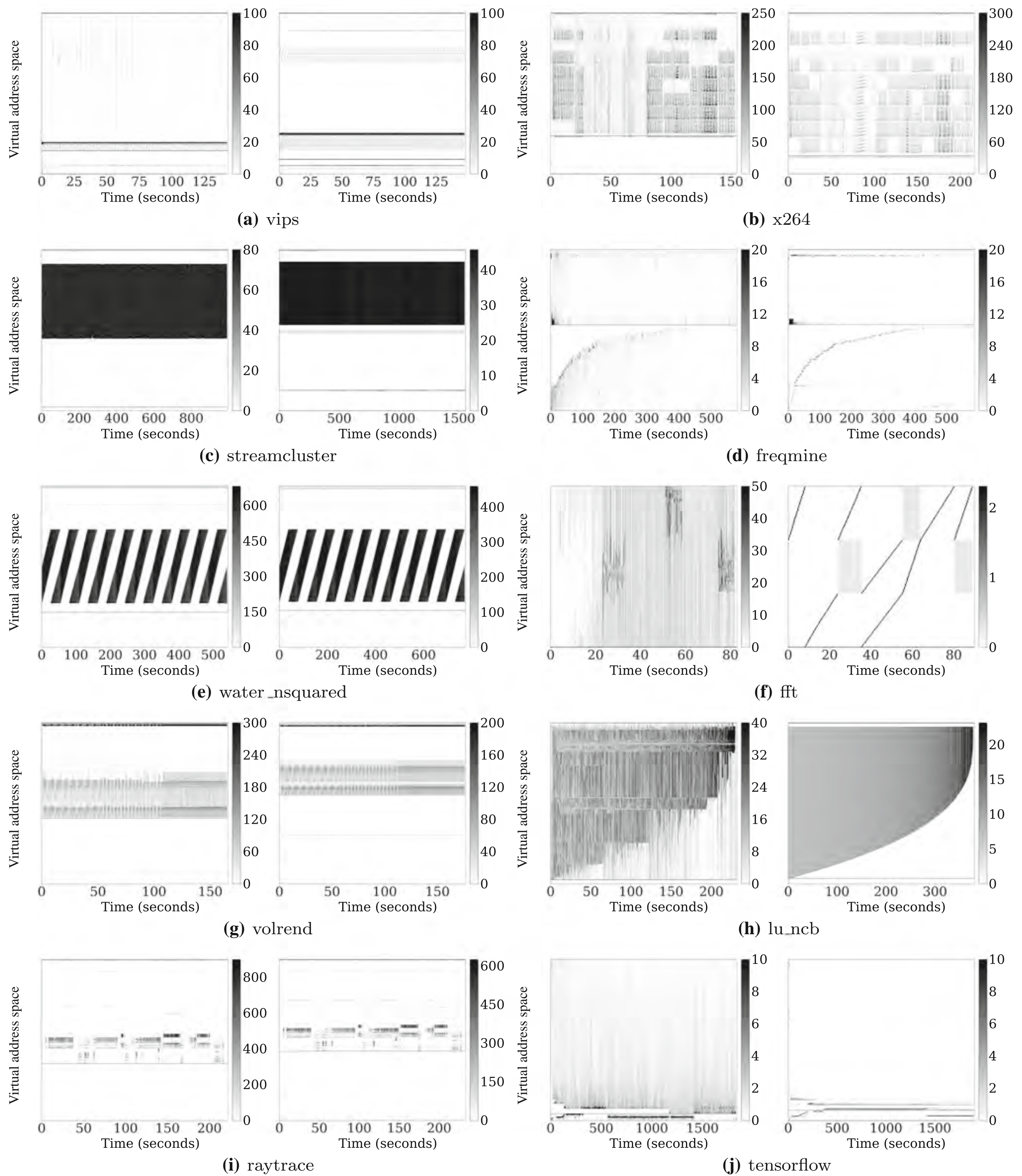


Fig. 6 Visualized trace of workloads. (cont.)

they are not hot areas. In the case of temporally hot areas, Daptrace also performs mostly well at identifying those areas. 433.milc (Fig. 5b), water_nsquared (Fig. 6e) and raytrace (Fig. 6i) show such cases. It is easy to see that hot areas in the three workloads change dynamically with unique patterns. The access patterns in x 264 (Fig. 6b) are also tracked well overall, but the pattern becomes a little blurred during an interval when the memory access

Author's personal copy

frequency is low. freqmine (Fig. 6d) and lu_ncb (Fig. 6h) also show dynamic access patterns, but they are slightly ambiguous due to relatively low access frequencies. Similarly, fft (Fig. 6f) shows three blurry hot areas for the same reason.

The trace results give another insight into Daptrace. In freqmine (Fig. 6d) and lu_ncb (Fig. 6h), we can see that Daptrace follows the change of access patterns quite properly. On the other hand, in fft (Fig. 6f), Daptrace fails to follow the sequential access pattern. The main reason fft is not properly tracked is that only a small area is hot at every moment and the area changes too rapidly, considering the virtual memory size of fft. From this observation, we can see that Daptrace follows dynamic access patterns with modest changes, and it does not follow rapidly changing access patterns. Nevertheless, such an access pattern is far from being a hot object. In fact, it is a memory-polluting access pattern. Thus, it would be rather wrong to identify such memory areas as hot.

In short, Daptrace properly identifies consistently hot memory areas and tracks dynamic access patterns as well. However, it does not identify rapid access patterns such as polluting access patterns. Moreover, Daptrace shows higher accuracy for frequently accessed areas, and we observe that the result is mostly accurate if access frequency is above 30.

4.3 Tracing overhead

We measured the runtime overhead and space overhead of Daptrace and compared the overheads with the baseline. Trace data visualized in Fig. 5 are used for the evaluation.

4.3.1 Runtime overhead

Figure 7 shows the runtime overhead of Daptrace and the baseline. Daptrace incurred only a small amount of overhead except for two cases: 458.sjeng and streamcluster. 458.sjeng and streamcluster showed an overhead of 7.97% and 20.0% respectively due to their cache sensitivity. Cache hit ratio decreased by 10.5% and 23.7% in 458.sjeng and streamcluster, respectively; the ratio decreased by only 0.28% in the other workloads. Nevertheless, the overhead of Daptrace is 1.95% on average, and it is 0.71% except for the two cases.

On the other hand, the overhead of the baseline is 89.5%, and it varies according to workload, Note that programs with large virtual memory do not always indicate large overheads. Indeed, the virtual memory size affects the quality of tracing rather than runtime overhead when the virtual memory size reaches a certain point. For example, fft shows relatively low runtime overhead, but the huge virtual memory size results in infrequent access monitoring, which makes access frequency observed even smaller than Daptrace. Nevertheless, the average overhead of the baseline is 27.7%, which is 14.2 $\times$ larger than Daptrace.

Figure 8 shows the overhead sensitivity to the size of workloads. We measured the runtime overhead of Daptrace by workload input type. The memory size of each workload by input type is specified in Fig. 3. The runtime overhead of each workload is maintained low except streamcluster. The overhead of streamcluster of input classes A, B and C are -3.4%, -0.7% and 20.0%, respectively. As mentioned before, streamcluster shows higher overhead due to its cache sensitivity. The average overheads of Daptrace with input classes A, B and C are 1.7%, 1.6% and 2.7%, respectively; they are 1.9%, 1.7% and 1.8% without streamcluster. Note that the average memory sizes of workloads with input classes A, B and C are 85 MiB, 128 MiB and 364 MiB, respectively. This shows that the runtime overhead of Daptrace is controlled low even with the large workload sizes.

4.3.2 Space overhead

In the context of space overhead, there are two meanings: memory space consumed during tracing and storage space required to store the trace result. Since it is difficult to measure the runtime memory usage of kernel modules, we focus on how much memory is required for core internal structures.

Both Daptrace and the baseline use the same structure for tracing. They use a result buffer of size 4 MB to avoid

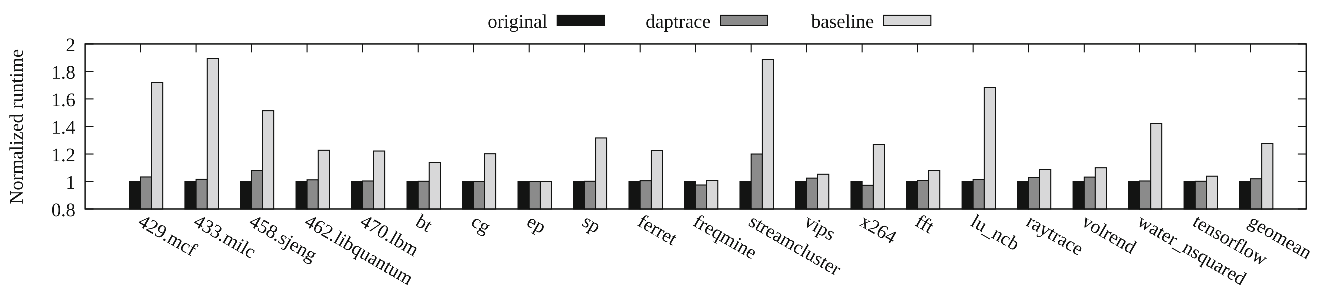


Fig. 7 Runtime overhead comparison. Class C input types are used

Author's personal copy

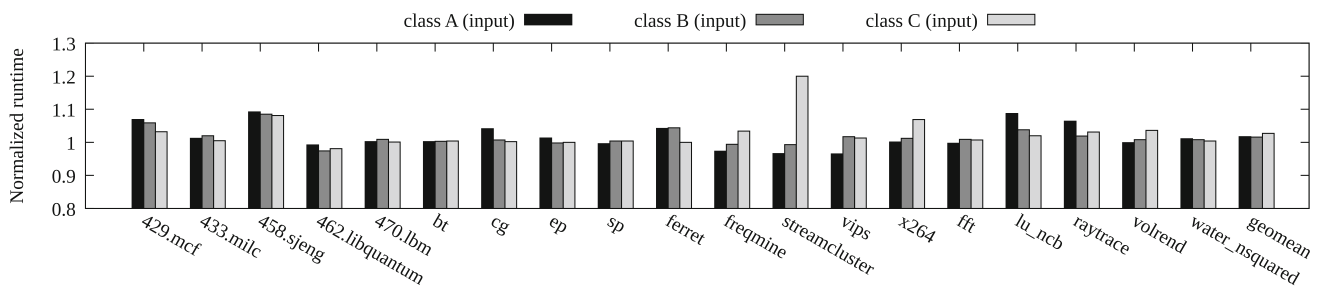


Fig. 8 Runtime overhead sensitivity to workload size

Table 3 Memory size of workloads by input class. We used virtual memory size averaged by time as memory size

Workload	Class A (MiB)	Class B (MiB)	Class C (MiB)
429.mcf	104	261	1680
433.milc	16	92	667
458.sjeng	179	179	179
462.libquantum	8	9	72
470.lbm	417	417	417
bt	428	727	1888
cg	277	752	1429
ep	313	326	329
sp	408	654	1606
Ferret	463	492	532
Freqmine	55	89	580
Streamcluster	91	98	200
Vips	112	112	144
× 264	22	22	93
fft	202	779	12,299
lu_ncb	18	42	510
Raytrace	51	51	51
Volrend	16	17	53
Water_nsquared	15	19	38
Geomean	85	128	364

frequent writing and use per-region structures of size 44 B each to maintain the statistics of regions. Thus, both consume at most 4.044 MB of memory space for tracing. Other memory consumptions such as kernel APIs, page caches are not covered in here.

On the contrary, Daptrace and the baseline show different storage space overhead. Figure 9 shows such overhead. In both cases, the trace size increases as the runtime of a workload increases. However, the overhead of the baseline is 98.4 × larger at maximum (sp), 1.87 × larger at minimum (volrend), and 20.6 × larger on average. The average overhead of Daptrace and the baseline are 5.38 MB and 111.24 MB, respectively.

4.4 Case study: out-of-core memory management

Out-of-core memory management is one of the schemes that make it possible to process data larger than main memory. Least recently used (LRU) scheme is widely used for the management, however, it does not always provide optimal performance. To show how useful Daptrace is in this case, we optimized various workloads using Daptrace for optimal out-of-core memory management, and we compared the runtime with original workloads.

9 workloads are chosen out of the 20 workloads we traced: 433.milc, 462.libquantum, 470.lbm, cg, sp, ferret, water_nsquared, fft and volrend. We simulate memory pressure simulation using cgroups [23]. The available memory was set to 70% of the working set size of a workload, which is a realistic case in cloud services where memory overcommitment of 1.5 × is recommended [24]. In this situation, we manually lock hot memory areas in the main memory using `mlock()` system call [25]. If it is not

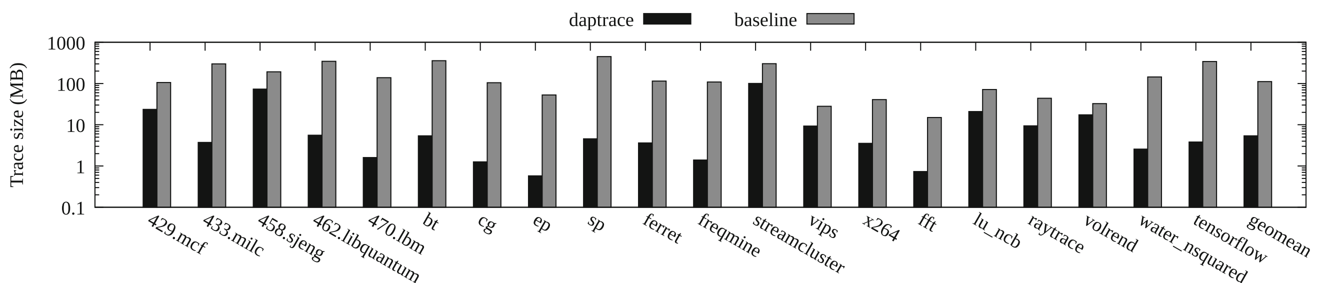


Fig. 9 Storage overhead comparison. Class C input types are used

Author's personal copy

possible to lock the entire hot memory area, we lock only a portion of the hot memory area. Locked areas in each workload are explained in Table 4.

Figure 10 shows the performance improvement of optimized workloads. Under memory pressure, the speedup varies by workloads from 1.08 × (sp) to 2.55 × (cg) and shows 1.58 × on average. We also measure the overhead of optimization under no memory pressure (Fig. 11). The overhead is 7.2% (ferret) at maximum and 0.9% on average.

This overhead and performance improvement show high potential for the application of Daptrace in out-of-core memory management; such high speedup is obtained by locking only some hot areas in memory statically during runtime. We expect that more speedups can be achieved with dynamic memory management using Daptrace. Similar cases such as heterogeneous memory are considered beneficiaries of Daptrace as well.

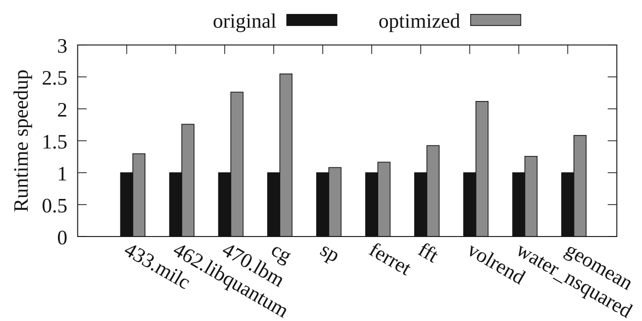


Fig. 10 Performance of workloads (70% memory)

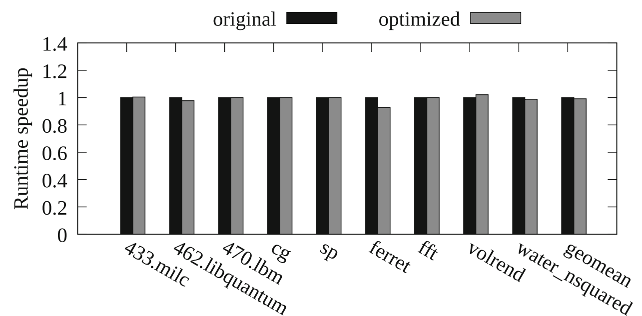


Fig. 11 Performance of workloads (full memory)

5 Related works

The high overhead of memory tracing is a critical issue in a wide range of fields such as profiling and development, thus it has been considered an important problem to reduce the overhead.

Spindle [7] has efficiently reduced the overhead incurred during instrumentation-based memory access tracing, using the static analysis of a workload that extracts predictable memory access patterns. The predictable patterns are exploited during dynamic analysis to reduce the number of instrumentation points. Compared to a well-known instrumentation-based tool, Pin [6], Spindle achieves 61 × speedup on average. However, the need for compile-time analysis limits its range of applications. memTrace [26] is also a memory tracer, and it targets a lightweight tracing of × 86 workloads. It focuses on the mismatch where a large portion of × 86 workloads is running on 64-bit processors, which results in a surplus of hardware registers. The tracer utilizes these registers for tracing × 86 workloads with runtime cross-ISA translation and achieves a low tracing overhead, 1.97 × on average. The work has a clear limitation in that it can only be applied to × 86 applications running on × 64 processors, but it is quite impressive to get help from hardware with no additional hardware support. These two works show relatively small overheads, however, the overheads are still easily observed because they interfere with the execution of workloads directly.

Xiao et al. [27] have studied page coloring-based cache management, with a memory tracing technique based on access bit tracking. Their cache management scheme colors hot pages identified by memory tracing. Because full-page access bit tracking incurs high overhead, it exploits spatial locality to reduce the number of pages to track. Their sampling-based tracking technique induces overheads of 7.1% with 10 ms sampling interval and 1.9% with

Table 4 Memory areas locked for optimization

Workload	Locked area
433.milc	Upper hot area
462.libquantum	Half of hot area in the middle
470.lbm	Half of whole area
cg	Bottom hot area
sp	Hottest bottom area and hot area in the middle
Ferret	75% of hot area at the top
Water_nsquared	40% of hot area in the middle
fft	Two large hot areas at the top and the middle
Volrend	Small hot region at the top and lower portion of warm area in the middle

Author's personal copy

100 ms sampling interval. However, their sampling technique does not guarantee that the number of pages to be tracked has an upper limit; it can increase as a program size grows. Moreover, the technique relies on memory access locality, which means the sampling technique might be inaccurate with modern computing workloads [1–3].

6 Conclusion

Due to the characteristics of modern computing workloads, huge working sets and low locality, memory pressure is becoming more prevalent. On the other hand, the capacity of memory devices has been growing at a much slower rate than that of CPU and storage devices. These two trends have led to main memory becoming the performance bottleneck of a system.

Several memory schemes have been proposed to mitigate this problem, which utilize various type of memory devices to give an illusion of large and fast main memory. The performance of these schemes relies on how effectively it identifies hot data, which can be obtained from memory tracing. However, the high overhead of memory tracing prevents using the tracers.

This paper presents Daptrace, a lightweight memory tracer which keeps the overhead in a limited range, regardless of the size of a workload. Our evaluations show that Daptrace incurs very small overhead (1.95%) and efficiently identifies hot data in memory. We also conducted a case study on out-of-core memory management, and Daptrace is observed to have high potential for application in such cases.

Acknowledgements This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF2015M3C4A7065646 and No. 2017R1A2B4005681).

References

1. Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A.D., Ailamaki, A., Falsafi, B.: Clearing the clouds. In: Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, volume 47 of *ASPLOS*. ACM Press, New York, USA, p. 37 (2012)
2. Basu, A., Gandhi, J., Chang, J., Hill, M.D., Swift, M.M.: Efficient virtual memory for big memory servers. ACM SIGARCH Comput. Architect. News **41**, 237–248 (2013)
3. Dulloor, S.R, Roy, A., Zhao, Z., Sundaram, N., Satish, N., Sankaran, R., Jackson, J., Schwan, K.: Data tiering in heterogeneous memory systems. In: Proceedings of the 11th European Conference on Computer Systems (EuroSys). ACM, p. 15 (2016)
4. Nitu, V., Teabe, B., Tchana, A., Isci, C., Hagimont, D.: Welcome to zombieland: practical and energy-efficient memory disaggregation in a datacenter. In: Proceedings of the 13th European Conference on Computer Systems (EuroSys). ACM, p. 16 (2018)
5. Intel's 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 (2017)
6. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. Acm Sigplan Notices **40**, 190–200 (2005)
7. Wang, H., Zhai, J., Tang, X., Yu, B., Ma, X., Chen, W.: Spindle: Informed memory access monitoring. In: 2018 *USENIX* Annual Technical Conference (ATC). USENIX Association, Boston, MA, pp. 561–574 (2018)
8. Snavely, A., Carrington, L., Wolter, N., Labarta, J., Badia, R., Purkayastha, A.: A framework for performance modeling and prediction. In: SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing. IEEE, pp. 21–21 (2002)
9. Hauswirth, M., Chilimbi, T.M.: Low-overhead memory leak detection using adaptive statistical profiling. Acm SIGPLAN Notices **39**, 156–164 (2004)
10. Extrae user guide. https://tools.bsc.es/sites/default/files/documentation/extrae-3.2.1-user-guide.pdf (2015)
11. Chang, P.P., Mahlke, S.A., Hwu, W.M.W.: Using profile information to assist classic code optimizations. Software **21**(12), 1301–1321 (1991)
12. Pettis, K., Hansen, R.C: Profile guided code positioning. In: ACM SIGPLAN Notices, vol. 25. ACM, pp. 16–27 (1990)
13. Jaleel, A.: Memory characterization of workloads using instrumentation-driven simulation. http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf (2007)
14. 433.milc, SPEC CPU2006 Benchmark Description. https://www.spec.org/cpu2006/Docs/433.milc.html (2011)
15. Waldspurger, C., Saemundsson, T., Ahmad, I., Park, N.: Cache modeling and optimization using miniature simulations. In: 2017 *USENIX* Annual Technical Conference (ATC). USENIX Association, Santa Clara, CA, pp. 487–498 (2017)
16. Lagar-Cavilla, A., Ahn, J., Souhlal, S., Agarwal, N., Burny, R., Butt, S., Chang, J., Chaugule, A., Deng, N., Shahid, J., Thelen, G., Yurtsever, K.A., Zhao, Y., Ranganathan, P.: 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, pp. 317–330 (2019)
17. Servat, H., Peña, A.J, Llort, G., Mercadal, E., Hoppe, H.-C., Labarta, J.: Automating the application data placement in hybrid memory systems. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, pp. 126–136 (2017)
18. Evans, J.: A scalable concurrent malloc (3) implementation for freebsd. In: Proc. of the bsdcan conference, Ottawa, Canada (2006)
19. Clarke, S., Walker, R.J: Composition patterns: an approach to designing reusable aspects. In: Proceedings of the 23rd international conference on Software engineering. IEEE Computer Society, pp. 5–14 (2001)
20. Liaw, A., Wiener, M., et al.: Classification and regression by randomforest. R News **2**(3), 18–22 (2002)
21. Krizhevsky, A., Hinton, G.: Learning multiple layers of features from tiny images. Technical report, Citeseer (2009)
22. Cifar-10 tensorflow benchmark. https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10 (2019)
23. Memory Resource Controller. https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt (2019)
24. Overcommitting CPU and RAM. https://docs.openstack.org/arch-design/design-compute/design-compute-overcommit.html (2018)

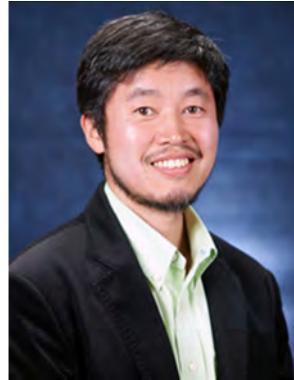
Author's personal copy

25. mlock(2): Linux manual page. http://man7.org/linux/man-pages/man2/mlock.2.html (2019)
26. Payer, M., Kravina, E., Gross, T.R: Lightweight memory tracing. In: Presented as part of the 2013 USENIX Annual Technical Conference (ATC 13), pp. 115–126 (2013)
27. Zhang, X., Dwarkadas, S., Shen, K.: Towards practical page coloring-based multicore cache management. In: Proceedings of the 4th ACM European conference on Computer systems. ACM, pp. 89–102 (2009)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Yunjae Lee is a M.S. student of the School of Computer Science and Engineering, Seoul National University. He received B.S. degree in Electrical and Computer Engineering from Seoul National University in 2018. His research interests include resource management, especially memory management in operating systems.

Yoonhee Kim she is the professor of Computer Science Department at Sookmyung Women's University. She received her Bachelors degree from Sookmyung Women's University in 1991, her Master degree and Ph.D. from Syracuse University in 1996 and 2001, respectively. She was a Research Staff Member at the Electronics and Telecommunication Research Institute during 1991 and 1994. Before joining the faculty of Sookmyung Women's University in 2001, she was the faculty of Computer Engineering dept. at Rochester Institute of Technology in NY, USA. Her research interests span many aspects of runtime support and management in distributed computing systems. She is a member of IEEE and OGF, and she has served on variety of program committees, advisory boards, and editorial boards.



Heon Y. Yeom is a Professor with the School of Computer Science and Engineering, Seoul National University. He received B.S. degree in Computer Science from Seoul National University in 1984 and his M.S. and Ph.D. degrees in Computer Science from Texas A&M University in 1986 and 1992 respectively. From 1986 to 1990, he worked with Texas Transportation Institute as a Systems Analyst, and from 1992 to 1993, he was with Samsung Data Systems as a Research Scientist. He joined the Department of Computer Science, Seoul National University in 1993, where he currently teaches and researches on distributed systems and transaction processing.