Load-Site-Based Filtering of Transiently Hot Objects to Reduce the Effective Working Set

Naoki Nakanishi
okinai1012@gmail.com
The University of Tokyo
Japan

Takato Hideshima
hideshima-takato182@g.ecc.u-tokyo.ac.jp
The University of Tokyo
Japan

Tomoharu Ugawa
tugawa@acm.org
The University of Tokyo
Japan

Abstract

Moving garbage collection (GC) provides opportunities to rearrange object placement to reduce the working set and thereby enhance cache locality. Hot-Cold Objects Segregation GC (HCSGC) identifies objects whose reference are loaded through load reference barrier in ZGC as *hot objects* and segregates them from the other objects. If such hot objects are repeatedly accessed after segregation, the effective working set can shrink. However, we found that a substantial fraction of hot objects were not accessed again after experiencing a few, or even no, accesses immediately after the segregation. This paper presents load-site based filtering (LSBF) to further reduce the working set. Our key observation is that certain load sites tend to load references to *persistently hot objects*, while others tend to load references to *transiently hot objects*. We implemented LSBF in HCSGC, where we classified load sites using offline profiling. Our evaluation showed that LSBF reduced the working set for our synthetic benchmark where HCSGC failed to reduce the working set. For the DaCapo benchmarks, we did not observe improvements by LSBF mainly because these programs were small enough that the working set fit within the cache capacity regardless of object arrangements. Nevertheless, we confirmed that the overhead of LSBF was limited.

CCS Concepts: • **Software and its engineering → Garbage collection**.

Keywords: Garbage Collection, Locality, Cache Optimization

ACM Reference Format:
Naoki Nakanishi, Takato Hideshima, and Tomoharu Ugawa. 2026. Load-Site-Based Filtering of Transiently Hot Objects to Reduce the Effective Working Set. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '26), June 30, 2026, Brussels, Belgium.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3828170.3828179

MPLR '26, Brussels, Belgium

ACM ISBN 979-8-4007-2433-6/2026/06
https://doi.org/10.1145/3828170.3828179

1 Introduction

Moving garbage collection (GC) provides opportunities to rearrange object placement to reduce the working set and thereby enhance cache locality. Much work has explored relocating objects using moving GC to create cache-friendly object layouts. Some approaches group objects that tend to be accessed together and place objects in the same group close to each other [12]. Others predict the access frequency of objects and place frequently accessed objects close to each other [3, 6].

Hot-Cold Objects Segregation GC (HCSGC) [16] identifies objects whose references are loaded through the load reference barrier (LRB) in ZGC [17] as *hot objects* and relocates them close to each other, segregating them from other objects. If such hot objects are repeatedly accessed after being segregated, the effective working set can shrink.

However, we revealed that a substantial fraction of hot objects were not accessed again after experiencing a few, or even no, accesses immediately after segregation (section 2). According to our analysis of the behavior of the DaCapo benchmarks [2], on average, 23.4% of relocations caused by the LRB are estimated to relocate *transiently hot objects*, which are not accessed after a while. As a result, *persistently hot objects* and cooled transiently hot objects tend to become neighbors, leading to a suboptimal working set.

Further investigation revealed that we can use load sites to distinguish between persistently and transiently hot objects. We hypothesized that references loaded at the same load site tend to follow the same data flow and that the future behavior of the referenced objects is also similar. Indeed, certain load sites tend to load references to persistently hot objects, while others tend to load references to transiently hot objects (section 3).

Based on this observation, we present load-site-based filtering (LSBF) to further reduce the working set (section 3). LSBF prevents transiently hot objects from being relocated close to persistently hot objects. To this end, LSBF classifies load sites into *persistently-hot load sites*, which tend to load references to persistently hot objects, and *transiently-hot load sites*, which tend to load references to transiently hot objects.

We implemented LSBF in HCSGC (section 4) as a profiling-guided optimization [5]. We profile the behavior of the program offline and classify its load sites based on profiling results. At run time, we assign two to-space pages, a *hot page* and a *cold page*, to each mutator thread. The classification is supplied to the JIT compiler so that *persistently-hot load sites* and *transiently-hot load sites* are compiled to relocate objects to the hot page and cold page unconditionally, respectively.

Our evaluation showed that LSBF reduced the working set for our synthetic benchmark where HCSGC failed to reduce the working set. For the DaCapo benchmarks, we did not observe improvements by LSBF mainly because these programs were small enough that the working set fit within the cache capacity regardless of object arrangements. Nevertheless, we confirmed that the overhead of LSBF was limited.

Our main contributions are summarized as follows:

- We show that HCSGC can mix transiently hot objects with persistently hot objects, resulting in a suboptimal working set (section 2).
- We show that transiently hot objects can be identified based on the load site that relocates objects and can be segregated from persistently hot objects (section 3).
- We implement LSBF of transiently hot objects in HCSGC (section 4).

2 Motivation

2.1 Cache and Working Set

The gap between processor speed and main-memory performance is known as the memory wall [15]. For memory-intensive workload, the CPU spends a significant fraction of time stalled, waiting for data to arrive from DRAM. Cache hierarchies mitigate this gap by exploiting access locality.

Modern CPUs are equipped with a three-level cache hierarchy: L1 and L2 caches, and a last-level cache (LLC). Each cache is organized into fixed-size *cache lines*, which are typically 64 bytes. From the perspective of software, if frequently accessed data are placed close to each other, programs can benefit from caches. This is because memory accesses are amplified to cache-line granularity: accessing a single word fetches the entire cache line.

Suppose that there are many 32-byte data structures, some of which are accessed frequently (hot) and the others are not (cold). If each cache line stores two hot data structures, the cache can store twice as many hot data structures as in the case where each cache line stores one hot and one cold data structure. According to Dieckmann et al. [4], the average sizes of instance objects varies between 16 and 23 bytes for Java programs in the SPECjvm98 benchmark suite, under a configuration where fields are aligned to 32 bits. Furthermore, even when the sizes of data structures exceed the cache line size, placing frequently accessed data close to each other is still beneficial because the unused space in the first and last cache lines of hot data structures can host parts of other hot data structures.

In addition to cache memory, modern CPUs are equipped with translation lookaside buffers (TLBs). When a CPU accesses memory, it translates a virtual address to a physical address. Virtual and physical addresses are mapped page by page, and the mappings are stored in multiple levels of page tables that resides in DRAM. This means that multiple extra memory accesses are carried out for each load/store instruction. TLBs cache these translations, eliminating these extra memory accesses. When the CPU accesses an address in a page whose translation is not cached in the TLB, the page tables are accessed, and the translation result replaces an existing TLB entry.

Again, from the perspective of software, if hot data structures are placed close to each other, programs can benefit from the TLB because memory accesses are regarded to be amplified to page granularity. If hot data structures are placed close to each other, they reside in fewer pages compared to the case where they are scattered across memory. Therefore, TLB entries are less likely to be evicted, and TLB miss rates decreases.

We define the *effective working set* during a time interval as the set of memory regions that are accessed after amplification by the memory hierarchy. Placing hot data structure close to each other reduces the size of the effective working set. This is the case for any kind of cache that amplifies memory accesses. In the rest of this paper, we use the term cache to also refer to TLB.

2.2 ZGC and Load Reference Barrier

ZGC [17] is a mostly-concurrent mark-and-copy GC. Notably, mutators continue running during the evacuation phase in ZGC.

In the evacuation phase, the collector evacuates the live objects from the pages selected in the previous phase. During the evacuation phase, mutators use the *load reference barrier* (LRB) to maintain the to-space invariant similar to Baker's relocation [1]. Under this invariant, mutators always access to-space objects. Furthermore, the mutators are *black*; no references in the stack refer to from-space objects (objects in the selected pages). When a mutator attempts to load a reference to a from-space object, it converts the reference to the reference to the copy in to-space of the object. If the object has already been copied to to-space, the reference is converted using forwarding information stored in a table. However, if it has not, the mutator copies the object on its own.

2.3 HCSGC: Hot-Cold Segregation GC

Hot-Cold Objects Segregation GC (HCSGC) [16] is a garbage collector that aims to improve locality by exploiting the property of the LRB of ZGC that mutators relocate objects. Because a mutator must obtain a reference to an object to

access it, loading a reference to the object typically precedes accesses to it. Thus, reference loads can approximate accesses to the referenced objects. Therefore, objects relocated by the LRB can be regarded as objects that have been accessed. Assuming that objects that are accessed once are likely to be accessed again, such objects are regarded as frequently accessed objects, or *hot objects*.

In ZGC, each mutator relocates objects to a thread-local to-space page. Thus, objects relocated by mutators are placed in pages that are separate from those used by the collector for evacuation. Furthermore, objects in the to-space are allocated using bump-pointer allocation. As a result, relocated hot objects are placed close to each other and spatially segregated from others.

HCSGC encourages this segregation (1) by selecting all pages for small objects for evacuation and (2) by enabling the LRB even when the collector is inactive. As a result, whenever a mutator loads a reference to an object for the first time since the last GC cycle, the object is relocated to the mutator's to-space page. Hence, it is relocated close to other relocated objects. If such objects continue to be accessed while other objects are not, the effective working set remains minimal. Note that this excessive evacuation itself is an overhead applied to the mutators.

2.4 Transiently Hot Objects

HCSGC depends on the assumption that relocated objects are continuously accessed. To examine whether this assumption holds, we analyze the behaviour of objects in the DaCapo benchmark suite [2].

We say that a relocation of an object is *consumed* if the object is accessed after a certain *transient time*, T, or later. Hereafter, time is measured in terms of the number of object accesses. The transient time models the period until data is evicted from the cache if it is not accessed. When an object is relocated, it is brought into the cache if it has not resided. We assume it stays in the cache during the transient time after the relocation. If the object or its neighboring objects are accessed during the transient time, the object stays in the cache even after the transient time.

The experimental setup is as follows (Details are the same as in section 5). We record reference-load events, object-access events, and GC events during executions of DaCapo benchmarks using the JVMTI interface of OpenJDK 14. We use HCSGC for the executions. Events are sampled. We sample one objects per 1024 bytes of allocation at allocation time and track only those objects. We record only reference-load events to tracked objects stored in fields of tracked objects, as well as object-access events on tracked objects. For tracked objects, we record all object-access events so that we do not miss accesses to objects whose reference-load events are recorded. In this experiment, we track only scalar objects; arrays are not tracked. Accesses to arrays are also ignored when counting the transient time.

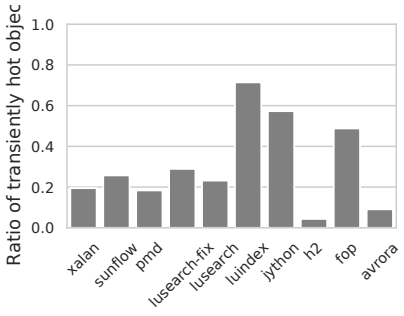

Figure 1. Ratio of unconsumed relocations ($T = 10,000$).

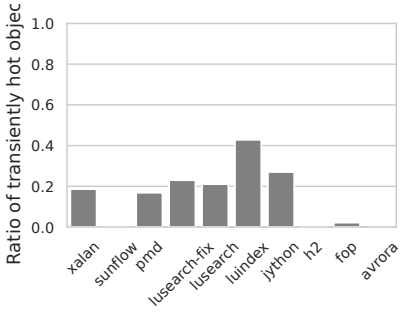

Figure 2. Ratio of unconsumed relocations ($T = 0$).

We simulate relocations of objects performed by mutators according to the recorded events. We regard a recorded reference-load event to object O as a *simulated relocation event* if it is the first such event since the last GC cycle. Note that O may be relocated earlier in the real execution when a reference to O is loaded from a field of an untracked object.

For each simulated relocation event of O, we examine whether the relocation is consumed. References to a single object O may be loaded multiple times. In our experiments, all simulated relocation events are considered consumed even if O is accessed only once after the last simulated relocation event of O.

Figure 1 shows the ratio of simulated relocation events that are *not* consumed when the transient time $T = 10,000$. This result revealed that a certain fraction of objects quickly became cold after relocation. We call such objects *transiently hot objects*. We also found that some objects are never accessed after relocation, as shown in fig. 2, which shows the ratio of unconsumed simulated relocation events when $T = 0$. In other words, some reference loads were not followed by accesses to their referents.

Because HCSGC relocates both transiently hot objects and *persistently hot objects*, which continue to be accessed after the transient time, the density of hot objects in the pages that have been mutator's to-space pages degrades. Although persistently hot objects in those pages will be relocated again after the next GC cycle, they are temporarily mixed with

transiently hot objects that have cooled down. This increases the effective working set size.

3 Load-Site-Based Filtering

To reduce the effective working set, we aim to create an area containing only persistently hot objects, namely *hot pages*, by filtering transiently hot objects during object relocation within the load reference barrier. This is because accesses to transiently hot objects, which occur only immediately after relocation, hit the cache lines fetched upon writes for relocation and thus do not require special treatment. Furthermore, considering the memory layout after the transient time has elapsed since relocation, we have a motivation to filter these out from the hot pages. The reason is that by that point, transiently hot objects have become cold, therefore mixing them with persistently hot objects during relocation means intermingling cold objects among hot objects, which increases the effective working set.

To identify objects to filter within the load reference barrier, we use information called load sites, the locations in the program where references to an object were loaded. This stems from our intuition that references loaded at the same load site tend to follow the same data flow, implying similar future behavior for the referenced objects.

Specifically, we model load sites as either *persistently-hot load sites*, which mostly load references to persistently hot objects, or *transiently-hot load sites*, which mostly load references to transiently hot objects. Depending on which one the load reference barrier occurs during, we relocate the object to different pages within the mutator's to-space pages. The area chosen for relocation during persistently-hot load sites becomes the aforementioned hot pages.

The design of using load sites for decisions within load reference barriers also has the advantage of not needing to record past behavior of objects. Compared to other decision criteria (such as allocation sites or access counts), it is expected to reduce space overhead.

3.1 Effectiveness of Load Sites

We experimentally confirmed that the load sites serve as a useful indicator for assessing whether an object is persistently hot or transiently hot. Specifically, we compared them with allocation sites—the locations in the program where an object was created—which are known to correlate closely with object behavior [7]. The benchmarks and heap sizes used in the experiment are the same as those in section 2.4.

Figure 3 shows the relationship between the load/allocation site referenced during object relocation within the load reference barrier and whether the object is persistently hot or transiently hot. To highlight impactful sites, the figure excludes sites with fewer than 0.5% of total references. The charts for load sites and allocation sites are similar. It was demonstrated that they have equivalent effectiveness for the purpose of filtering transiently hot objects.

4 Implementation

We implemented load-site-based filtering (hereinafter referred to as LSBF) of transiently hot objects, as proposed in section 3, within HCSGC. We identified transiently hot load sites by profiling program behavior offline. Specifically, we tracked objects relocated at each load site using JVMTI in the same way as in section 2.4 and classified sites where over 99% of the objects were transiently hot as transiently-hot load sites. Our system accepts a list of the transiently-hot load sites as an option at runtime, and treats any sites not included in that list as persistently-hot load sites.

While the mutator threads hold a single to-space page in HCSGC, they now hold two: one is a *hot page*, and the other is a *cold page*. When relocating an object within load reference barriers, if the current load site is persistently-hot, the destination is in the hot page; if it is transiently-hot, it is in the cold page. When a page becomes full, the mutator thread switches to a newly allocated page.

By informing the JIT compiler of the load sites' classification, we make it possible to unconditionally select the destination for object relocation within the compiled code. Since the classification is determined offline, the destination does not change at runtime for each load site. We therefore embed code that unconditionally relocates objects into the hot page for the compilation results of persistently-hot load sites, and code for unconditional relocation into the cold page for transiently-hot load sites. This is expected to reduce overhead of LSBF.

5 Evaluation

We evaluate our approach to answer the following research questions:

RQ1: Does LSBF reduce the effective working set?
RQ2: Does LSBF improve program throughput?
RQ3: Does LSBF introduce performance overhead compared to HCSGC?

5.1 Methodology

We expect that LSBF reduces the effective working set for some of the programs for which HCSGC fails to sufficiently reduce it compared to ZGC. Thus, we compare three GC algorithms: ZGC, HCSGC, and HCSGC with LSBF (HCSGC+LSBF).

To judge whether the effective working set of one algorithm is smaller than that of another, we compare the number of cache misses. If we plot the number of cache misses against the effective working set size, the curve consists of two segments. As long as the effective working set fits within the cache capacity, the program exhibits few cache misses. However, the curve has a knee: once the effective working

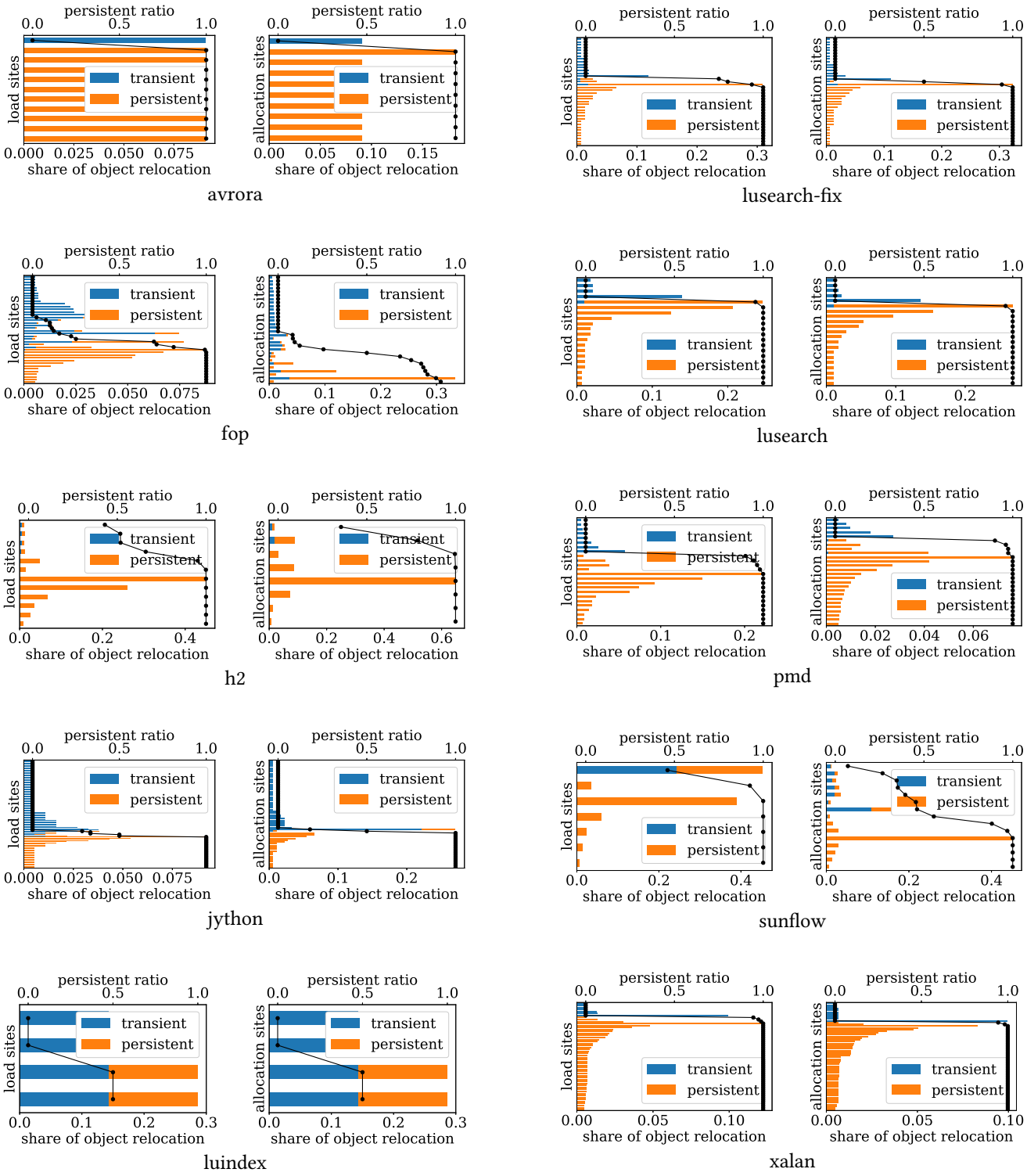

Figure 3. Ability of load/allocation sites to distinguish between persistently and transiently hot objects in each benchmark program. The line charts show the persistent ratio, or the ratio of relocations of persistently hot objects to total relocations referencing each site (top horizontal axis). The bar charts shows the share of object relocations for each site in the benchmark (red/blue indicating relocation of persistently/transiently hot objects; bottom horizontal axis). Sites with a share below 0.005 are excluded, and the remaining sites are sorted in ascending order by persistent ratio.

set exceeds the cache capacity, the number of cache misses increases with the effective working set size.

We develop a synthetic benchmark program that is designed so that HCSGC fails to minimize the effective working set. The program is parameterized by a *size factor*, which represents the number of accessed objects and objects whose references are loaded. As the size factor increases, the effective working set size increases. We plot the number of cache misses against the size factor and compare the size factors at which the knee appears in the curves for different GC algorithms.

We compare the effective working set with respect to the L1 cache, the LLC and TLBs. We have defined the effective working set during a time interval as the set of memory regions that are accessed after amplification by the memory hierarchy in section 2. For CPU caches, accesses are amplified to the cache-line size, which is 64 bytes in our environment. For TLBs, accesses are amplified to the page size, which is 4 KB in our environment; we disable large pages in our experiments.

We also measure the execution time for various size factors to examine whether LSBF improves program throughput.

Additionally, we compare cache misses and execution time using the DaCapo benchmark suite [2] (dacapo-9.12-MR1-bach). We exclude batik, eclipse, tomcat, tradebeans, and tradesoap because they were not able to run with HCSGC. Norlinder [9], which proposes another improvement of HCSGC, also excluded these benchmarks. Although Norlinder [9] describes that "DaCapo programs are probably small enough to fit in the L3 cache of our benchmark machine," we examine DaCapo benchmarks mainly to evaluate performance overhead. Also, our machine has only 8.3 MB shared LLC while theirs has a 64 MB LLC.

We use the default input size for all programs. We also use a *huge* input size for `h2`, which is the only program that supports this size. We denote `h2` with the *huge* input as `h2huge`.

We execute 10 iterations in a single process instance for each program except `h2huge`. For `h2huge`, we execute a single iteration because its execution time is sufficiently long. We measure execution time and cache misses over the entire execution of the program using the Linux `perf` profiler.

5.2 Experimental Setup

All experiments were conducted on OpenJDK 14, on top of which HCSGC is implemented[1], with the following command-line options:

```
-XX:+UnlockExperimentalVMOptions
-XX:-UseNUMA
-XX:+TieredCompilation
-XX:TieredStopAtLevel=1
-XX:+UseZGC
-XX:+UseColdPage†
-XX:HotCycles=1†
-XX:ColdConfidence=0†
-XX:+UseRelocateAllSmallPages†
-XX:+UseLazyRelocate†
```

The options marked with † were not supplied to ZGC. The JVM was restricted to using the C1 JIT compiler because we implemented LSBF only in the C1 compiler. For HCSGC+LSBF, we supplied the list of transiently-hot load sites as an additional command line option. The list of transiently-hot load sites was generated by an offline profiling of the entire execution of the same benchmark program. Therefore, the programs behaved exactly the same during profiling and evaluation.

The hardware and OS used in our experiments are summarized in table 1, and the specifications of the caches are listed in table 2. The page size was always 4 KB because we used the default setting of OpenJDK, which does not enable large-page support. Thus, the L2 TLB can cover 6 MB of memory.

We set the heap size to 3800 MB for the synthetic benchmark so that GC takes place occasionally. For DaCapo benchmarks, table 3 summarizes the heap sizes used for profiling to classify load sites and the minimum heap sizes. The minimum heap size is the smallest heap size for which the program completes within 10× the execution time when running with a sufficiently large heap using HCSGC. For profiling, we set the heap sizes so that GC takes place occasionally. We used this heap size for the analysis in section 2 and section 3. For evaluation, we used 2×, 5×, and 10× the minimum heap sizes.

5.3 Synthetic Benchmark Program

We developed a synthetic benchmark program, `hcskiller`. The program is carefully designed so that reference loads

Table 1. Hardware and software configuration

Operating system	Linux Ubuntu 22.04.4 LTS
CPU	Intel(R) Xeon(R) W-2235 CPU @ 3.80 GHz
Memory	32 GB
CPU cores	6
NUMA nodes	1

Table 2. Cache and TLB specifications

Name	Capacity / entries	Notes
L1 cache	32 KB	per core
L2 cache	1 MB	per core
L3 cache	8.3 MB	shared across cores
L1 TLB	64 entries	
L2 TLB	1536 entries	

[1] https://github.com/TobiasWrigstad/pldi2020-artefact

Table 3. Heap sizes. The sizes in the "profiling" column are also used for the analysis in section 2 and section 3.

	heap size (MB)	
benchmark	profiling	minimum
avrora	500	54
fop	600	107
h2	2600	434
jython	1200	157
luindex	120	110
lusearch	1600	104
lusearch-fix	2400	104
pmd	400	116
sunflow	1800	154
xalan	1200	180
h2huge	N/A	1669

to persistently hot objects and transiently hot objects are interleaved. As a result, HCSGC fails to minimize the effective working set. We vary the number of involved objects and examine the largest number of objects for which the effective working set fits within the cache capacity.

The hcskiller maintains a large balanced binary tree with 2^{30} nodes. Each node has four fields, an integer `key`, a `value` field, and references to the `left` and `right` children. The `value` field contains a value object with 62 integer fields. In our experimental setting, each object has a 16-byte header, and fields are aligned to 8-byte boundaries[2]. Therefore, each node occupies 48 bytes and each value object occupies 512 bytes. The node objects are designed to be persistently hot objects, and the value objects are designed to be transiently hot objects.

The main loop of this benchmark program, shown in fig. 4, repeatedly performs a lookup in the tree. The keys of this tree are contiguous integers starting from 0. At each iteration, a search key is selected randomly from the range 0 to N. Thus, N acts as a size factor that determines the number of objects involved in the execution; $2N$ nodes are repeatedly accessed, and references to those nodes and N value objects are repeatedly loaded. This is because the search range corresponds to a prefix of the key space, and the program therefore repeatedly searches within a balanced subtree.

In each iteration, the program tests whether the found node has a value or not and increments a counter if it does. Because we construct the tree such that all nodes have values, this test always succeeds. Although the program loads a reference to the value object during this test, the loaded references are never dereferenced. Thus, the value objects are cold objects. In LSBF, these cold objects are recognized as transiently hot objects.

This main loop allocates memory occasionally to trigger GC. More precisely, it creates A instances of the 512-byte

[2]pointer compression is not available with ZGC

```
for (long i = 0; i < ITERATIONS; i++) {
    int key = rand.nextInt(N);
    var value = tree.lookup(key);
    if (value != null) count++;
    /* allocate occasionally */
    if (i % P == 0)
        for (long j = 0; j < A; j++)
            new Value();
}
```

Figure 4. Pseudo code of main loop of `hcskiller`.

value objects once every P iterations. In our experiments, $P = 2^8$ and $A = 2^4$.

5.4 Results for hcskiller

We first examine whether LSBF reduces the effective working set size for hcskiller. Figure 5 plots the number of TLB misses against the size factor, N. Let N_{cap} be the maximum N such that the number of misses is small. These results clearly show that HCSGC failed to reduce the effective working set while HCSGC+LSBF successfully reduced it: $N_{\text{cap}} = 2^{13}$ for HCSGC and ZGC, and $N_{\text{cap}} = 2^{17}$ for HCSGC+LSBF.

These results match the theoretical expectation. In this experiment, the sizes of persistently hot and transiently objects were 48 and 512 bytes, respectively. Because we designed hcskiller so that $2N$ objects were persistently hot and N objects were transiently hot, the fraction of persistently hot objects among all hot objects was:

$$\frac{48 \times 2 + 512}{48 \times 2} \approx 6.33 \approx 2^{2.66}.$$

This is close to the ratio between the effective working set sizes of HCSGC and HCSGC+LSBF in the experiment.

In this experiment, a transiently hot object was 512 bytes, which was smaller than the page size. Thus, the entire transiently hot object was placed in the same page as persistently hot objects. We designed hcskiller so that $2N$ objects were persistently hot and N objects were transiently hot. Because the sizes of persistently hot and transiently hot objects were 48 and 512 bytes, respectively, transiently hot objects occupied a substantial portion of the pages containing persistently hot objects.

Figure 6 plots the number of LLC misses against N. These results indicated a similar tendency, but the effect was less clear than in fig. 5. This is because transiently hot objects were larger than the cache-line sizes. Consequently, only part of each transiently hot object contributed to amplify accesses to persistently hot objects.

For L1 cache misses, plotted in fig. 7, the differences were less clear. Nevertheless, we observed that $N_{\text{cap}} = 2^6$ for HCSGC and $N_{\text{cap}} = 2^7$ for HCSGC+LSBF. We also observed another knee in the curves at $N = 2^{14}$ for HCSGC and $N = 2^{17}$ for HCSGC+LSBF. Around these points, TLB misses

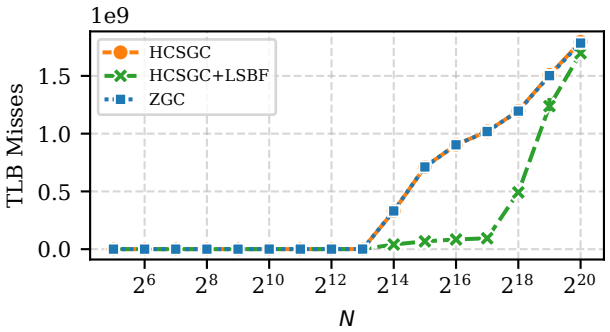

Figure 5. TLB misses for hcskiller

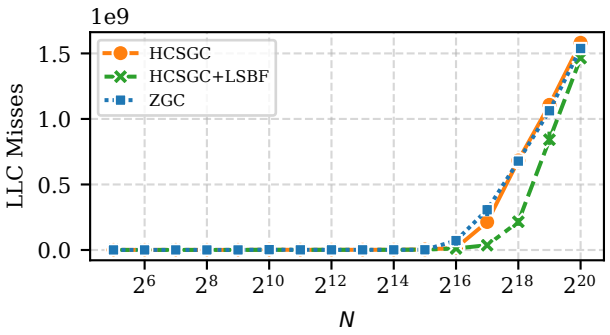

Figure 6. LLC misses for hcskiller

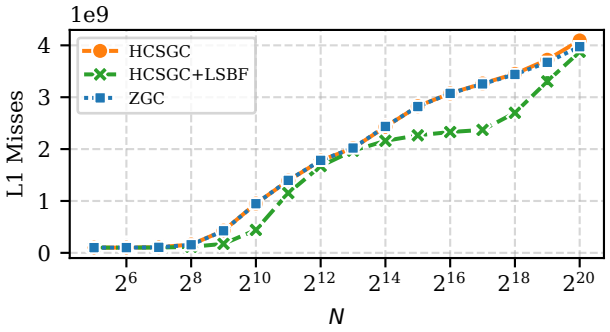

Figure 7. L1 cache misses for hcskiller

started to increase (see fig. 5). As a result, additional memory accesses were caused by page walks during address translations, which caused L1 misses.

Figure 6 plots the execution time of hcskiller against N. HCSGC+LSBF was consistently faster than ZGC and HCSGC, mirroring the reduction in cache misses.

In summary, LSBF successfully reduced the effective working set size and improved performance for a program where HCSGC failed to minimize the working set.

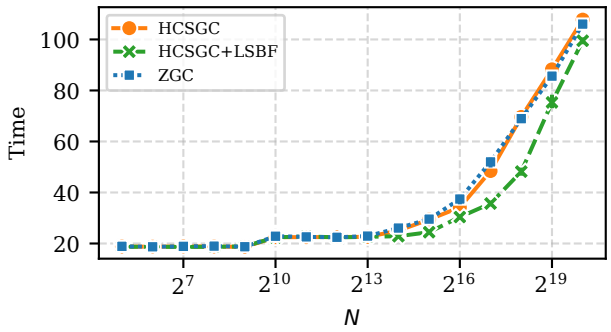

Figure 8. Execution time for hcskiller

5.5 Results for DaCapo Benchmarks

Figures 9 and 11 show the number of TLB, LLC, and L1 cache misses per second, and fig. 12 shows the execution time normalized to ZGC. Each program was executed with 2×, 5×, and 10× the minimum heap size listed in table 3. These results indicate that LSBF did not improve HCSGC for the DaCapo benchmarks, but the performance overhead was limited.

The number of TLB and cache misses were similar across GC algorithms for most programs, and the misses were infrequent. This suggests that the effective working set fit within the cache capacity regardless of object arrangements. For h2huge, we observed that TLB and LLC misses increased for HCSGC+LSBF compared to HCSGC. Nevertheless, the increase was modest, and the impact on the execution time was limited; HCSGC+LSBF was 1.8% slower than HCSGC in the worst case (10× minimum heap size).

6 Related Work

Cache-friendly object placement has long been studied. It is well known that arranging objects in allocation order and preserving this order during GC results in reasonably good locality of reference. White [13] pointed out that garbage collection may disturb the locality of reference of the program by relocating objects that were previously near each other. Petrank and Rawitz theoretically showed that finding an object arrangement that minimizes cache misses is NP-hard [10].

Therefore, most practical approaches rely on heuristics to improve locality. Several studies have proposed techniques to improve locality with acceptable overhead in garbage-collected systems. Some approaches group objects that tend to be accessed together at GC time and place objects in the same group close to each other[8, 11, 14]. Moon [8] integrated a partial depth-first search with breadth-first search in copying GC so that clusters of objects connected by references are likely to be relocated to the same page. Later, Wilson [14] improved Moon's algorithm so that similar heap layouts can be obtained with smaller overhead.

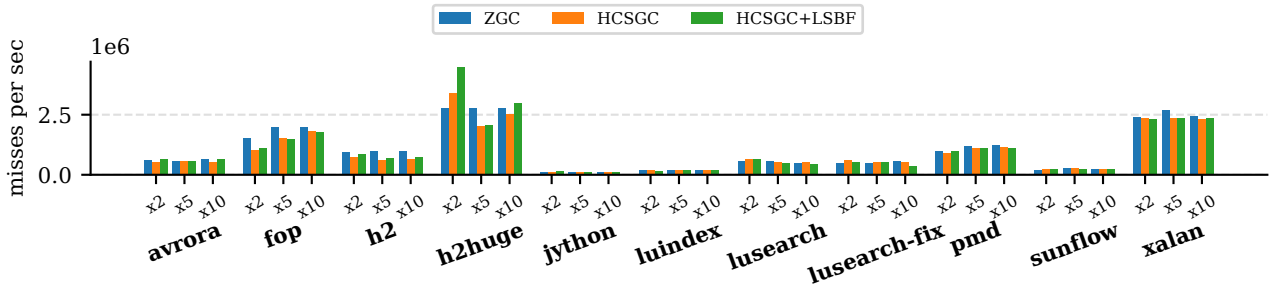

Figure 9. TLB misses per second for DaCapo

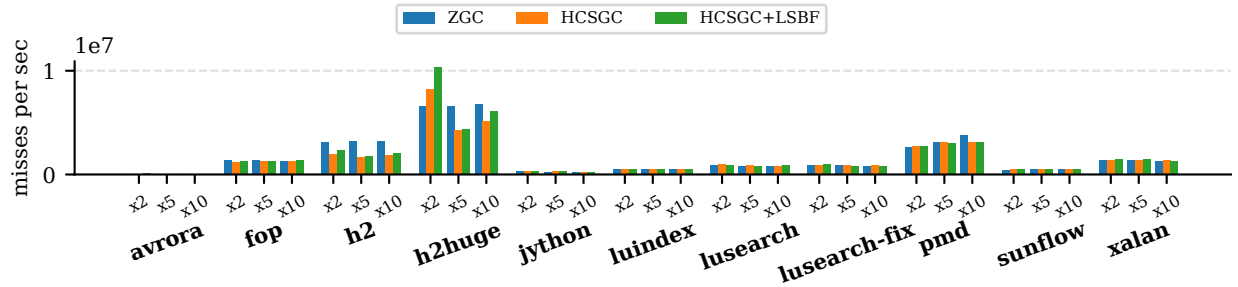

Figure 10. LLC misses per second for DaCapo

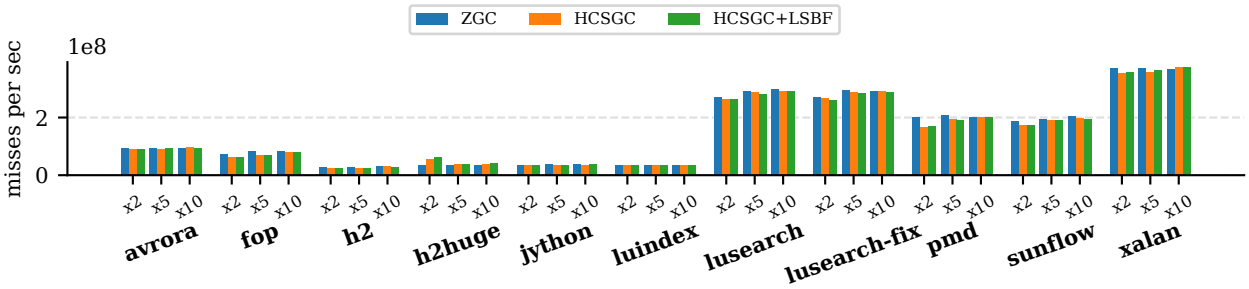

Figure 11. L1 cache misses per second for DaCapo

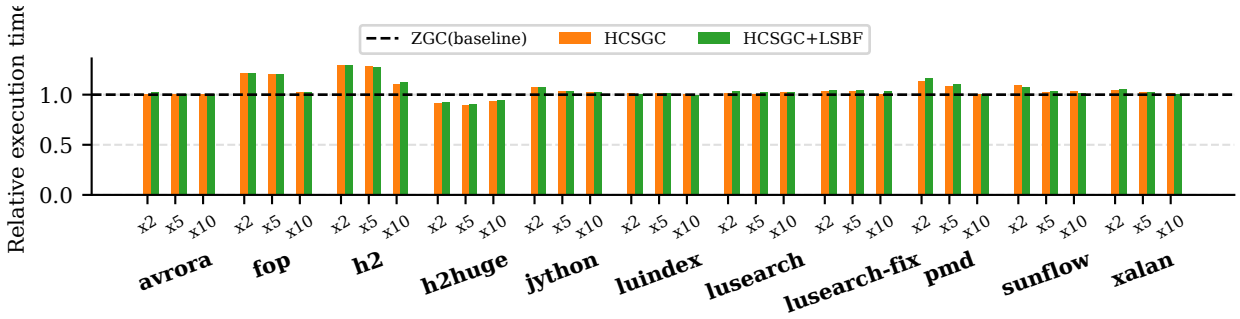

Figure 12. Execution time for DaCapo

Another approach uses profiling to predict the access frequency of objects and groups of objects that are accessed together. Our LSBF falls into this category. Chen et al. [3] use runtime profiling to identify object instances that are accessed together. Huang et al. [6] identify hot fields by piggybacking profiling for JIT compilation and relocate objects connected by references in hot fields together. Our LSBF predict object behavior based on their load sites.

Norlinder et al. [9] propose an improvement to HCSGC [16]. Their approach avoids unnecessary evacuations by preventing the collector from evacuating objects from pages that were not accessed by the mutator by the end of the GC cycle. Our LSBF is orthogonal to this improvement.

7 Conclusion

We have revealed that a substantial fraction of objects whose references are loaded—which HCSGC identifies as hot objects—are not accessed after a few, or even no, accesses immediately after relocated by GC. To segregate such transiently hot objects from persistently hot objects and reduce the working set a while after GC, we have proposed load-site-based filtering (LSBF). This is grounded in experimental observations showing that the ability of load sites to distinguish between persistently and transiently hot objects is as effective as that of allocation sites.

We have implemented LSBF in HCSGC as a profiling-guided optimization. Using LSBF based on the classification of persistently-hot and transiently-hot load sites obtained through offline profiling, we found a benchmark where the working set was smaller and cache/TLB misses were reduced compared to HCSGC. Furthermore, we confirmed that our implementation, which eliminates runtime conditional branches for segregation by consulting load site classifications during JIT compilation, exhibits low overhead on the DaCapo benchmark.

Acknowledgments

This work was partially supported by JSPS KAKENHI Grant Numbers 23K24822 and 26K02887.

References

[1] Henry G. Baker. 1978. List processing in real time on a serial computer. *Commun. ACM* 21, 4 (April 1978), 280–294. https://doi.org/10.1145/359460.359470

[2] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. ACM Press, 169–190.

[3] Wen-ke Chen, Sanjay Bhansali, Trishul Chilimbi, Xiaofeng Gao, and Weihaw Chuang. 2006. Profile-guided proactive garbage collection for locality optimization. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 332–340.

[4] Sylvia Dieckmann and Urs Hölzle. 1999. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmark. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99)*. Springer-Verlag, Berlin, Heidelberg, 92–115.

[5] Joseph A. Fisher and Stefan M. Freudenberger. 1992. Predicting conditional branch directions from previous runs of a program. *SIGPLAN Not.* 27, 9 (Sept. 1992), 85–95. https://doi.org/10.1145/143371.143493

[6] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. 2004. The garbage collection advantage: improving program locality. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 69–80.

[7] Richard E. Jones and Chris Ryder. 2008. A study of Java object demographics. In *Proceedings of the 7th international symposium on Memory management (ISMM 2008)*. ACM, 121–130. https://doi.org/10.1145/1375634.1375652

[8] David A. Moon. 1984. Garbage collection in a large LISP system. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (Austin, Texas, USA) *(LFP '84)*. Association for Computing Machinery, New York, NY, USA, 235–246.

[9] Jonas Norlinder, Albert Mingkun Yang, David Black-Schaffer, and Tobias Wrigstad. 2024. Mutator-Driven Object Placement using Load Barriers. In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. 14–27.

[10] Erez Petrank and Dror Rawitz. 2002. The hardness of cache conscious data placement. *SIGPLAN Not.* (2002), 101–112.

[11] Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Appel, and Jaswinder Pal Singh. 2002. Creating and preserving locality of java applications at allocation and garbage collection times. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 13–25.

[12] David Siegwart and Martin Hirzel. 2006. Improving locality with parallel hierarchical copying GC. In *Proceedings of the 5th international symposium on Memory management*. 52–63.

[13] Jon L White. 1987. Address/memory management for a gigantic LISP environment or, GC considered harmful. *SIGPLAN Lisp Pointers* 1, 3 (1987), 17–25.

[14] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. 1991. Effective "static-graph" reorganization to improve locality in garbage-collected systems. *SIGPLAN Not.* 26, 6 (1991), 177–191.

[15] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News* 23, 1 (1995), 20–24.

[16] Albert Mingkun Yang, Erik Österlund, and Tobias Wrigstad. 2020. Improving program locality in the GC using hotness. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 301–313. https://doi.org/10.1145/3385412.3385977

[17] Albert Mingkun Yang and Tobias Wrigstad. 2022. Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK. *ACM Trans. Program. Lang. Syst.* 44, 4 (2022), 22:1–22:34.