Improving Program Locality in the GC using Hotness

Albert Mingkun Yang
Uppsala University
Sweden
albert.yang@it.uu.se

Erik Österlund
Oracle
Sweden
erik.osterlund@oracle.com

Tobias Wrigstad
Uppsala University
Sweden
tobias.wrigstad@it.uu.se

Abstract

The hierarchical memory system with increasingly small and increasingly fast memory closer to the CPU has for long been at the heart of hiding, or mitigating the performance gap between memories and processors. To utilise this hardware, programs must be written to exhibit good object locality. In languages like C/C++, programmers can carefully plan how objects should be laid out (albeit time consuming and error-prone); for managed languages, especially ones with moving garbage collectors, a manually created optimal layout may be destroyed in the process of object relocation. For managed languages that present an abstract view of memory, the solution lies in making the garbage collector aware of object locality, and strive to achieve and maintain good locality, even in the face of multi-phased programs that exhibit different behaviour across different phases.

This paper presents a GC design that dynamically reorganises objects in the order mutators access them, and additionally strives to separate frequently and infrequently used objects in memory. This improves locality and the efficiency of hardware prefetching. Identifying frequently used objects is done at run-time, with small overhead. HCSGC also offers tunability, *e.g.,* for shifting relocation work towards mutators, or for more or less aggressive object relocation.

The ideas are evaluated in the context of the ZGC collector on OpenJDK and yields performance improvements of 5% (tradebeans), 9% (h2) and an impressive 25–45% (JGraphT), all with 95% confidence. For SPECjbb, results are inconclusive due to a fluctuating baseline.

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

Keywords: data locality, GC, cache optimisation, prefetching

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '20, June 15–20, 2020, London, UK
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3385977

ACM Reference Format:
Albert Mingkun Yang, Erik Österlund, and Tobias Wrigstad. 2020. Improving Program Locality in the GC using Hotness. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3385412.3385977

1 Introduction

Multi-level caches are designed to hide the memory latency caused by the increasing gap between CPU processing speed and memory access times. The designs of these memory hierarchies assume good object locality, which does not fall out for free, especially not in high-level languages where "everything is an object" and data structures consequently distributed over a large memory range. In theory, the optimal object layout (in the sense of placing objects to enjoy maximal cache benefits) could be achieved via careful engineering. However, in practice this rarely happens due to two challenges, which we now discuss in turn.

The first challenge is that it is time consuming (and/or extremely difficult) to calculate a program's optimal layout [15, 20]. To some extent, this could be mitigated by the (and in managed languages widespread) use of bump-pointer allocation schemes, which maintains a pointer pointing at the beginning of free space, and allocation means "bumping" the pointer by requested size. For applications that tend to access objects in the order they were created, the bump-pointer allocator automatically generates a suitable layout which gives good performance. Assuming most programs exhibit this behaviour, many compacting garbage collectors choose sliding order, where all objects that are deemed live are slid to one end, squeezing out garbage, so that their allocated order is preserved and no "hole" of unused space exists. Abuaiadh et al. [1] confirms this by showing that destroying this order could drastically reduce application throughput.

Franco et al. [10] show that it only takes a small fraction of interleaving allocations that do not follow the access order to generate a noticeable slowdown. And even if such allocations do not exist, the "access order mirrors creation order" assumption may still not hold consistently throughout an application. An application may go through multiple phases with different access patterns on the same set of objects, different from the allocation order, but still stable within each phase. This brings us to *the second challenge*:

maintaining optimal layout throughout an application's different phases of execution. Languages that employ a moving garbage collector could be well-suited to tackle this challenge as it already involves changing objects' locations without changing how they are connected.

There are two main families of moving collectors: copying collectors traverse the object graph and copy objects encountered along the way to the destination space; mark-compact collectors identify live objects and compact the heap by relocating live objects as well as update pointers to the objects that are moved. Prior work has built upon moving collectors to intelligently rearrange objects layout to improve locality [2, 8, 9, 13, 22], where new objects layout is created by GC threads based on collected access info. In contrast, our approach allows mutators to reorganise objects as they access them, which not only improves objects locality, but also creates a layout that is prefetching friendly, and is inherently responsive to phase changes in applications.

The paper makes the following contributions:

Refining live objects into hot and cold: We introduce a notion of hot and cold objects, where hot objects are those that were touched by the mutator since the previous GC cycle (§3.1.2) and show how this information can be obtained cheaply. We then use this categorisation during object relocation, treating hot and cold objects differently, which results into hot–cold segregation, and increasing performance (§3.3).

Reorganise objects in mutator's accessing order: With the help of concurrent relocation, mutators can participate in moving of objects, directly affecting the order in which objects are laid out, which is prefetching friendly. This is achieved without using any bookkeeping data structures to record the accessing order (§3.2).

Implementation in OpenJDK: We describe the implementation (all of §3) as an extension to Oracle's recently announced ZGC collector, along with the necessary modifications to the existing ZGC implementation.

Evaluation of our design: We validate our design and demonstrate the effect of tuning the system using a family of tuning knobs on a range of benchmarks from JGraphT, DaCapo and SPECjbb 2015 (§4) with > 45% performance increase at best.

In addition, §5 discusses related work, and §6 concludes.

2 A Primer on ZGC

Before explaining the design of HCSGC in §3, we start with a brief overview of the ZGC collector that HCSGC is based on [16–18]. This additionally serves the purpose of showing the rationale and motivation for several of our choices for the design and implementation of HCSGC.

ZGC is a non-generational, mostly concurrent, parallel, mark-compact, region-based garbage collector. It is included OpenJDK releases since JDK11 (Dec 2018) and enabled by the special flags `-XX:+UnlockExperimentalVMOptions` and `-XX:+UseZGC`. ZGC uses metadata bits in pointers in combination with load barriers to ensure a mutator never sees a stale pointer despite concurrent compaction.

By *non-generational*, we mean that there is no segregation of objects by age (or any other classification for that matter), and a GC cycle involves marking all live objects in the whole heap. By *mostly concurrent*, we mean that there are stop-the-world (STW) pauses (three in a GC cycle), but that these are very brief, and that the majority of garbage collection is done concurrently with the application's threads (aka the mutators). By *parallel*, we mean that multiple threads are used to perform GC work: in STW pauses; in *marking* (tracing the entire heap to find all live objects); and in *compacting* (relocation of live objects to defragment memory and free garbage objects). By *region-based* we mean that all allocations are served from pages of preordained size classes.

Coloured pointers are an abstraction that is used consistently throughout ZGC in relation to its load barrier-based design: pointers have colours (captured by meta data stored in the higher-order bits of pointer addresses), and at every moment in time, all threads agree on what colour is the "good colour" (these agreements all take place during a brief stop-the-world pause, more on this in §2.2). The `new` operator always returns a pointer with good colour. Loading a pointer from heap to stack always involves a check—a *load barrier*—and a good-coloured pointer will always hit the fast path which incurs no additional work[1]. Otherwise, it will hit the slow path and the slot where this pointer resides will be updated with a good coloured alias to avoid hitting the slow path subsequently (denoted self-healing in ZGC parlance). For example, a *stale pointer* (a pointer to an object that has been moved concurrently during compaction, meaning the address may point to an outdated copy of the object, or another object, or even nothing) is guaranteed to not have the good colour. Loads of such a pointer by a mutator thread will be trapped in the load barrier, and the pointer will be replaced by a correct pointer to the new location and this updated pointer will be returned to the mutator.

2.1 Allocation and Deallocation in ZGC

The ZGC heap is divided into pages of three size classes: small, medium and large. Objects are allocated on one of those pages, depending on their size, as shown in Table 1. For small and medium pages, bump-pointer allocation scheme is used, and when the current page can not satisfy the requested size, a new page is allocated. An object larger than 4 megabytes will have its own page; in other words, there is only one single object on each large page.

Memory reclamation happens on the granularity of a page and as part of relocation. Depending on how sparsely populated a page is, it may get *selected for evacuation.* During the relocation phase, all live objects on pages selected for

[1] Technically, different load barrier are used by mutators and GC.

Table 1. ZGC Page Size Classes.

Page Size Class	Page Size	Object Size
Small	2 Mb	[0, 256] Kb
Medium	32 Mb	(256 Kb, 4 Mb]
Large	$N \times 2\,(> 4)$ Mb	> 4 Mb

evacuation will be moved elsewhere, after which the page (including all remaining garbage objects) is recycled. This means that a garbage object may not be freed for long (or ever) if it resides on a page dominated by long-lived objects.

2.2 The Phases of a ZGC Cycle

A ZGC cycle has three stop-the-world (STW) pauses (all mutators are stopped), and three concurrent phases, as shown in Fig. 1. A new ZGC cycle will not start until the previous one is finished (no overlapping ZGC cycles).

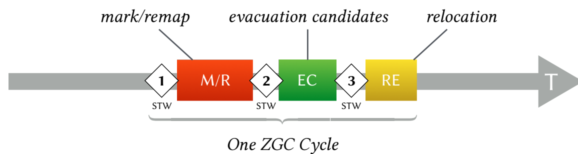

Figure 1. A ZGC cycle has three stop-the-world pauses (drawn as diamonds) and three phases (drawn as rectangles), shown in the order they occur. The grey arrow denotes time.

STW1. A ZGC cycle starts with a STW pause (which we denote STW1), in which the good colour is selected, all roots are pushed into mark stacks (for parallel processing[2]), and tinted with the good colour. The good colour will be either M0 or M1; if M0 is selected, M1 will be the good colour in next ZGC cycle, and the two alternate.

M/R. After the STW1 pause, all threads agree on the good colour and ZGC transitions into the Mark/Remap phase (denoted M/R), and performs classical object graph traversal to identify and mark all live objects. If a stale pointer is found during this process, it is updated with the current address of the object it refers to, and tinted with the good colour. This guarantees that there are no stale pointers in heap after the M/R phase has completed. In addition to marking all live objects, per-page liveness information (the total number and the total size of live objects on each memory page) is recorded as well. The liveness information is used to select pages for evacuation, as outlined at the end of §2.1.

STW2. The M/R phase ends with a second stop-the-world pause (STW2), which ensures that all mark stacks are empty, and that marking is completed.

[2]Both mutators and GC threads have their own thread-local mark stack to reduce synchronisation cost, and GC threads perform work-stealing among themselves to balance the marking workload. Additionally, mutators will flush their thread-local mark stacks regularly for idle GC threads to pick up.

EC. Next, using the previously collected per-page liveness information, all small pages that are allocated prior to STW1 and whose ratio of live bytes (the sum of the sizes of all live objects) is below a certain threshold (75% by default) are identified and sorted based on live bytes in ascending order. Then, the first N pages in the sorted list will be flagged as *evacuation candidates* (denoted EC). The value of N is derived by maximising the number of selected pages subject to the following constraint:

$$\frac{\text{live bytes on } N \text{ pages}}{\text{page size}} \leq 75\%$$

Medium pages are treated the same, but not large pages. Since each large page only contains a single object, which is either live or dead, we can decide whether that large page should be kept or reclaimed right away (in the current phase, rather than waiting for RE), without adding them to EC. Therefore, only small and medium pages are ever in EC.

STW3. EC selection ends with a third and final stop-the-world pause (STW3). In STW3, the good colour changes to R, which effectively invalidates all pointers causing mutators to hit the slow path on loading them. The slow path of the load barrier will inspect if the to-be-loaded object resides on a page in EC; if so it will be relocated to another page, then the slot this pointer resides will be self-healed with the good colour. By the end of STW3, all roots pointing into EC are relocated and have the good colour.

RE. Now ZGC transitions into the relocation phase (RE). In RE, all live objects in EC will be relocated to new pages. This causes incoming pointers to relocated objects to become stale. A per-page forwarding table is used to record a map from old addresses to new and are consulted by the mutators when accesses to stale pointers are trapped in the load barriers, or during the marking of the next M/R phase (if no mutator accessed them). Relocation is concurrent with mutators: if a mutator loads a stale pointer pointing into EC, it will hit the slow path of the load barrier, and compete with GC threads for relocation. The linearization point is a CAS operation when inserting the corresponding entry into the forwarding table. Whoever succeeds in the CAS will use its local value (which is the same as the one in the forwarding table), while others will discard their local value, and read the new value from the forwarding table.

Note: the good colour is changed twice per GC cycle, and the window for each choice of good colour is shown in Fig. 2.

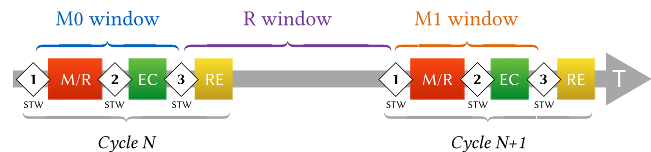

Figure 2. The window for each colour (M0, R and M1).

In addition to the above, ZGC has another phase between STW2 and EC that handles reference processing for Java's soft, weak and phantom references. Because that aspect is independent of this work, we omit its description entirely.

3 Hot-Cold Objects Segregation GC

We now describe the design of HCSGC and its implementation on top of ZGC. The objective of HCSGC is to transparently improve an application's effective use of cache memory and hardware prefetching. To this end, HCSGC attempts to place objects in such a way that they are only brought into cache when they are about to be used *by the mutator*; moreover, HCSGC attempts to lay them out according to the mutator access order. Thus, if an object is brought into cache due to an access by a particular mutator thread, we expect the subsequent accesses by the same thread to hit the adjacent object, or at least fall within the same or adjacent cache line. For now, we restrict ourselves to only handling objects whose size is smaller or close to the cache line size, which reside on small pages according to Table 1. We will discussion this choice further in §3.4.

We leverage the fact that relocation is concurrent with mutator accesses, as shown in §2.2. This means that mutators and GC threads are competing (or alternatively, collaborating) to perform relocation. A mutator that attempts to follow a pointer to an object on a page in EC that has not yet been relocated will perform relocation of that object, which copies it into a new page. If a mutator follows a path in the object graph to such to-be-relocated objects, all these objects will moved and consequently laid out in memory according to the mutators' access order, achieving both close placement of related objects, and in an optimal order if the access path is repeated. Following this realisation, we provide three ways of exposing mutators to *more* relocation:

- *Enlarging EC* by flagging more small pages as evacuation candidates. This allows more objects' locations to improve.
- *Lazy relocation*; deferring the RE phase until the next M/R phase. This gives mutators an advantage when competing with GC threads to relocate objects on pages in EC.
- *Dedicated cold pages* where we move less frequently accessed objects during relocation, to avoid them being accidentally brought into memory by close proximity to more frequently used objects.

3.1 Revised EC selection

As mentioned in § 2.2, the criteria used for EC selection in ZGC is the live ratio, because the sole job of ZGC is to reclaim unused memory. From that standpoint, there is not much to gain from relocating a page if it is already densely populated. With the added goals of HCSGC, a revision of the EC selection strategy is warranted. Thus, we go through two possible strategies for EC selection with additional goals in mind, each with its own merits.

3.1.1 Including All Small Pages in EC. A crude-but-simple way to enlarge EC is to include all small pages in EC. This has an immediate upside: it gets rid of the logic for sorting and finding the first N pages. The price of such crude behaviour is the (potential) added cost of mutators performing additional relocations (copying) as part of the first access to every object. For objects that are not accessed by mutators, GC threads will perform the relocation. If a machine is not over-provisioned, *i.e.,* it has idle cores, such extra work may not affect the execution time, since GC threads run concurrently with mutators. In § 4, we will show in a benchmark that such extra work stays hidden in an unloaded system, but materialises when computing resources become constrained. We expose this behaviour as an on/off tuning knob called RelocateAllSmallPages.

This strategy fails if the most frequently accessed objects are already concentrated on a few pages and the access pattern is stable—forcing extra copying work on mutators for no gain. To avoid such cases, we introduce additional logic in HCSGC that enables a classification of live objects into *hot and cold*. Using this extra per-object property, we show that it is possible to perform more intelligent EC selection.

3.1.2 Hot and Cold Objects. We define a hot object as one that has been accessed by a mutator since the last GC cycle. All live objects that are not hot are considered cold. Hotness as thus defined can be easily captured by mutators and GC threads through a small extension of the load barrier. Similar ideas were used in [6, 7].

- *Mutators* flag an object as hot on the slow path of a load barrier (because if accessed, it is hot by definition); *and*
- *GC threads* on finding pointers with R colour while traversing the object graph in the M/R phase, will flag the corresponding objects as hot. Recall Fig. 2, pointers with R colour means that they were accessed by mutators since STW3 of the previous GC cycle.

Per-object hotness metadata is recorded in a bitmap called *hotmap*, adapted from the *livemap* used in ZGC for tracking liveness info. Similar to livemap, hotmap is reset at the beginning of each M/R phase; this renders all objects cold effectively. Hotness is recorded by mutators and GC threads during the M/R phase. A per-page total size of hot objects (hot bytes) is also calculated. We expose the collection of hotness information as an on/off tuning knob called Hotness.

After the M/R phase and before the next phase starts, hot bytes are used for EC selection. Per-object hotness information is used for proactive hot–cold objects segregation (§3.3).

With the help of this hotness information, we can make a more intelligent EC selection, as covered in the next section.

3.1.3 EC Selection via Weighted Live Bytes. We introduce the concept of *weighted live bytes* as an indicator more

fine-grained than plain live bytes when performing EC selection. Per-page Weighted Live Bytes is defined as:

$$WLB = \begin{cases} \textit{cold bytes} & \textit{if hot bytes} = 0 \\ \textit{hot bytes} + \textit{cold bytes} \times (1 - \textit{cold conf.}) & \text{otherwise} \end{cases}$$

The *cold confidence* is a value in the range [0–1.0] that indicates how confident we are that a cold object will stay cold in the near future—a simple model of temporal locality. If zero, *weighted live bytes* simply "degrades to" ZGC's original *live bytes*. In contrast, a cold confidence of 1.0 means that *weighted live bytes* only considers *hot bytes*. This value is exposed as tuning knob, whose default value is 0 to match the original behaviour of ZGC. If a page contains only cold objects, we simply use *cold bytes* (which is equal to *live bytes*, the original ZGC behaviour), since there are no hot objects mixed with cold ones. Using this tuning knob, a relatively small number of hot objects buried in a page fairly populated by many live but cold objects could still go into EC, causing segregation of hot and cold objects. The lower the weight of cold bytes, the higher the probability that a page with many cold (but live) objects will be selected for evacuation.

With this new criteria, we are able to influence how large a portion of the heap that will get selected as EC, and segregate hot and cold objects living on the same page, without paying the price of relocating pages already mainly occupied by hot objects. However, there is a caveat here: if a page is well populated with hot objects, we cannot use the new criteria to have it added to EC, even if the access order of those hot objects are changed. Therefore, mutators will not enjoy the same layout improvement as in the relocating-all-pages case.

Now that there are potentially more pages going into EC, and relocation serves an additional purpose to deallocating garbage, we move on to discus techniques for exposing mutators to more of the relocation work, in §§3.2 to 3.3.

3.2 Deferring Relocation Performed by GC Threads

Even if we flag more pages (and in a more informed manner using hotness) as evacuation candidates, mutators and GC threads are still competing to relocate objects on EC pages. This means that GC threads may sometimes relocate an object before mutators try to access it, which means that we miss an opportunity to improve the object's placement.

The original placement of the RE phase right after the EC phase in ZGC is motivated by the desire to put the cost of performing relocation (copying) on GC threads. This saves a mutator the cost of relocation of objects it accesses—it only needs to read their updated addresses from the forwarding table. With the addition of hotness information, relocation provides an opportunity to rearrange object layout, not just reclaiming unused memory. This motivates exposing mutators to more relocation overhead assuming that improved locality on subsequent accesses turns it into a net win.

Therefore, instead of having GC threads race with mutators to relocate objects in EC, we defer the RE phase (and

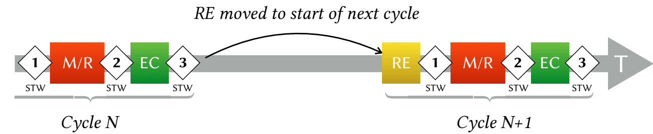

Figure 3. Deferring relocation to the next GC cycle exposes mutators to more relocation work. In ZGC, a GC cycle ends with RE; in HCSGC a GC cycle starts with RE, allowing mutators to relocate undisturbed between two GC cycles.

hence the work delegated to GC threads) to the next GC cycle, as show in Fig. 3. This way, objects may be relocated according to mutators' access pattern between *two* GC cycles. The cost is that floating garbage will be retained longer, until the start of the *next* GC cycle. Thus, with this change, each GC cycle (except the first, of course) *starts* with releasing memory, instead of *ending* with releasing memory.

Note that this feature is completely orthogonal to hotness information: hotness information is (optionally) used to select EC, but deferring relocation done by GC threads can be done invariant of hotness. We expose this feature as an on/off tuning knob called LazyRelocate.

3.3 Speculative Hot–Cold Segregation

In addition to shifting more relocation work to mutators, it is possible to pro-actively segregate hot and cold objects in memory to increase the chances of cache-friendly object placement. Originally in ZGC, each GC thread maintains a thread-local page to which objects in EC will be relocated, motivated by avoiding contention on bump pointer allocation. With the introduction of per-object hotness information, we can leverage this property during relocation to move hot and cold objects to different destinations in memory.

We use the original thread-local page for hot objects, and introduce another page that we denote the *cold page* (when enabled, we call the other thread-local page the *hot page*) and use it as the target destination for relocating cold objects. Consequently, each GC thread in HCSGC has two thread-local pages, for hot and cold objects, respectively.

Note that this only affects GC threads, so is completely orthogonal to relocation performed by mutators. However, exposing mutators to more relocation work using LazyRelocate will likely reduce the pressure on the GC threads' hot pages. We expose this feature as an on/off tuning knob called ColdPage, which requires that Hotness must be on.

3.4 Operating Only on Small Pages

For EC selection and hotness collection, only small pages are supported. Since hotness is recorded on the granularity of objects, a large object with only a few fields accessed will be considered hot according to our scheme. However, based on such inaccurate over-approximation, we may decide to place those large objects together in order to improve cache

utilisation, but this actually reduces the effective size of cache, since large parts of those objects are not used. Smaller objects are more resilient to this problem. Thus, HCSGC only deals with small pages and leaves medium and large page intact. In fact, even small pages may be too coarse-grained; recall in Table 1, small pages can hold objects up to $256K$ bytes, which is still much larger than common cache line size (64 bytes). Ideally, a new page size class in which the max object size is on the magnitude of cacheline size could have been a more suitable candidate. We leave this for future work.

4 Evaluation

All benchmarking except SPECjbb2015 is run on an Intel® Core™ i7-4600U CPU @ 2.10GHz with 2 cores (2 hyper-threads/core), 12GB RAM, 32KB L1, 256KB L2, 4MB L3, running Debian 11 (bullseye) with Linux kernel version 5.2.17. SPECjbb2015, which requires a large heap, is run on an AMD Opteron Processor 6276 @ 3091.357 MHz with 32 cores (2 hyper-threads/core), 128GB RAM, 16KB L1d, 64KB L1i, 2MB L2, 6MB L3, running Debian 10 (buster) with Linux kernel version 4.19.67. For runs on the server, we use *numactl* to make sure only physical cores are used to avoid cache thrashing. The OpenJDK commit we build HCSGC on is authored on 2019-10-14, and the C/C++ compiler used is GCC 9.2.1.

4.1 Configurations

We run our benchmarks for a large number of configurations created by enumerating all the tuning knobs introduced in §3 and forming "all combinations" of on/off knobs and ColdConfidence values of 0.0, 0.5 and 1.0. All configurations we used are shown in Table 2. For convenience, we recap our tuning knobs in one place:

Hotness When on, objects' hotness is recorded and stored in the hotmap. This has the overhead of updating the hotmap which in its current implementation involves a CAS operation.

ColdPage When on, GC threads use a separate page as destination for cold objects during relocation. This option requires that Hotness is enabled.

ColdConfidence A number 0–1.0 that influences the weight of cold objects when calculating live bytes. When 0, cold objects are treated the same as as live (*e.g.,* the default ZGC behaviour); when 1.0, cold objects are treated as garbage (*e.g.,* only count hot bytes as live). This option requires that Hotness is enabled.

RelocateAllSmallPages When on, all small pages will be placed in EC during relocation. In theory, especially in combination with LazyRelocate, this allows mutators to relocate objects in access order. This option does not rely on (or take advantage of) hotness information.

LazyRelocate When on, defers the relocation done by GC threads to the next GC cycle (Fig. 3).

Notably, Config 0 is our baseline—the original ZGC behaviour which we are trying to improve. Config 1 is the using our modified ZGC, but all tuning knobs are disabled, meaning the behaviour is *equivalent* to Config 1, but sources are not identical. We expect no significant difference between Configurations 0 and 1. In Configurations 2–3, hotness is not recorded, so neither ColdPage nor ColdConfidence can be used, but we may relocate all pages or do lazy relocation or both. For the rest of the configurations, hotness is recorded. In Configurations 5–7, we leverage hotness to change EC selection (a larger value of ColdConfidence means a larger EC set). For Config 5, ColdConfidence is set to 0 which gives us the original ZGC EC selection, as discussed in §3.1.3, thus this configuration shows the overhead of recording hotness. In Configurations 8–10, 11–13 and 14–16, we turn on LazyRelocate or ColdPage or both. Finally, Config 17–18 shows if or how LazyRelocate works in the presence of RelocateAllSmallPages and with ColdPage.

4.2 Data Collection and Visualisation

In order to study the impact of HCSGC, we collect data on three aspects: execution time, cache statistics, and GC statistics. Next, we go through each of them and describe how we collect the data and visualise it.

How Figures are Structured. Figures 4, 5, 7, 8, 9, 10, 11, and 12 all follow the same layout, each with no less than *seven* plots with the following layout (elaborated below):

Execution Time	*Cache Statistics*	*GC statistics*
Wall-clock time in seconds	Average total loads, L1 and LLC misses normalised against ZGC (using `perf`)	Average GC cycles per run
–"–, w/ mean and confidence interval		Average (†) pages relocated per run
–"–, normalised against ZGC		Heap usage in % using ZGC

† **Note**: average of *median* small pages relocated per run.[3] ZGC is Config 0 in all plots. All plots have Config ID on x axis, except for heap usage whose x axis shows execution time (seconds). Negative values in the plots normalised against ZGC means *speedup* or *fewer* loads or misses for HCSGC.

Execution Time. For each configuration in Table 2, we measure the wall-clock execution time of N runs (elaborated below), producing a sample of size N, which is visualised using box plots [19], as exemplified by the top plot in Fig. 4. The notations used: the band inside the box is the second quartile (the median), and the lower and higher end of the box are the first and third quartile, respectively. The first quartile (Q_1) splits off the lowest 25% data from the highest 75%, while the third quartile (Q_3) splits off the highest 25% data from the rest. The inter-quartile range (IQR) is defined

[3] For each run, we get the number of relocated small-pages per GC cycle. Then, we calculate the median from this list. Finally, we calculate the average of the median over multiple runs.

Table 2. Configurations used in benchmarking. Config 0 is our baseline: unmodified ZGC. 0/1 means a flag was turned off/on. As for ColdConfidence, we pick [0, 0.5, 1.0]. Config 5 turns on hotness tracking but does not use it.

Tuning Knobs	ZGC	HCSGC Configurations																	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Hotness	n/a	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
ColdPage	n/a	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
ColdConfidence	n/a	0	0	0	0	0	0.5	1.0	0	0.5	1.0	0	0.5	1.0	0	0.5	1.0	0	0
RelocateAllSmallPages	n/a	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1
LazyRelocate	n/a	0	1	0	1	0	0	0	1	1	1	0	0	0	1	1	1	0	1

as the difference between the third quartile and the first quartile, $IQR = Q_3 - Q_1$. Outliers are defined as data falling outside this range, $[Q_1 - 1.5 \times IQR, Q_3 + 1.5 \times IQR]$, which are further classified into mild outliers, and extreme outliers. Outliers falling outside the range $[Q_1 - 3 \times IQR, Q_3 + 3 \times IQR]$ are called extreme outliers, otherwise, they are called mild outliers. Mild and extreme outliers are denoted using '+' and '∘', respectively. The whiskers indicate the furthest points from the median that are not outliers.

In order to find the mean estimate and its associated confidence interval, we use bootstrapping: taking the original sample, resampling from it to form a new sample, called bootstrap sample, that is also of size N. The bootstrap sample is taken from the original by using sampling with replacement. Each data point in the original sample has the equal probability to be included into the bootstrap sample. This process is repeated 10,000 times, and for each of these bootstrap samples we calculate its mean, called bootstrap mean. Finally, we calculate the mean for all bootstrap means, which is the mean estimate for wall-clock execution time, and its 95% confidence interval is marked by 2.5 and 97.5 percentile, as exemplified by the middle plot in Fig. 4. *If the confidence interval of two configurations do not overlap, then we can conclude, with confidence level* 95%*, that there is a significant difference between the two configurations.* In order to illustrate the relative difference against the baseline, we normalise the mean estimate and show the difference between each configuration and Config 0 right below in the same figure. A negative number means reduced execution time.

For synthetic and JGraphT benchmarks, we launch the VM 31 times and each invocation runs the entire program end-to-end once. The first execution is discarded, because the first VM invocation in a series of measurements may change system states that alters the subsequent measurements persistently, *e.g.,* dynamically loaded libraries stays in physical memory. Therefore, the sample size is 30. For the DaCapo benchmarks, we use 5 VM invocations and 25 benchmark iterations per VM invocation. The first 15 iterations are for warm-up, and we only preserve the final 10 iterations, which is used to calculate the average. Therefore, the final sample size is 5. This methodology is taken from [11]. For SPECjbb benchmark, we run it end-to-end 5 times, and use its built-in reporting metrics, throughput and latency scores.

Cache Statistics. We use `perf` to measure cache metrics: `L1-dcache-loads`, `L1-dcache-load-misses`, and `LLC-load-misses`, reflecting how locality is effected by HCSGC. These metrics are also impacted by extra GC activities introduced by HCSGC, like RelocateAllSmallPages. We prefix the commands of launching JVM, and when JVM exits, `perf` reports the number of events we specified above. The statistics is for the whole process so we cannot distinguish mutators from GC threads (or other VM internal threads). For DaCapo benchmarks that use warm-up, we show cache statistics for the complete run, since `perf` collects events for the whole VM invocation. Thus, take these results with a grain of salt.

GC Statistics. In addition to measuring the average number of GC cycles per run, we extend ZGC's builtin logging support to print the number of small pages in EC per cycle, and calculate the average of the median for all runs to see how HCSGC tuning knobs increase relocation. (This metric is not available in Config 0, but should be identical to Config 1.) The data was obtained from the runs in which execution time was measured (with negligible overhead).Finally, we show the heap usage in % over time for one run using unmodified ZGC. This shows the allocation rate of the application, and its tendency to hold on to objects created.

4.3 Our Expectation

Hot and cold objects on a fairly populated page will remain together if allocated together, since ZGC will never add it to EC. We expect that RelocateAllSmallPages and a large value of ColdConfidence could excavate hot objects buried in such cases, and mutators could enjoy better locality, reflected in reduced execution time and cache misses.

The difference between RelocateAllSmallPages and ColdConfidence may be small on execution time (if the machine is not fully loaded), but the difference in EC size (the number of pages relocated) per GC cycle could be large, which will materialise if the machine is saturated.

Additionally, with this extra complexity (larger EC, lazy relocation, cold page) introduced by HCSGC, each GC cycle

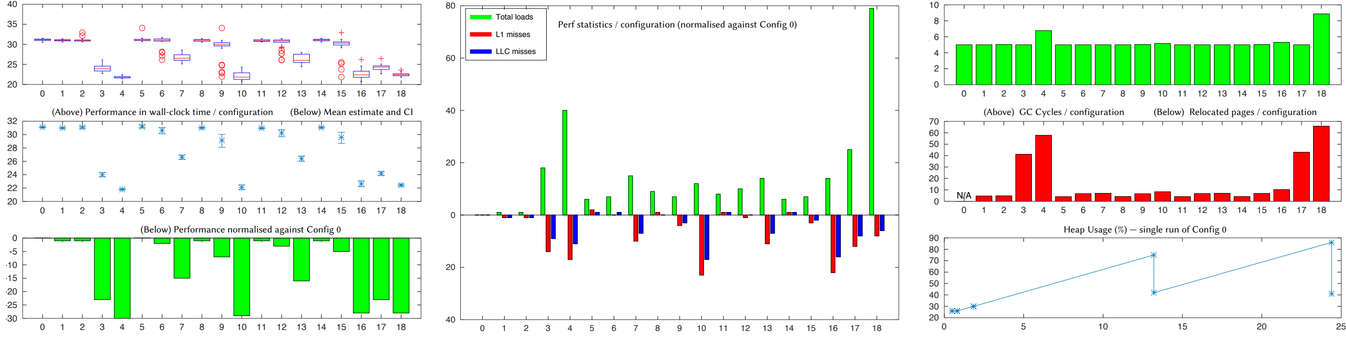

Figure 4. Synthetic benchmark following the layout described in §4.2. For the bottom-left plot, recall that negative y values means speedups; *e.g.,* Config 4 is 30% faster than Config 0. Note the absence of slowdowns.

will take longer time, which paints a pessimistic view of the heap usage for ZGC. Therefore, more GC cycles may be triggered, which can be desirable, as objects' placement can only change during GC relocation. Of course, this assumes the machine is not overloaded so that extra GC activities will not compete with mutators for CPU.

4.4 Sanity Check: Synthetic Micro-Benchmark

As a form of sanity check, we construct a synthetic Java benchmark that exhibits a stable but unpredictable access pattern to illustrate that our approach is able to capture such patterns, and reorganise objects to improve cache utilisation.

In the benchmark, we create an array of 2×10^6 elements, each pointing to a 32-byte object (including VM metadata). Here is the main body of the benchmark in pseudo-Java:

```
1  for (int i = 0; i < 200; ++i) {
2    rand = new Random(0); // use same seed each loop
3    for (int j = 0; j < 800 * 1000; ++j) {
4      index = rand.nextInt(...) // preferably another cache line
5      f(index); // access element in index
6      ++ops;
7      if (ops % 10 == 0) { /* allocate garbage to trigger GC */ }
8    }
9  }
```

The inner loop accesses elements residing in a different cache line, guided by a random generator. The outer loop is used to simulate recurring mutator accesses. Because the random generator is reset using the same seed for each outer loop, each inner loop will perform identical access sequences.

The remainder of this section explores the impact of HCSGC under its many configurations on this benchmark on performance, loads, cache misses and hotness.

Impact on Execution Time. The execution time under each configuration is shown in Fig. 4. As this artificially contrived benchmark has clear hot–cold segregation and a recurring stable access pattern, we can observe the accumulative effect of various tuning knobs, corresponding to the different tiers of performance above.

The largest improvement is seen in Configurations 4, 10, 16, and 18, which have both large EC (due to relocating all pages or 100% cold confidence) and lazy relocation enabled. The second largest improvement comes from Configurations 3 and 17, with large EC due to relocating all pages. Next, Configurations 7 and 13 show some improvement as well, with large EC is due to 100% cold confidence. Finally, Configurations 2, 5, 8, 11, and 14 show no improvement at all. The reason is that objects pointed by each array slot never die—hot objects are always surrounded by live but cold objects, and those pages will not be added to EC as they are fairly populated, so objects on those pages will never be relocated.

Impact on Loads and Cache Misses. Fig. 4 shows cache metrics: Configurations 3, 4, 7, 10, 13, 16, 17, and 18 exhibit large reduction in L1/LLC misses. Note that for Configurations 3–18, there are large increase of total loads, but they are mostly served by cache L1/LLC (small L1/LLC misses), so it is still a net gain. (For excessive loads to shadow the benefit of reduced L1/LLC misses, those loads need to be 10× more, as the access latency of LLC is roughly 10× of that of L1.) Configurations showing large cache miss reduction are consistent with the corresponding improved performance in Fig. 4, showing that HCSGC is effective in improving locality.

Impact on Cycles and Relocation. To understand the nature of the extra work performed by HCSGC, we compare the number of GC cycles and the amount of relocation of the various configurations with out ZGC baseline. As shown in Fig. 4, the extra loads are due to extra GC cycles and/or enlarged EC. Since the machine is not overloaded, these extra GC activities are served by the otherwise idle CPUs, which are invisible in the execution time.

Adapting to Phase Changes. Fig. 4 shows that HCSGC's object placement improves performance for an application with a single stable access phase. Ideally however, HCSGC should adapt naturally to phase changes that alter what objects are accessed by an application, and in what order. To test this, we extended the single-phase code to simulate

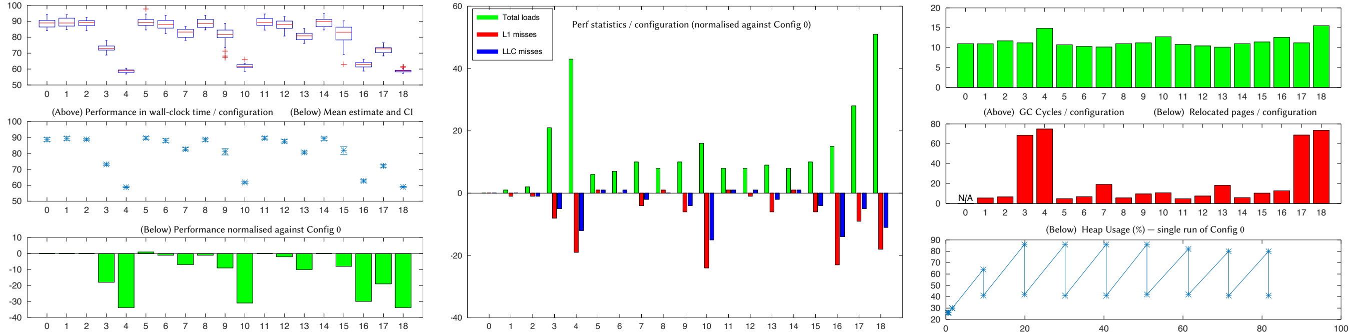

Figure 5. Synthetic benchmark going through three phases with different access patterns. (Layout following §4.2.)

going through multiple, phases, each with its own access pattern. As shown below, we use three phases, where each phase has its own seed so that within each phase, the access order is stable, but different across the phases.

Running this modified benchmark using HCSGC yielded performance results shown in Fig. 5, which is similar to the results of single-phase version presented in Fig. 4, meaning HCSGC can indeed react to phase changes.

```
1  for (int phase = 0; phase < 3; ++phase) {
2    for (int i = 0; i < 200; ++i) {
3      rand = new Random(phase); // not a constant anymore
4      ... // same as before
5    }
6  }
```

Overhead of Ample Relocation. As mentioned in §3.1.1, we hypothesise that the cost of relocating *all* pages will materialise in an overloaded system. In order to verify that, we augment the benchmark by adding an array created in the beginning, but never accessed. This "cold array" has length 2×10^7, meaning the hot–cold ratio is $1 : 10$. We use the same setup, but constrain the VM to a single core using *taskset*. The result is shown in Fig. 6, and large overhead is observed for Configs 3, 4, 17, and 18, which use RelocateAllSmallPages. On the other hand, ColdConfidence can still yield performance improvements, as in Configs 7, 10, 13, and 16.

4.5 Graph Algorithms with JGraphT

We run two benchmarks from from the JGraphT library [14]: *maximal clique (MC)* (`BronKerboschCliqueFinder`), which

Table 3. LAW Graph nodes and edges.

Dataset	Nodes	Edges	Heap (MB)
uk (complete)	100,000	3,050,615	*n/a*
uk (CC)	28,128	900,002	1,024
uk (MC)	5,099	239,294	4,096
enwiki (complete)	5,616,717	128,835,798	*n/a*
enwiki (CC)	28,126	80,002	600
enwiki (MC)	43,354	170,660	4,096

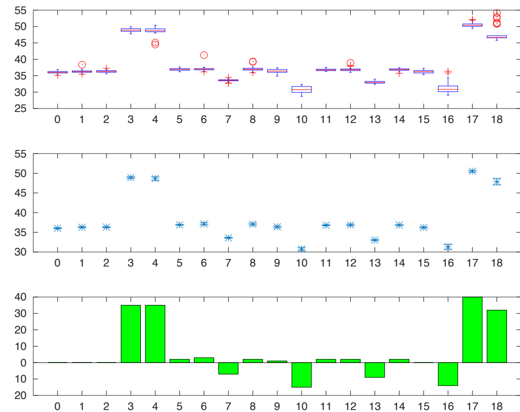

Figure 6. High overhead of RelocateAllSmallPages (on in configurations 3, 4, 17 and 18; else off) when there are many cold objects and computing resources are constrained.

implements the Bron-Kerbosch maximal clique enumeration algorithm as described in [21], and *(weakly) connected components (CC)* (`BiconnectivityInspector`), which implements biconnected components algorithm as described in [12]. Inspired by recent GC work [23, 24], we use the graph datasets *uk-2007-05@100000* and *enwiki-2018*, which are from Laboratory for Web Algorithms (LAW) [4, 5]. We implement a minimal driver which does nothing more than call the APIs from LAW to load the graph, insert all nodes to a new graph from JGraphT, and calls a method from JGraphT on the graph where almost all processing time is spent.

Processing the whole graph takes several days so we only use part of the graph as the inputs. The total graph size, the part actually used and heap size used are shown in Table 3. For *uk(CC)* and *enwiki(CC)*, as shown in Fig. 7 and Fig. 8, not much garbage is created, so the number of GC cycles is small, and most of them occur within the first 5 seconds of VM startup. However, that is enough to reorganise objects in the order facilitating mutator access, reflected by low cache missed and reduced execution time. For *uk(MC)* and *enwiki(MC)*, as shown in Fig. 9 and Fig. 10, some allocation is done by the Bron-Kerbosch algorithm, which triggers GC often, and we

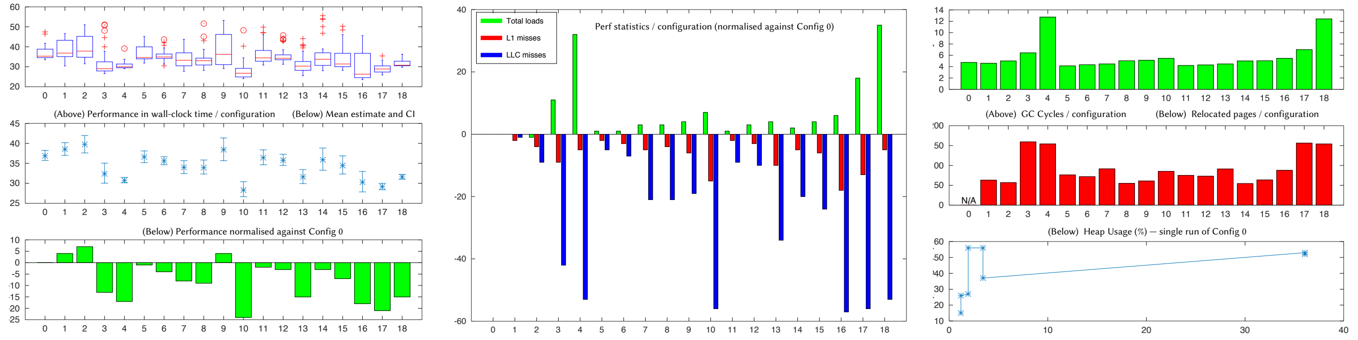

Figure 7. Connected components (CC) with JGraphT using the uk dataset. (Layout following §4.2.)

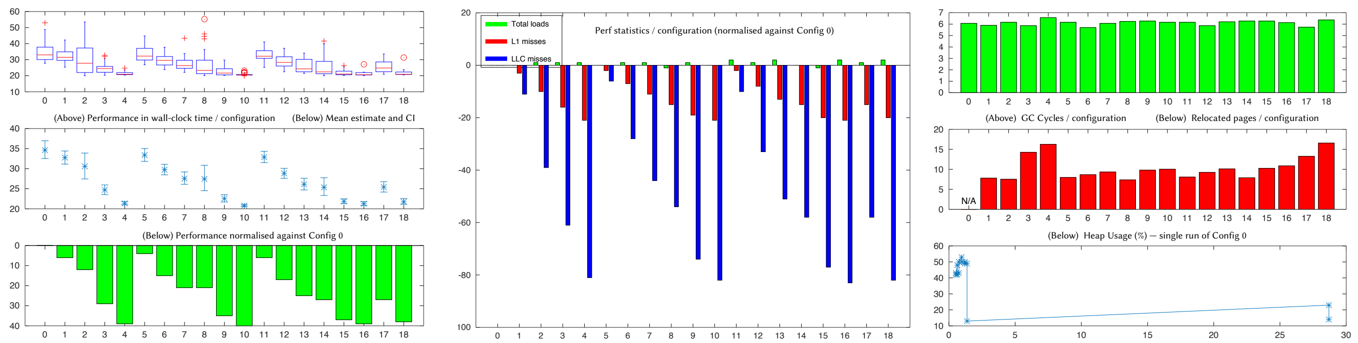

Figure 8. Connected components (CC) with JGraphT using the enwiki dataset. (Layout following §4.2.)

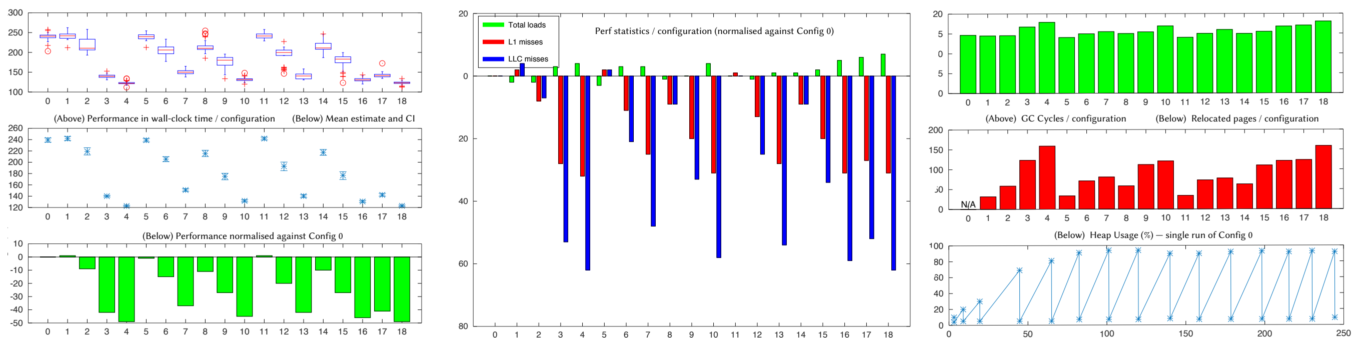

Figure 9. Bron-kerbosch (MC) algorithm with JGraphT using the UK dataset. (Layout following §4.2.)

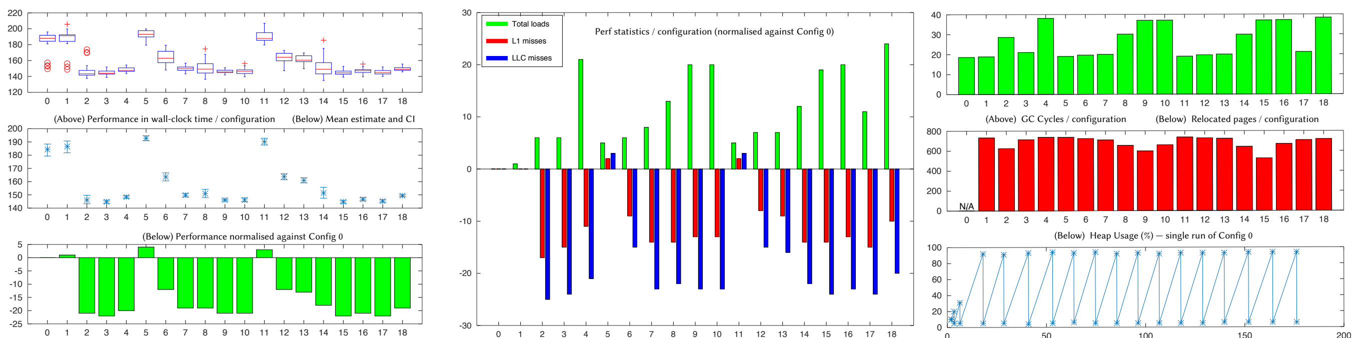

Figure 10. Bron-kerbosch (MC) algorithm with JGraphT using the enwiki dataset. (Layout following §4.2.)

see periodic GC cycles. Similar to *uk(CC)* and *enwiki(CC)*, cache misses are reduced along with execution time. Note that for *uk(MC)* there is a large difference between Config 2 and Config 3, which means that many hot objects reside on pages that are well populated so that those pages are never added to EC according to original ZGC criteria. Therefore, there is a clear staircase pattern as we increase the value of ColdConfidence in Configs 5–7, 8–10, 11–13, and 14–16; this means that we can excavate hot objects buried among cold objects using this knob.

4.6 Tradebeans and h2 from DaCapo

Next we look at benchmarks from the DaCapo suite of Java benchmarks [3], with release version 9.12-bach-MR1. Since HCSGC takes advantage of recurring and stable access patterns, it is not intended to be used with short running applications. Therefore, we only look at benchmarks that support the *huge* input size setting. Thus, the selected benchmarks are *tradebeans* and *h2*[4]. The heap size is set to 4GB, and the performance is shown in leftmost in Fig. 11 and Fig. 12. For *tradebeans*, HCSGC does not improve performance much, which we attribute to the fact that so many objects are very short lived. For such a program, locality benefits must come through placement at allocation-time, but HCSGC may only improve locality for objects that live more than one GC cycle. For *h2*, we see improvements of 5–9% for several configurations, with <2% overhead for tracking hotness (Configuration 5). RelocateAllSmallPages outperforming ColdConfidence may indicate that the same set of hot objects are accessed but with different access pattern.

4.7 SPECjbb2015

We run SPECjbb2015 with composite setting (single VM, single host) on the server with 64G heap. SPECjbb reports two scores corresponding to throughput and latency as shown in Fig. 13. As the confidence intervals are overlapping, we cannot say whether HCSGC impacts the performance of SPECjbb. We attribute this to the fact that most objects in SPECjbb do not survive a GC. Heap usage in Fig. 13 may give the impression that long-lived objects are accumulated in heap, but that is not true. SPECjbb ramps up the allocating rate in order to find when the response time stops meeting the requirement. Therefore, the heap usage after a GC cycle becomes larger, due to higher allocation rate. Additionally, ZGC starts a new GC cycle earlier to avoid out-of-memory given higher allocation rate. If we want to exclude the effect of concurrent allocation, we can check the survival rate of objects allocated prior to GC start, which is ~1%, indicating that most objects do not survive a GC cycle.

[4] *tradesoap* also supports *huge* input size. We did however see crashes similar to the description in an open issue (https://github.com/dacapobench/dacapobench/issues/113) of DaCapo (a concurrency bug which only sometimes lead to crashes), so in the end we were not able to include it.

4.8 Summary

The fact that large improvement is observed from JGrapht benchmarks, while almost no effect from SPECjbb, indicates that HCSGC depends on applications exhibiting stable access patterns on long-lived objects in order to improve locality.

Future works include using a feedback loop to auto-tune HCSGC knobs and mix configurations. A potential direction could be collecting cache miss rate, which can be used for more aggressive segregation if the result is positive or backing off otherwise. Combination of HCSGC with a generational scheme could allow short-lived objects to be handled by young GC, while HCSGC focuses on improving locality of long-lived objects. Finally, adding a new page size class, where max object size is of magnitude of cache-line size, would allow fine-grained relocation.

5 Related Work

Chilimbi and Larus use an online profiling technique to track recently accessed objects and build a temporal affinity graph [9] based on the frequency of accesses to pairs of objects within a temporal window. They devise a new copying algorithm that take advantage of affinity so that objects with high affinity will be grouped together. In contrast to this pioneering work, HCSGC does not need to construct any such graph as mutators are tasked with reorganising the objects by accessing them. Not only will objects with high temporal affinity be grouped together, but also they are laid out in the optimal order (assuming a stable access pattern). As far as we can tell, Chilimbi and Larus' approach does not suffer from problems due to objects being hot across several mutators, who will then compete to relocate them.

Adl-Tabatabai et al. [2] uses hardware performance monitoring to identify cache miss-intensive traversal paths through linked data structures, and then remedy the situation by having the JIT compiler inserting appropriate prefetch instructions. Our approach clusters frequently accessed objects together and reorganises them in the order mutators accessing them, which is friendly to plain hardware prefetching.

Online object reordering [13] takes advantage of a moving garbage collector (a generational copying collector) to improve spatial locality. It uses sampling to identify hot methods, from which hot fields are discovered. When a (minor or major) GC cycle starts, hot fields are copied before cold ones. Note that while this captures and improves topological locality (hot fields together with their parent), our approach work also on objects that not connected by a pointer.

Chen et al.'s work on locality optimisation [8] collects object access information via throttled samples, and subsequently uses this information to improve temporal cache locality and page locality (objects recently accessed are laid out continuously in a hierarchical decomposition order). In contrast, our approach rearranges the hot objects in the order that mutators access them regardless of their structural

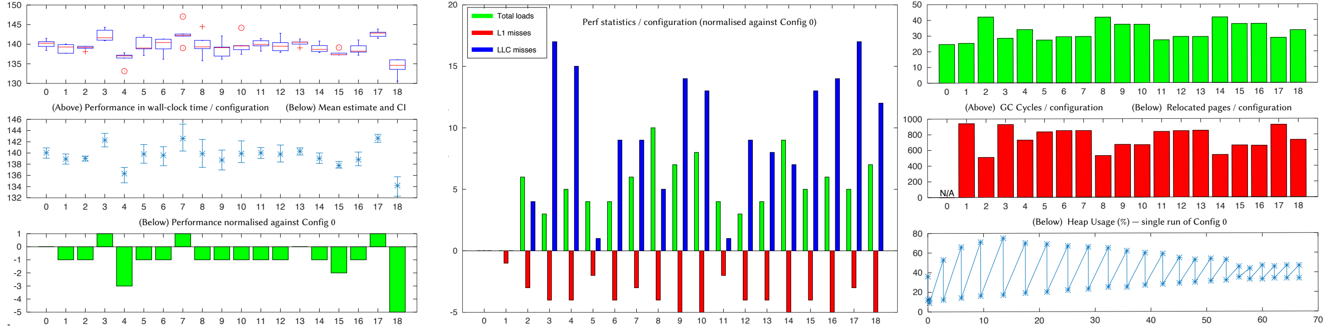

Figure 11. Results for DaCapo's *tradebeans* benchmark. (Layout following §4.2.)

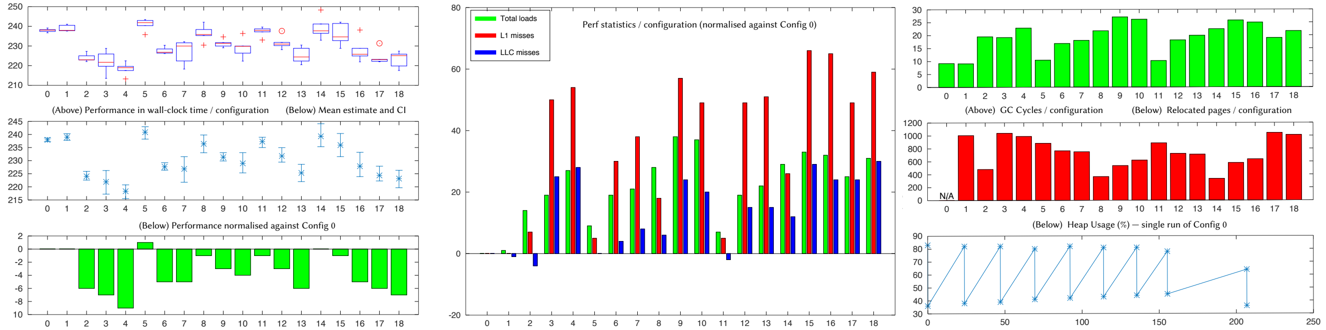

Figure 12. Results for DaCapo's *h2* benchmark. (Layout following §4.2.)

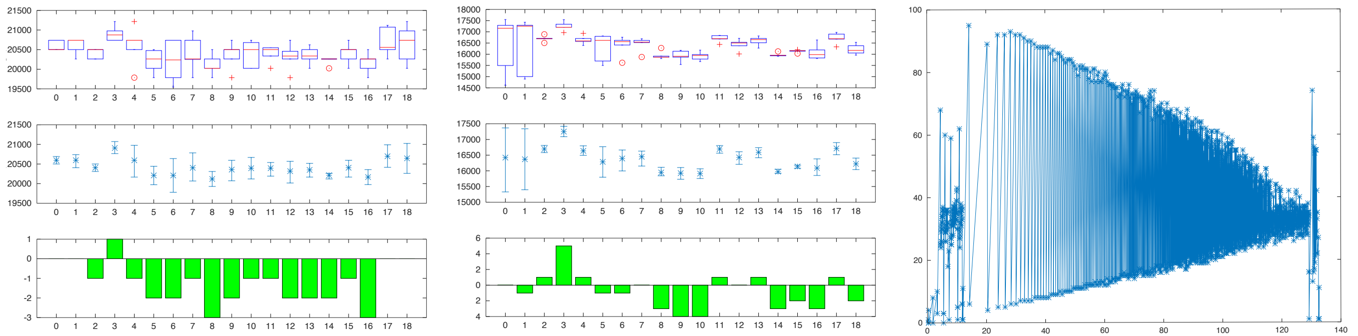

Figure 13. SPECjbb2015. Leftmost: throughput score. Middle: latency score. (Higher is better in both cases.) Rightmost: Heap usage of a single run of Config 0 with exec time in minutes on the x axis.

relationship. We believe that this improves both cache and page locality if the access pattern is stable. However, Chen et al.'s work is more flexible in terms of being able to run independent of GC cycles and being throttled if an application does not exhibit any locality that can be leveraged.

6 Conclusion

HCSGC is a highly configurable approach to transparently improve cache locality of managed languages by using mutators to perform relocation of objects in a prefetching-friendly way, where stable access paths enjoy good locality. HCSGC supports various modes of operation, some of which impose very little overhead for a crude design, and others which make more informed decisions at the expense of additional run-time tracking overhead.

While our SPECjbb results are inconclusive due to a fluctuating baseline, HCSGC shows speedup for tradebeans (5% at best) and h2 (9% at best), with 95% confidence. For JGraphT, several HCSGC configurations deliver substantial speedups: ≈20%, ≈35% and ≈45%, all with 95% confidence. For all benchmarks, a few configurations yield negative performances, but considerably less than the performance increases.

HCSGC demonstrates great potential that GC, via hotness tracking, could deliver significant performance improvement by distinguishing live objects into hot and cold, and further segregating these groups, with marginal overhead.

References

[1] Diab Abuaiadh, Yoav Ossia, Erez Petrank, and Uri Silbershtein. 2004. An Efficient Parallel Heap Compaction Algorithm. *SIGPLAN Not.* 39, 10 (Oct. 2004), 224–236. https://doi.org/10.1145/1035292.1028995

[2] Ali-Reza Adl-Tabatabai, Richard L. Hudson, Mauricio J. Serrano, and Sreenivas Subramoney. 2004. Prefetch Injection Based on Hardware Monitoring and Object Metadata. *SIGPLAN Not.* 39, 6 (June 2004), 267–276. https://doi.org/10.1145/996893.996873

[3] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[4] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM Press, 587–596.

[5] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[6] Michael D. Bond and Kathryn S. McKinley. 2008. Tolerating Memory Leaks. *SIGPLAN Not.* 43, 10 (Oct. 2008), 109–126. https://doi.org/10.1145/1449955.1449774

[7] Michael D. Bond and Kathryn S. McKinley. 2009. Leak Pruning. *SIGARCH Comput. Archit. News* 37, 1 (March 2009), 277–288. https://doi.org/10.1145/2528521.1508277

[8] Wen-ke Chen, Sanjay Bhansali, Trishul Chilimbi, Xiaofeng Gao, and Weihaw Chuang. 2006. Profile-guided Proactive Garbage Collection for Locality Optimization. *SIGPLAN Not.* 41, 6 (June 2006), 332–340. https://doi.org/10.1145/1133255.1134021

[9] Trishul M. Chilimbi and James R. Larus. 1998. Using Generational Garbage Collection to Implement Cache-conscious Data Placement. *SIGPLAN Not.* 34, 3 (Oct. 1998), 37–48. https://doi.org/10.1145/301589.286865

[10] Juliana Franco, Martin Hagelin, Tobias Wrigstad, Sophia Drossopoulou, and Susan Eisenbach. 2017. You Can Have It All: Abstraction and Good Cache Performance. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Vancouver, BC, Canada) *(Onward! 2017)*. ACM, New York, NY, USA, 148–167. https://doi.org/10.1145/3133850.3133861

[11] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications* (Montreal, Quebec, Canada) *(OOPSLA '07)*. ACM, New York, NY, USA, 57–76. https://doi.org/10.1145/1297027.1297033

[12] John Hopcroft and Robert Tarjan. 1973. Algorithm 447: Efficient Algorithms for Graph Manipulation. *Commun. ACM* 16, 6 (June 1973), 372–378. https://doi.org/10.1145/362248.362272

[13] 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. *SIGPLAN Not.* 39, 10 (Oct. 2004), 69–80. https://doi.org/10.1145/1035292.1028983

[14] jgrapht 2019. https://jgrapht.org/

[15] Rahman Lavaee. 2016. The Hardness of Data Packing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 232–242. https://doi.org/10.1145/2837614.2837669

[16] Per Lidén. 2017. CFV: New Project: ZGC. OpenJDK Mailing List. http://mail.openjdk.java.net/pipermail/announce/2017October/000237.html.

[17] Per Lidén and Stefan Karlsson. 2018. JEP 333: ZGC: A Scalable Low-Latency Garbage Collector. http://openjdk.java.net/jeps/333. Accessed: 2019-04-05.

[18] Per Lidén and Stefan Karlsson. 2018. The Z Garbage Collector—Low Latency GC for OpenJDK. http://cr.openjdk.java.net/~pliden/slides/ZGC-Jfokus-2018.pdf. Accessed: 2019-04-05.

[19] Robert Mcgill, John W. Tukey, and Wayne A. Larsen. 1978. Variations of Box Plots. *The American Statistician* 32, 1 (1978), 12–16. https://doi.org/10.1080/00031305.1978.10479236

[20] Erez Petrank and Dror Rawitz. 2002. The Hardness of Cache Conscious Data Placement. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon) *(POPL '02)*. Association for Computing Machinery, New York, NY, USA, 101–112. https://doi.org/10.1145/503272.503283

[21] Ram Samudrala and John Moult. 1998. A graph-theoretic algorithm for comparative modeling of protein structure11Edited by F. Cohen. *Journal of Molecular Biology* 279, 1 (1998), 287 – 302. https://doi.org/10.1006/jmbi.1998.1689

[22] 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. *SIGPLAN Not.* 37, 11 (Nov. 2002), 13–25. https://doi.org/10.1145/583854.582422

[23] Po-An Tsai and Daniel Sanchez. 2019. Compress Objects, Not Cache Lines: An Object-Based Compressed Memory Hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 229–242. https://doi.org/10.1145/3297858.3304006

[24] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. 2019. Panthera: Holistic Memory Management for Big Data Processing over Hybrid Memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 347–362. https://doi.org/10.1145/3314221.3314650