Dynamic Prediction of Collection Yield for Managed Runtimes

Michal Wegiel

Computer Science Department
University of California, Santa Barbara
mwegiel@cs.ucsb.edu

Chandra Krintz

Computer Science Department
University of California, Santa Barbara
ckrintz@cs.ucsb.edu

Abstract

The growth in complexity of modern systems makes it increasingly difficult to extract high-performance. The software stacks for such systems typically consist of multiple layers and include managed runtime environments (MREs). In this paper, we investigate techniques to improve cooperation between these layers and the hardware to increase the efficacy of automatic memory management in MREs.

General-purpose MREs commonly implement parallel and/or concurrent garbage collection and employ compaction to eliminate heap fragmentation. Moreover, most systems trigger collection based on the amount of heap a program uses. Our analysis shows that in many cases this strategy leads to ineffective collections that are unable to reclaim sufficient space to justify the incurred cost. To avoid such collections, we exploit the observation that dead objects tend to cluster together and form large, never-referenced, regions in the address space that correlate well with virtual pages that have not recently been referenced by the application. We leverage this correlation to design a new, simple and light-weight, yield predictor that estimates the amount of reclaimable space in the heap using hardware page reference bits. Our predictor allows MREs to avoid low-yield collections and thereby improve resource management.

We integrate this predictor into three state-of-the-art parallel compactors, implemented in the HotSpot JVM, that represent distinct canonical heap layouts. Our empirical evaluation, based on standard Java benchmarks and open-source applications, indicates that inexpensive and accurate yield prediction can improve performance significantly.

Categories and Subject Descriptors   D.3.4 [*Programming Languages*]: Processors—Memory Management (Garbage Collection)

General Terms   Languages, Management, Performance

1. Introduction

To support the vast diversity of deskside and server workloads, modern system software stacks have grown both in depth and complexity. Two key layers of extant software stacks are a general purpose operating system (OS), e.g., Linux, and managed runtime environments (MREs), e.g., Java and C# virtual machines. Although independent and isolated, these software layers provide similar services for programs that include memory management, access to protected resources, and resource scheduling.

In this work, we investigate how to better coordinate the activities of memory management between the hardware, OS, and MREs to improve the performance of applications that employ them. Automatic memory management (garbage collection) is commonly implemented by MREs. While increasing programmer productivity and application reliability through memory safety, automatic memory management can also negatively impact both application throughput (through additional MRE processing) and interactivity (by imposing pauses). Minimizing the cost of garbage collection to match or exceed that of explicit memory management has been the subject of active research for several decades [19, 21, 33, 34, 22, 32, 6, 37, 15].

Key advances in automatic memory management that have led to significant improvements in program performance include support for parallelism and concurrency, generational collection, and compaction [21, 17, 22, 6, 7, 32, 10, 25]. Parallel and concurrent systems exploit increasing numbers of processing cores to improve application scalability, throughput, and to reduce pause times. Generational systems outperform other garbage collection (GC) schemes in the common case by managing young objects independently from old objects. Compaction enables very fast linear (bump-pointer) object allocation and can significantly improve memory hierarchy performance by eliminating heap fragmentation.

Moreover, recent GC systems introduce ways to better coordinate the activities of the MRE GC and OS virtual memory support [32, 22, 13, 27, 37, 15, 38, 36, 14]. These systems exploit the OS virtual memory subsystem and CPU memory management unit, and optimize for access locality

to minimize TLB and cache misses, to reduce the overhead of garbage collection.

In this work, we build upon and extend this prior work to improve MRE-OS-hardware interaction in a way that facilitates *GC avoidance*. In particular, we design and implement a simple prediction scheme that identifies, with very low overhead, the amount of free space a particular GC invocation is likely to yield from dead objects. GC systems can employ this yield prediction to avoid ineffective collections that are unable to reclaim sufficient space to justify the incurred cost, by trading off a small space overhead (equal to the small yield that would have been collected by the skipped GC). Most extant systems trigger GC unconditionally when a program exceeds some threshold on its heap use, without regard for GC yield. Systems that trigger GC proactively, rely on complex monitoring and analysis of program behavior [35].

The *Yield Predictor (YP)* provides a simple solution to distinguishing productive GCs by estimating GC yield using hardware page reference bits that the OS uses to implement virtual page replacement. Key to its efficacy, YP exploits the statistical properties of modern programs that dead objects tend to cluster together in large groups (larger than the 4KB virtual page size), and that pages that have not been recently referenced by the application are likely to be dead. We validate these properties with empirical data and describe how YP makes use of them to estimate GC yield. We implement YP for three state-of-the-art parallel compactors within the production-quality, open-source, HotSpot virtual machine from Sun Microsystems.

In summary, we contribute:

- A demonstration that MREs can significantly benefit from existing architectural support for memory management when GC is given access to page tables and can leverage the standard hardware (HW) mechanism used for marking pages referenced by a process.

- The design and implementation of YP, a generic GC yield predictor that employs virtual page reference bits to determine the amount of dead space present in the heap with low overhead and high accuracy.

- Techniques for improved MRE/OS/HW coordination in the context of memory management showing that well-arranged cross-layer interaction benefits system performance, and identification of a new use for the existing hardware facility (i.e., page reference bits) and empirical validation of its effectiveness.

- A more resource-efficient GC mechanism that gives the memory manager more control over the space/time trade-off and allows for well-informed GC scheduling decisions at run-time.

- An investigation of the applicability of YP to three canonical heap layouts used by state-of-the-art parallel/concurrent compactors: sliding, copying, and remapping. Our analysis reveals that YP is equally effective regardless of the object relocation strategy.

- A comprehensive experimental evaluation of YP based on standard community benchmarks and open-source applications. We show that YP consistently provides high prediction accuracy and that avoidance of unproductive GCs can substantially improve (44–59%) the performance of both server- and client-side benchmarks.

In the following sections, we provide background on the problem domain, detail the design, implementation, and evaluation of YP, and discuss related work.

2. Background

State-of-the-art MRE memory management systems trigger garbage collection (GC) unconditionally upon complete (or partial, in the case of concurrent GC) exhaustion of the space designated for object allocation. The key limitation of this approach is that every collection is performed irrespective of whether it is worth paying for. Our investigation of *GC productivity*, i.e., the total size of dead objects that a collection cycle reclaims, shows that many GCs are unproductive and are able to reclaim only a small fraction of the heap. We empirically evaluate this phenomenon further in Section 5.3. We observe from our experiments that a number of different (deskside and server) Java applications have an average GC yield below 5% of the heap space.

If we are to skip unproductive GCs, we must have a fast and accurate mechanism for estimating the total size of dead objects in the heap. Marking, which identifies live data via traversing the reachable object graph starting from the roots, is the most commonly-used mechanism to do this and can precisely compute the amount of dead space. However, marking takes significant time and, according to our measurements, comprises between 50% (for the Compressor) and 90% (for the Mapping Collector) of total GC time, depending on the compaction algorithm. Thus, partial reduction of collection cost by skipping only the phases that follow marking is not satisfactory due to its lower potential for performance improvement.

The goal of our work is to design and implement effective and practical yield prediction that is lightweight. Our approach to enabling such prediction relies on improved coordination between the activities of OS and hardware virtual memory support and MRE memory management.

2.1 Virtual Memory

General-purpose OSes support virtual memory to isolate address spaces of distinct processes and provide a convenient uniform linear addressing. Appel and Li [1] describe a number of ways that user-level programs can take advantage of virtual memory operations, including user-level signal handlers, multiple page mapping, and page protection. A similar study that includes performance evaluation as well as recom-

mendations for exposing dirty-bit information to MREs can be found in [16].

Most virtual memory implementations divide the virtual address space of a process into *pages*, each typically 4KB in size. The mapping between virtual pages and physical page frames is stored in an OS-maintained *page table*.

Under memory pressure, the kernel uses *swap space* to evict pages that are unlikely to be accessed in the future. To implement swapping, pages tables reserve two bits per page, indicating whether a specific page is *dirty* and has been recently *referenced*. These bits are set by hardware upon memory store/read.

2.2 Parallel Compaction

A general introduction into GC can be found in [21, 33, 34]. Modern compactors are parallel and increasingly take advantage of virtual memory operations. Below, we overview the state-of-the-art parallel/concurrent compactors that we focus on herein. The first phase in the GC cycle for each of these (and most) GCs is *marking*, which identifies live objects through parallel/concurrent tracing from a program's root references. The remaining phase(s) are specific to each compactor.

The Compressor   The Compressor (CP) [22] is a parallel/concurrent, two-phase GC that performs compaction by evacuation of live objects from one virtual space to the other. CP manipulates page mapping to avoid space overhead of a copying collection. CP evacuates objects page by page, (un)mapping pages on-the-fly, and updates pointers using relocation information stored in auxiliary data structures (marking bitmap/block offset array). CP achieves full compaction (no fragmentation) and preserves object order. Concurrent CP uses page protection to implement a read/write barrier based on SEGV faults and double mapping to enable incremental compaction.

The HotSpot Compactor   The HotSpot collector (HS) [17, 32] is parallel, comprises two phases, and performs sliding order-preserving compaction within a single space. HS permits some amount of fragmentation in the heap as long as the amount of wasted space is sufficiently small (configurable parameter). In each compaction, HS computes the density prefix, which is not moved. The remaining objects are relocated. HS implements a block-based compaction and uses the liveness bitmap and per-block statistics to update pointers. GC threads fill regions with objects as they become empty or compacted.

The Mapping Collector   The Mapping Collector (MC) [32] is a parallel/concurrent, nearly-one-phase non-moving compactor that exploits clustering of dead objects in the heap to reclaim free space on a per-page basis. MC first identifies dead regions and then unmaps them, thus avoiding object copying and pointer adjustment altogether and simplifying GC design. Since small and unaligned clusters cannot be fully unmapped, MC imposes a space overhead. Although variable, this overhead tends to be small in practice and it can be bounded via a fallback to conventional compaction (such as that provided by HS or CP).

3. The Yield Predictor (YP)

Prior work shows that for modern Java applications, objects with similar life spans tend to be spatially clustered in the heap and that dead objects often form clusters larger than the 4KB virtual page size [32, 34]. The design of YP leverages these statistical properties and the observation that pages of dead objects (dead pages) are never accessed by the program and, as a result, eventually become not-recently-referenced (NRR) from the OS kernel perspective. YP exploits this relationship between NRR and dead pages to estimate the total number of dead pages in the heap. When the number of dead pages is small, an impending garbage collection is likely to be ineffective, i.e., unable to reclaim sufficient space to justify the cost of performing GC. We employ YP to avoid such collections in state-of-the-art compaction systems. We trade a small space overhead for significant performance gains that result from skipping low-yield GCs. We analyze these trade-offs in Section 5.

3.1 YP Design

YP takes two parameters: *skip threshold* and *young-old ratio*. We investigate YP's sensitivity to both in Section 5. The skip threshold determines the free proportion of the heap that is necessary to trigger regular collection (e.g., for the skip threshold of x we skip all GCs that we predict to reclaim not more than x% of heap space). The young-old ratio identifies pages that are recently-referenced (RR). YP considers a time window between two consecutive GCs and divides it into two contiguous parts: young and old, according to the young-old ratio. Pages with a last-access timestamp in the young partition are considered to be live.

There are two sources of potential inaccuracy in YP's yield prediction process. The first is a page that we identify as NRR, that is not actually dead but is instead, not accessed recently. The second is a page that is dead that we have not yet identified as NRR. Although all dead pages are guaranteed to be found eventually, there may be a delay before YP correctly classifies a page as dead.

Since YP considers only the total number of dead pages in the heap, as opposed to determining the status (live/dead) for each individual page, the dead space estimation errors that these two phenomena introduce do not add up, but instead, cancel each other out. Thus, the goal of YP parameter tuning is to make sure that the two misclassifications (dead-as-live and live-as-dead) occur at similar frequency. Therefore to optimize accuracy we need to choose the best old-young ratio. For large ratios live-as-dead misclassification dominates. For small ratios dead-as-live misclassification dominates. For the right choice of the young-old ratio

the two misclassifications are similarly frequent and YP accuracy reaches its optimum.

YP periodically consults the OS kernel to obtain a list of recently-referenced (RR) pages within the heap. A dedicated polling thread in the MRE wakes up at regular time intervals and retrieves the addresses of RR pages. For each page in the heap, YP records the time when a page was last believed to be RR, using a `timestamp` array stored in the MRE.

Each time the polling thread tests the reference bits, it clears them atomically. To avoid interference with the kernel swapping mechanism that also relies on RR bits, we introduce two new bits per page: *mre-cleared* and *os-cleared*. This extension is software-only. YP shares hardware RR bits with an OS, but whenever YP (or an OS) clears a hardware RR bit, we set the mre-cleared (or the os-cleared) bit in software so that no information is ever lost. We multiplex hardware bits and use software bits to indicate if a HW bit was cleared. To read an RR bit of a page, YP (or an OS) computes a *logical-or* of the hardware RR bit and the os-cleared bit (or the mre-cleared bit). When MRE (or an OS) clears the RR bit, it also must clear the os-cleared (or mre-cleared) bit.

Key to our approach is accurate RR page tracking. First, we must distinguish between an application access to a page and a GC access to a page, and only consider the former as an RR trigger (since GC may reference pages not reachable by the program). To enable this, before every minor/major collection, YP takes a snapshot of the current RR page bits, and after each GC clears the reference bits that are set as a side-effect of the collection. Moreover, we disable the polling thread during GC.

In addition, YP measures the time spent in GC and advances the values in the `timestamp` array accordingly after each collection. This is necessary to eliminate the impact of the stop-the-world GC pauses on timestamps. During GC pauses, mutators are inactive and live pages are not used, which can make them appear to be NRR. Advancing timestamps eliminates this problem.

YP maintains a boolean array (`mispredicted_dead`), for all pages in the heap. Each entry indicates whether a live page has been misclassified as dead. Such pages are never considered dead again. The intuition behind this is that many applications allocate permanent data structures that subsequently are rarely used which can lead to YP false positives.

Since compactors typically move objects (MC is the exception), after each collection for such compactors, YP recomputes the `mispredicted_dead` array to reflect the relocation that has occurred in the heap. For each live target page, we consider the (live) source pages containing objects being moved to the target page. If any source page has ever been misclassified as dead (including during the current GC cycle) then the target page is marked as `mispredicted_dead`. We perform this propagation since the target page is also likely to be misclassified.

YP makes predictions of the potential GC yield while mutators are suspended (i.e., at a safepoint). Prediction is short and simple and therefore does not need to execute concurrently. Parallelization is not necessary either as prediction cost is proportional to the number of pages in the heap. The polling thread executes concurrently and asynchronously to mutators. It does not employ locking or synchronization and imposes negligible overhead, especially when executed on a separate CPU/core.

3.2 Yield Prediction

Table 1 shows the pseudocode for the steps that YP executes during each full GC. YP first stops the polling thread. Next, it obtains RR pages and updates the page timestamps (lines 1–4). The predictor then iterates over the heap pages to determine which pages are dead (lines 5–14); YP skips any pages previously found to result in false positives (`mispredicted_dead[page]` is true). For other pages, we consider their age, i.e., how long since they were accessed, and compare that against a percentage of the distance (in time) between the current and previous full GC. This percentage is a YP parameter (the young-old ratio).

If the predicted amount of free space is larger than the skip threshold, the system falls back to regular compaction (i.e., does not skip GC). Following compaction, YP attempts to shrink the heap back to the size it was prior to GC-skipping (if any) to reduce space overhead (line 21). Finally, we re-compute the `mispredicted_dead` array (lines 23–35), using the auxiliary array `propagated_dead`. We traverse the live heap pages computing a target location for each such page. If the source page has ever been mispredicted dead or has been predicted dead in the current GC cycle (note that this page is live), the target page becomes `mispredicted_dead`.

If the predictor expects low yield, it skips the compaction and grows the heap (lines 16–18). The expansion corresponds to the predicted free space, but is never smaller than the minimum value (`min_expansion`, 128KB in our implementation). This minimum is necessary to ensure mutator progress, i.e., to ensure that the mutator is able to allocate the data that triggered the GC originally. In the GC epilogue (lines 38–42), we clear the reference bits and advance the timestamps by the GC pause time.

We never skip the first collection as it is typically highly productive. Instead, we use this collection to bootstrap the predictor and initialize its data structures.

When skipping an unproductive collection, we extend the heap by the estimated total size of dead objects (which themselves are not reclaimed as without marking one cannot identify them). This creates space overhead. This overhead is small in practice because we skip only low-yield collections and shrink the heap when possible on subsequent full collections.

Note that GC skipping is substantially different than heap over-provisioning. Executing an application with a larger

```
 1: rr_list = get_rr_pages(heap_start, heap_end)
 2: for page in rr_list do
 3:   timestamp[page] = current_time
 4: end for
 5: dead_cnt = 0
 6: limit = OLD_YOUNG_RATIO · (current_time − last_full_gc)
 7: set_all_entries(predicted_dead, false)
 8: for page in [heap_start, heap_end] do
 9:   age = current_time − timestamp[page]
10:   if age >= limit and not mispredicted_dead[page] then
11:     predicted_dead[page] = true
12:     dead_cnt += page_size
13:   end if
14: end for
15: if dead_cnt ≤ SKIP_THRESHOLD · heap_size then
16:   dead_cnt = max(dead_cnt, min_expansion)
17:   expand_heap(dead_cnt)
18:   total_expansion += dead_cnt
19: else
20:   fall_back_to_regular_gc
21:   try_to_shrink_heap(total_expansion)
22:   update_if_heap_shrunk(total_expansion)
23:   set_all_entries(propagated_dead, false)
24:   for page in [heap_start, heap_end] do
25:     if has_live_objects(page) then
26:       if mispredicted_dead[page] or predicted_dead[page] then
27:         target = relocation_target(page)
28:         propagated_dead[target] = true
29:         if crosses_next_page(page, target) then
30:           propagated_dead[successor(target)] = true
31:         end if
32:       end if
33:     end if
34:   end for
35:   mispredicted_dead = propagated_dead
36:   update_if_relocated(heap_start, heap_end)
37: end if
38: clear_rr_pages(heap_start, heap_end)
39: for page in [heap_start, heap_end] do
40:   timestamp[page] += gc_time
41: end for
42: last_full_gc = current_time
```

Table 1. Pseudocode for yield prediction executed by YP during each full collection. SKIP_THRESHOLD and YOUNG_OLD_RATIO are the two YP parameters.

heap does not prevent unproductive GCs, although it does reduce the total number of collections. YP ensures with high probability that the system triggers GC only when it is worth doing so. Thus, YP enables better resource management and gives the memory manager greater control over space/time trade-offs. For example, when an expensive GC algorithm is used, an MRE might be more conservative when deciding to trigger a collection. In addition, the user need not determine the right heap size a priori.

Note that on 64-bit platforms, the space costs are the same as on 32-bit platforms – the arrays that we use have one entry per page and pertain only to the area used and mapped by the old generation.

With concurrent GC, YP has similar or even more potential for improving performance. Each cycle of concurrent GC, despite imposing shorter pause times, costs more than the corresponding cycle of the stop-the-world GC.

4. Implementation

We integrate YP into three state-of-the-art parallel compactors in order to investigate its generality and applicability to different collectors. These compactors represent three canonical heap layouts that underlie all modern GC algorithms (including concurrent ones). We use exactly the same prediction algorithm with each compactor.

We implement YP in HotSpot [24], an open-source (GPL), production quality JVM (from Sun Microsystems) written in C++ (source code version 7-ea-b10, released 3/2007). HotSpot uses a generational [30] heap layout comprising the permanent, old, and young generation. The permanent generation contains run-time meta-data for the loaded classes. The young generation is further subdivided into *eden* and two equally-sized survivor spaces (called *from* and *to*). Objects are initially allocated in the eden. Within the young generation a copying collector [11] evacuates live objects from the eden-space and from-space to the to-space and promotes objects that survive several minor collections to the old generation. Major collection (compaction) takes place upon space exhaustion in the old generation.

4.1 Kernel Extensions

We have implemented YP using Linux kernel 2.6.17 configured with high memory disabled and SMP enabled. YP consists of a kernel module which, upon loading, creates a new entry in the *proc* filesystem using the `proc_mkdir` and `create_proc_info_entry` functions. The entry, is located at */proc/ref/bits* and is writable but not readable. A polling thread in an MRE repeatedly sleeps for 10ms, opens the */proc/ref/bits* file, writes three words to it (in order to obtain a list of RR pages within a given address range), and closes the file. These words are: start and end addresses of the memory range plus a pointer to an array for the results. The kernel invokes the callback registered by the module, copies the three words from the user space, inspects page table entries corresponding to the specified address range and copies the results into the MRE-provided array (in userland). The first array entry contains the number of the returned pointers to RR pages.

We obtain the page table entries (PTEs) for subsequent pages using the macros: `pgd_offset`, `pud_offset`, `pmd_offset`, and `pte_offset_map`. We clear the reference bits in PTEs atomically after testing with the help of `ptep_test_and_clear_young`. The polling thread holds a spin lock for the page table of the current process during the entire operation.

To avoid interference with kernel swapping, we make use of the two unused bits in page flags (bits 21 and 22) which we define as *PG_kernel_cleared* and *PG_mre_cleared*. Each physical page managed by the kernel has a page frame descriptor (*struct page*) associated with it, which contains an *unsigned long flags* field. The flags determine if a page is referenced, dirty, locked, etc. Note that these software flags are distinct from hardware page flags present in PTEs. The kernel module sets the *PG_mre_cleared* bit whenever clearing the RR bit in a PTE. The kernel sets the *PG_kernel_cleared* bit whenever it clears the RR bit in a PTE. The latter requires minor modification to the kernel (the code fragments that get/set PTE RR bits).

4.2 Alternative Approaches

We have investigated two other approaches to implementing YP: mlock-based and kswapd-based. The mlock-based design employs page pinning for the old generation (via the POSIX `mlock` system call). Pinning eliminates interference of page access bit clearing (done by an MRE) with kernel swapping mechanism. This approach is simple but requires heap pages to be locked in physical memory.

In the kswapd-based design, instead of an MRE periodically clearing page access bits, we reuse an existing kernel thread (kswapd daemon) and decrease its sleeping interval. Kswapd clears page access bits whenever it wakes up. We increase the frequency of kswapd wake-up to match the bit clearing frequency needed by YP. MREs have read-only access to RR bits and the kernel swapping mechanism benefits from higher sampling frequency of RR pages (better accuracy). However, this approach assumes the existence of kswapd and its certain behavior (periodic wakeup and RR bits clearing) which makes it less portable (e.g., it works in Linux 2.4 but not in 2.6).

5. Evaluation

We empirically evaluate YP using three state-of-the-art parallel compactors (cf. Section 2.2): the Mapping Collector (MC), the Compressor (CP), and the HotSpot compactor (HS). We first overview our experimental methodology and benchmark suite. In the subsequent subsections, we present results from our experiments that measure YP prediction accuracy and cost as well as the impact of YP on the application throughput, GC pause times, and memory footprint. In addition, we systematically evaluate YP sensitivity to different values of its two parameters: *skip threshold* and *young-old ratio*.

5.1 Methodology

Our experimental platform is a dedicated dual-core Intel Core 2 Duo (Conroe B2) machine clocked at 2.66GHz with the unified 4M 16-way L2 cache and 32K 8-way L1 cache, 2GB main memory, running Debian GNU/Linux 3.0 configured with the 2.6.17 kernel. The virtual page size is 4KB.

We use HotSpot version 7-ea-b10 deployed within OpenJDK 1.6.0 and compiled with GCC 3.2.3, in the optimized client-compiler (C1) mode.

We employ YP for old-generation collection, i.e., full-heap, major GCs, only. Minor collections use a parallel copying collector in the young generation. For each benchmark, we investigate four heap sizes within a range that captures significant to medium GC activity, wherever possible. Each of our experiments uses a fixed-size heap, which consists of the young, old, and permanent generation. The young generation is 25% of the old generation. The permanent generation is 12MB (HotSpot default). We disable all explicit GC invocations and adaptive generation resizing. We employ 2 parallel GC threads as we use a dual-core machine. Survivor spaces occupy 33% of the young generation (the remainder is used by the eden). When reporting heap size, we sum up the size of all three generations.

We repeat each measurement 5 times and report the average as well as standard deviation where appropriate. We evaluate YP in detail for the skip threshold set to 5% and the young-old ratio set to 1%. In addition, we investigate its sensitivity to other skip thresholds (0%, 3%, and 10%) and young-old ratios (2–90%).

Our evaluation is based on 16 Java programs which include standard Java benchmarks and open-source Java applications [12]. We use the subset of the DaCapo [9] and SPEC JVM'98 [29] benchmark suites. In addition, we employ SPEC PseudoJBB'00 [29] and VolanoMark [31]. We selected these benchmarks to capture a wide range of application behaviors while focusing on programs with significant GC activity. Table 2 reports performance data for these benchmarks obtained using HS: heap size ranges, execution times, and general GC statistics. We investigate server-side multi-threaded workloads using VolanoMark, Pseudo-JBB'00, and Hsqldb. The remaining benchmarks are desk-side utilities.

We run the default variants of the DaCapo benchmarks and use the input size of at least 100 for JVM'98. We execute VolanoMark with 42 chat rooms for 100 iterations and PseudoJBB with 5 warehouses for 10^5 iterations.

5.2 Dead Object Clustering

The prediction capabilities of YP depend on dead object clustering, a widely-known phenomenon, previously reported in [32, 34]. We have gathered basic clustering statistics across the benchmarks, such as average, minimum, and maximum cluster size as well as the percentage of dead space fully covered by 4KB pages. The results are summarized in Table 3. For each benchmark we report the average values obtained across all GCs that occurred for the heap size ranges that we use. We have observed that most clusters are smaller than 4KB, however average cluster size is above 200KB. We have found that at least 50% of the dead space is fully covered by 4KB pages. Such clustering generally

	Benchmark Program	Heap Size Range [MB]	Execution Time [s]	GC Time [s]	GC Count	GC Cost [%]	Reclaimable Space [%]	Program Description
GPL	beautyj	61 – 64	18.2	13.0	60	71.4	2.0	Source code beautifier
	findbugs	82 – 97	13.0	2.7	5	21.1	31.0	Java bug detector
	jaranalyzer	14 – 17	4.5	0.1	3	3.0	23.2	Dependency manager
	javaguard	16 – 22	7.0	3.7	69	53.8	0.5	Bytecode obfuscator
	jdepend	30 – 33	20.5	6.7	77	32.7	0.3	Dependency analyzer
DaCapo	chart	45 – 48	6.2	0.4	3	5.8	53.7	Line graph plotter
	fop	14 – 20	4.3	1.9	31	43.6	4.9	XSL-FO parser/formatter
	hsqldb	92 – 95	12.2	7.2	11	58.7	0.9	In-memory database
	pmd	40 – 46	6.2	1.2	7	18.6	51.1	Source code analyzer
	xalan	44 – 68	6.0	1.1	21	18.0	68.7	XML to HTML transformer
JVM	compress	41 – 47	2.6	0.0	3	1.8	49.3	LZW packer
	javac	33 – 42	2.9	0.2	3	8.0	60.9	Java compiler
	mtrt	19 – 22	8.6	4.9	97	57.3	0.7	Multi-threaded ray-tracer
	raytrace	14 – 17	2.7	1.7	58	63.0	0.1	3D scene renderer
	volano	31 – 34	46.6	16.2	233	34.8	0.3	Multi-user chat server
	psjbb	119 – 125	25.4	12.5	70	49.3	3.3	Three-tier database system

Table 2. Benchmark baseline statistics obtained using HS with YP disabled. Col. 2 is the heap size range (we use 4 heap sizes across this range for each benchmark; heap size includes all generations). Cols. 3–6 show execution time, total GC time, total number of GCs, and the percentage of execution time that is consumed by GC (all for minimum heap sizes). Col. 7 shows the average GC yield across heap sizes as a percentage of the old generation size. The final column provides a brief description of the benchmark functionality. Highlighted entries are multi-threaded server programs.

Bench-mark	Average size [byte]			Minimum size [byte]			Maximum size [byte]			4KB page coverage [%]		
	CP	HS	MC	CP	HS	MC	CP	HS	MC	CP	HS	MC
beautyj	2.0K	2.2K	468.9	24.9	243.3	16.0	137.0K	119.6K	403.1K	44.1	43.2	81.3
findbugs	4.4K	4.4K	4.9K	16.0	16.0	16.0	1.5M	1.7M	2.0M	92.5	93.3	93.7
jaranalyzer	1.6K	1.7K	1.8K	16.0	16.0	16.0	8.8K	28.8K	13.0K	16.7	22.1	31.4
javaguard	430.1	557.9	384.1	28.1	26.4	16.0	7.7K	9.0K	44.0K	16.0	16.1	51.6
jdepend	1.7K	32.5K	245.3	1.3K	2.0K	16.0	7.9K	311.3K	66.7K	18.2	24.4	27.3
chart	77.6K	72.6K	45.7K	15.3	14.7	16.0	2.9M	3.6M	6.5M	99.4	99.4	99.4
fop	1.4K	1.8K	603.0	91.9	88.2	16.0	56.2K	92.5K	147.8K	26.0	29.0	58.3
hsqldb	5.0K	119.5	109.1	16.2	15.8	16.0	1.4M	27.7K	193.5K	22.5	22.6	75.7
pmd	126.6K	187.4K	17.3K	14.8	10.3	11.0	4.5M	6.6M	5.1M	99.3	99.5	98.7
xalan	46.8K	46.4K	127.9K	15.7	15.9	14.5	3.4M	4.0M	14.3M	99.5	99.4	99.8
compress	5.9M	2.9M	5.1M	19.2	25.0	16.0	10.5M	9.2M	13.2M	100.0	100.0	100.0
javac	13.2K	12.9K	8.6K	16.0	16.0	16.0	2.8M	3.1M	4.9M	94.3	95.7	94.7
mtrt	1.9K	2.3K	206.0	120.5	506.5	16.0	186.8K	69.7K	92.9K	11.6	6.8	31.1
raytrace	114.9	579.8	90.8	26.8	544.5	16.0	986.2	808.2	9.5K	0.4	0.4	13.2
volano	486.8	482.7	157.3	48.5	102.7	16.0	8.0K	7.3K	44.1K	12.4	12.6	35.5
psjbb	3.1K	3.0K	997.0	95.5	89.0	16.0	204.5K	329.1K	1.3M	41.0	46.2	59.5
average	397.4K	208.4K	341.0K	116.6	238.8	15.6	1.7M	1.8M	3.0M	49.6	50.7	65.7

Table 3. Dead space clustering statistics. For each benchmark, we report average dead cluster size (Cols. 2–4), minimum dead cluster size (Cols. 5–7), maximum dead cluster size (Cols. 8–10), and the percentage of dead space fully covered by 4KB pages (Cols. 11–13). All these are average values across GCs and heap sizes obtained with YP disabled. K is 1024 bytes, M is 1024K.

holds for both client- and server-side Java applications and is stable across inputs and heap sizes.

5.3 GC Yield

Column 7 in Table 2 shows the average GC yield for each benchmark. This value is the percentage of reclaimable space in the old generation on average across heap sizes and indicates how effective the GC is on average at reclaiming dead space. Across these programs, 9 have unproductive GCs (yield below 5%). In the remaining 7 benchmarks, the GCs are mostly productive (yield above 23%). We have also observed that the first full collection for all programs is typically productive even if a particular benchmark has a low GC yield on average.

5.4 Prediction Accuracy and Cost

We evaluate the prediction error of YP relative to the total heap size as well as relative to the old generation size. Specifically, if the exact amount of reclaimable space is x bytes and the predictor estimates that as y bytes, we compute the prediction error as $|x - y|/size$, where $size$ is heap or generation size. We measure relative error (as opposed to absolute error) because GC yield itself is typically expressed and used in practice as a percentage.

Baseline	Old Generation Size						Heap Size						Maximum Prediction Cost [%]		
Skip Threshold	0%			5%			0%			5%					
Benchmark	CP	HS	MC	CP	HS	MC	CP	HS	MC	CP	HS	MC	CP	HS	MC
beautyj	1.6	1.0	1.0	7.1	7.3	6.2	1.0	0.6	0.6	4.4	4.6	3.9	0.0	6.0	5.2
findbugs	7.1	5.4	5.8	6.7	5.5	5.9	4.6	3.2	3.9	4.3	3.3	4.0	3.8	2.8	4.4
jaranalyzer	9.1	8.8	11.5	9.3	8.5	11.5	1.8	1.8	3.0	1.9	1.7	3.0	1.2	1.6	1.8
javaguard	1.8	2.2	2.1	7.4	7.8	8.3	0.5	0.6	0.6	1.9	2.0	2.2	2.0	2.8	1.7
jdepend	0.6	0.3	0.3	6.6	6.6	6.7	0.3	0.2	0.2	3.2	3.2	3.3	0.9	1.3	1.3
chart	8.6	8.1	7.8	9.3	8.2	7.7	4.7	4.0	4.3	5.2	4.1	4.3	2.8	2.3	2.0
fop	1.7	1.4	1.2	7.2	7.1	7.4	0.4	0.4	0.3	1.7	1.8	1.9	2.4	1.5	1.8
hsqldb	0.4	0.4	0.4	8.7	5.8	7.0	0.3	0.3	0.3	5.8	3.9	4.7	9.5	9.5	4.2
pmd	5.1	12.4	7.8	5.6	11.8	7.0	2.8	6.7	4.5	3.0	6.4	4.1	9.1	6.5	5.2
xalan	2.7	5.7	3.0	2.7	5.5	3.3	1.5	3.3	1.8	1.6	3.2	2.0	3.2	4.0	6.3
compress	4.8	7.4	4.2	4.4	7.5	4.3	2.6	4.0	2.2	2.4	4.1	2.3	2.7	1.3	1.9
javac	4.1	7.0	4.6	3.9	7.0	5.5	2.1	3.5	2.6	2.0	3.5	3.1	3.6	3.5	5.3
mtrt	0.2	0.2	0.3	7.3	8.8	7.9	0.1	0.1	0.1	2.2	2.7	2.4	2.1	-0.4	3.8
raytrace	0.1	0.1	0.3	7.1	15.8	7.6	0.0	0.0	0.1	1.5	3.2	1.6	1.2	1.5	2.2
volano	0.6	0.3	0.8	4.8	7.5	9.3	0.3	0.1	0.4	2.3	3.5	4.5	3.6	3.3	3.7
psjbb	2.8	2.8	4.0	6.3	5.9	7.3	1.9	1.9	2.8	4.3	4.1	5.2	8.4	7.9	8.1
average	3.2	4.0	3.4	6.5	7.9	7.1	1.6	1.9	1.7	3.0	3.5	3.3	3.5	3.5	3.7

Table 4. Prediction error and cost. In Cols. 2–13 we report average yield prediction error, across the heap sizes, relative to the old generation size (Cols. 2–7) and heap size (Cols. 8–13). We present the results for two values of the GC skip threshold: 0% and 5% and three compactors: CP (Compressor), HS (HotSpot), and MC (Mapping Collector), each run with YP enabled and the young-old ratio set to 1%. The final 3 columns show the YP time overhead (maximum across the heap sizes).

We summarize the accuracy results in Table 4, which contains data averaged across the heap sizes. For each benchmark, we report prediction error for the 0% and 5% skip threshold, relative to the old generation size and heap size. The young-old ratio is 1%. The results for the 0% threshold (when no GC is skipped) lend insight into prediction accuracy unaffected by avoided GCs which is important in benchmarks whose GCs are mostly productive.

Across benchmarks and compactors, average error is below 4% (for the 0% threshold) and below 8% (for the 5% threshold) relative to the old generation size. This corresponds to 2% and 4% relative to the heap size. We investigate accuracy for other thresholds in Section 5.6. Accuracy is worse for the 5% threshold because GC skipping increases fragmentation in the heap.

Figure 1 shows detailed accuracy plots, across heap sizes, for selected benchmarks and the 5% threshold. We report average prediction error (data points) and standard deviation (error bars) from 5 measurements relative to the old generation size. The graphs show that accuracy also varies across the heap sizes.

To implement yield prediction, we employ a polling thread in the MRE that periodically samples the hardware page protection bits through an OS kernel module. During each GC, the MRE also executes the YP algorithm (cf. Section 3). Both of these operations can impose a performance penalty. The final three columns in Table 4 compare the execution times with and without prediction for CP, HS, and MC, to evaluate this overhead. With prediction on, we set the skip threshold to 0%. Thus, we do not skip any collections, and we isolate the performance penalty incurred by YP. That is, for each GC, we do complete prediction and collection

work in addition to the polling thread running concurrently. We report the maximum overhead as the percent increase in total execution time, across the heap sizes for CP, HS, and MC. This overhead is below 4% on average. Server-side benchmarks (e.g., hsqldb and psjbb) have the highest overhead as they fully utilize both CPU cores and the polling thread needs to preempt the application threads.

### 5.5 Impact on Applications

In this subsection, we focus on the eight benchmarks with low GC yields, i.e., those below 5% in Column 7 in Table 2. In the remaining programs, most collections cannot be skipped (as they are productive) and YP affects performance only marginally (maximum overhead is 3% on average for these programs). We set the GC skip threshold to 5% and the young-old ratio to 1%.

We first evaluate the impact that YP has on overall execution time by comparing the benchmark performance when prediction (and GC skipping) is enabled and disabled. In Table 5 (Cols. 2–4), we show the application throughput improvement for minimum heap sizes for each compactor. On average, across benchmarks, we observe significant improvements in execution time: e.g., reductions of 59% for CP, 47% for HS, and 44% for MC, on average.

Cols. 5–7 in Table 5 show the percentage of GCs eliminated (the skip rate) on average for each program across heap sizes. The skip rate varies between 64% and 87%, and has an average of 75% for HS and MC, and an average of 77% for CP; YP is able to avoid most GCs in these programs.

Since YP eliminates unproductive GCs, it thereby increases minimum mutator utilization [5] and program performance. By doing so, YP also reduces the number of
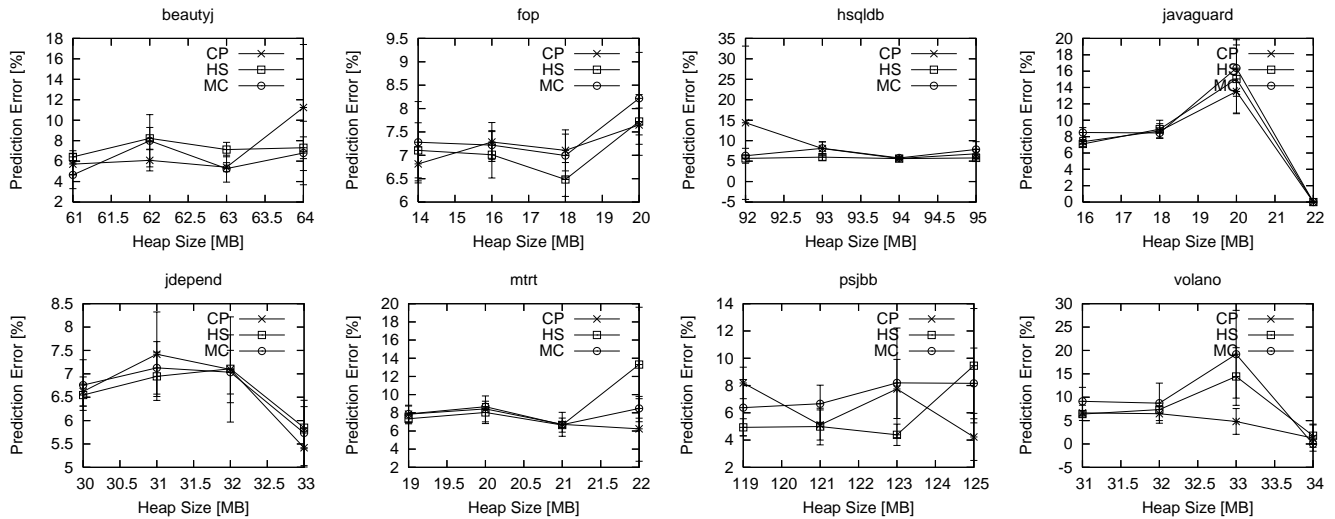
Figure 1. Prediction error relative to the old generation size across heap sizes for all compactors and 8 benchmarks (those with the most unproductive GCs). We report average and standard deviation (error bars) from 5 runs. Yield prediction is turned on, the GC skip threshold is 5%, and the young-old ratio is 1%.

Benchmark	Execution Time Reduction [%]			GC Skip Rate [%]			Maximum Pause Time Reduction [%]			Space Overhead [%]					
										Vs. Old Generation			Vs. Heap		
	CP	HS	MC	CP	HS	MC	CP	HS	MC	CP	HS	MC	CP	HS	MC
beautyj	84.8	79.3	74.1	82.9	81.0	81.0	-3.1	1.1	1.3	12.7	11.6	6.1	7.7	7.4	3.9
javaguard	47.5	42.0	31.7	66.7	69.1	69.7	-4.5	-3.2	0.0	23.4	22.0	20.5	5.8	5.5	5.4
jdepend	50.6	41.6	35.6	77.3	76.8	80.0	-1.2	-5.3	6.9	11.1	12.7	11.2	5.4	6.2	5.6
fop	43.9	37.7	30.5	65.9	68.6	67.4	-5.4	0.0	8.4	17.3	24.4	17.5	4.1	6.3	4.2
hsqldb	58.4	13.9	45.1	82.2	82.6	83.3	4.5	-2.1	0.1	5.1	9.9	4.9	3.4	6.7	3.3
mtrt	86.4	83.8	82.3	73.6	72.4	72.1	-20.1	2.8	3.3	20.9	20.5	19.9	6.3	6.1	6.0
volano	37.6	33.9	21.5	87.0	82.4	85.2	15.0	-0.7	27.8	9.2	8.6	8.9	4.4	4.1	4.4
psjbb	59.0	42.3	34.3	77.1	64.7	64.0	6.2	-3.6	-0.9	6.1	6.2	3.6	4.1	4.3	2.6
average	58.5	46.8	44.4	76.6	74.7	75.3	-1.1	-1.4	5.9	13.2	14.5	11.6	5.1	5.8	4.4

Table 5. Statistics for all compactors obtained for yield prediction turned on, the GC skip threshold of 5%, the young-old ratio of 1%, and for minimum heap sizes. Cols. 2–4 show execution time reduction due to YP. Next, in Cols. 5–7 we report the percentage of skipped (unproductive) GCs. Reduction in maximum GC pause times is shown in Cols. 8–10. The last 6 columns present space overhead imposed by GC skipping in YP, relative to the old generation size and heap size.

pauses an application experiences and increases the intervals between pauses. In the Cols. 8–10 in Table 5, we report the impact that YP has on maximum pause times. YP tends to increase pause times since when multiple GC are skipped, the heap size becomes larger, and the collection that is finally performed imposes a longer pause (while being more productive). Occasionally, however, YP skips an expensive compaction with the net effect of reducing the maximum pause time. On average, in CP and HS, YP increases maximum pauses by 1%, and in MC, it reduces them by 6%.

The trade-off that YP makes to achieve these performance gains is in predictor overhead (reported previously as below 4%) and in heap space. Cols. 11–16 in Table 5 show the space overhead that YP imposes for each compactor as a percentage of the old generation size and heap size. Each skipped collection creates a temporary space overhead in the

heap that is reduced or eliminated by the next conventional GC. This overhead results from skipping potentially multiple consecutive GCs. Relative to the old generation size the overhead is below 15%. The overhead does not exceed 6% relative to the total heap size.

We next present application throughput without (Figure 2) and with (Figure 3) YP and GC skipping. Each figure shows per-benchmark plots, each with 3 performance curves that correspond to CP, HS, and MC, respectively. We report average execution time (data points) and standard deviation (error bars) computed from 5 runs for each heap size. From the differences between the graphs in these two figures, we observe that YP consistently outperforms conventional GC for all three compactors across heap sizes.

Note that YP outperforms a system employing heap over-provisioning to run GC less often. Giving more space to HS,
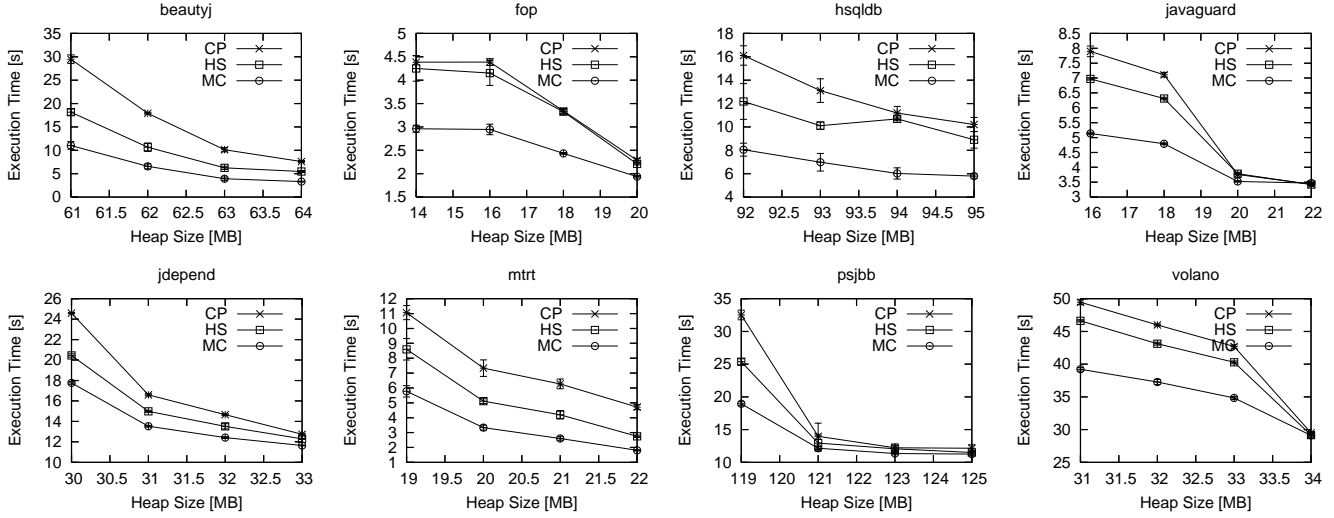
Figure 2. Benchmark execution times across heap sizes for all compactors. We report average and standard deviation (error bars) from 5 runs. Yield prediction is turned off.
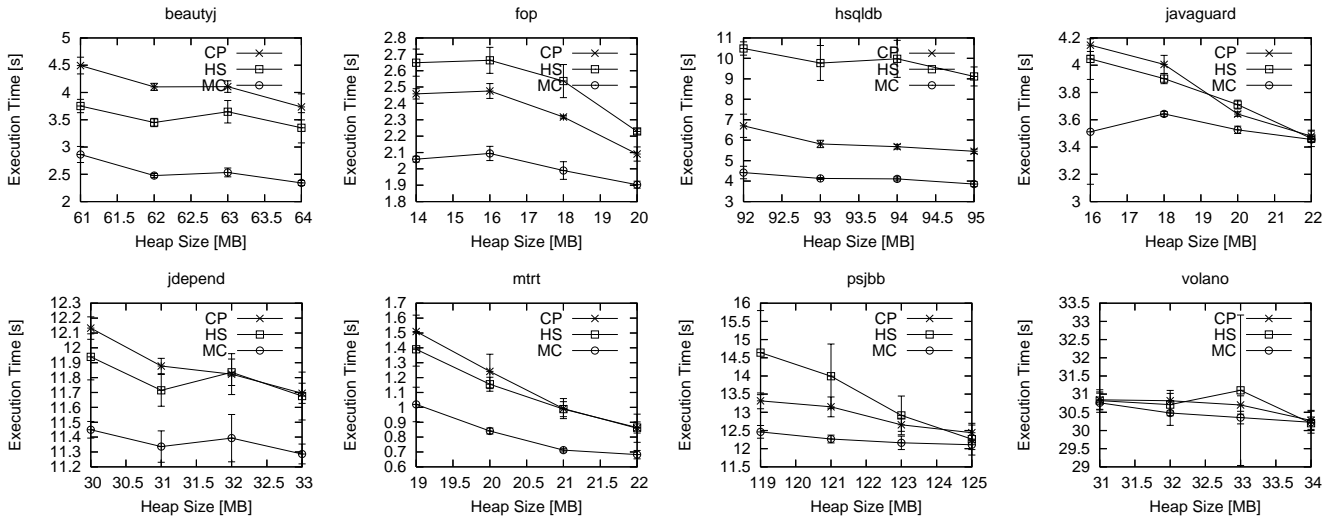
Figure 3. Benchmark execution times across heap sizes for all compactors. We report average and standard deviation (error bars) from 5 runs. Yield prediction is turned on, the GC skip threshold is 5% and the young-old ratio is 1%.

MC, and CP (as much as YP space overhead) does not lead to better execution times than YP obtains.

5.6 Other Parameter Values

We have also evaluated YP for the GC skip threshold of 3% and 10% to understand better how this parameter impacts application performance. Table 6 summarizes the results and compares them with the ones obtained for 5%. Overall, as the threshold increases, the prediction accuracy decreases, the space overhead increases, the skip rate increases, and we observe better performance gains. Thus, skip threshold selection is a space/time trade-off.

We have also investigated different values of the young-old ratio, a YP parameter which determines what proportion of the window between two subsequent GCs is considered young. The detailed YP evaluation we have presented thus far is for the 1% dead-young ratio. We have found this value to result in optimal prediction accuracy (we have checked 1%, 2%, 5%, 10%, 20%, 50%, and 90%). Figure 4 shows the impact of the young-old ratio on prediction error, space overhead, maximum pause time reduction, and execution time. Accuracy monotonically decreases when the young-old ratio increases (prediction error increases from 7% to 14%). This is because in a steady-state execution phase, programs allocate mostly short-lived objects. The remaining metrics are not overly sensitive to the young-old ratio. This is mostly because the prediction error never exceeds 16% for the ratios that we checked. Nonetheless, execution time reduction is worse for higher values of the young-old ratio.

Skip Threshold	3%			5%			10%			Average		
Compactor	CP	HS	MC	CP	HS	MC	CP	HS	MC	CP	HS	MC
Prediction Error (Vs. Old Generation)	5.4	6.2	5.7	6.5	7.9	7.1	8.1	10.3	8.8	6.7	8.1	7.2
Prediction Error (Vs. Heap)	2.5	2.8	2.7	3.0	3.5	3.3	3.9	4.8	4.1	3.1	3.7	3.4
Space Overhead (Vs. Old Generation)	12.9	14.1	10.8	13.2	14.5	11.6	15.9	16.1	14.3	14.0	14.9	12.2
Space Overhead (Vs. Heap)	5.0	5.6	4.0	5.1	5.8	4.4	6.3	6.6	5.8	5.5	6.0	4.7
Collection Skip Rate	70.6	71.8	70.1	76.6	74.7	75.3	83.2	80.5	80.6	76.8	75.7	75.3
Execution Time Reduction	55.7	41.8	42.0	58.5	46.8	44.4	62.3	53.5	47.2	58.8	47.4	44.5
Maximum Pause Time Reduction	-3.0	-0.7	4.2	-1.1	-1.4	5.9	0.7	0.3	10.6	-1.1	-0.6	6.9

Table 6. YP statistics for different GC skip thresholds (3%, 5%, and 10%) for each compactor (CP, HS, and MC). We report average values across benchmarks and heap sizes. Young-old ratio is 1%. All values are percentages.
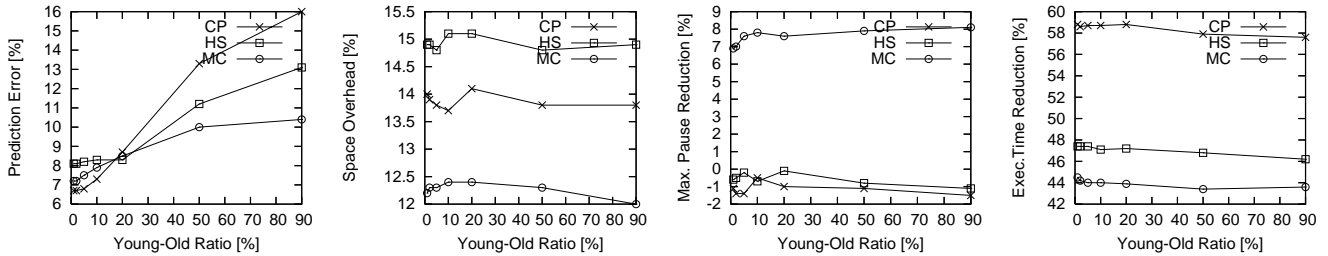
Figure 4. Impact of the young-old ratio on prediction error, space overhead, maximum pause time reduction, and execution time reduction. The values of the young-old ratio we use are: 1%,2%,5%,10%,20%,50%, and 90%. For each compactor (CP, HS, and MC) we report average values obtained across the three skip thresholds (3%, 5%, and 10%).

6. Related Work

Static and dynamic prediction in the context of automatic memory management includes object lifetime prediction [2, 18, 8, 20, 26, 39, 28, 23] as well as heap size prediction [37, 13, 3, 36, 38]. In contrast, YP focuses on yield prediction. No prior work to our knowledge exploits page reference bits to predict GC yield accurately.

Like YP, MicroPhase [35] strives to improve the GC triggering mechanism to maximize the GC yield. MicroPhase recognizes phase boundaries and proactively invokes GC during phase transitions when many objects are expected to die. The system cooperates with the OS kernel to implement efficient profiling. In contrast, YP uses reference bits to predict GC yield and is therefore simpler while extracting the phase behavior implicitly.

Garbage collection hints (GCH) [4] is a profile-directed method for guiding garbage collection. GCH uses off-line profiling to identify favorable collection points in the program code where GC dynamically chooses between nursery and full-heap collections based on an analytical garbage collector cost-benefit model. In contrast, YP does not use off-line profiling and leverages hardware to make yield prediction.

The systems below are related to YP because they often actively interact with hardware and operating systems. However, they either do not leverage the mechanism of RR bits or do not implement yield prediction.

The Pauseless GC [6] is a parallel/concurrent compactor that avoids pauses through hardware read barriers, fast user-mode trap handlers, an additional intermediate TLB privilege level, and fast cooperative preemption via interrupts. The compactor consists of three phases, called mark, relocate, and remap. The mark phase periodically refreshes the liveness bitmap and computes page occupancy statistics used to determine which pages are most empty. The relocate phase finds pages that contain few live objects, evacuates live data from those pages, and frees the underlying physical memory. Pages with no live data are unmapped. Evacuated virtual pages containing live objects are protected to trigger traps upon access. Mutators using stale pointers raise traps which update pointers to refer to new object locations. The remap phase traverses the object graph executing a read barrier against each pointer to ensure the completeness of lazy pointer forwarding.

Numerous collectors leverage virtual memory operations. The Compressor (Section 2.2) employs both page mapping and protection. The Mapping Collector (Section 2.2) remaps free space in the address space. MarkCopy [27] reduces the memory footprint of a copying collector through on-the-fly (un)mapping of the copied pages.

Some collectors [13, 37, 15, 38, 36, 14] cooperate with the OS virtual memory manager to reduce the collector-induced paging. The Bookmarking Collector [15] records summary information about outgoing pointers from evicted pages to avoid accessing non-resident pages during compaction. CRAMM [37] and IV heap sizing [13] use VM paging behavior to predict and set dynamically the most-suitable, application-specific, heap size that adapts to chang-

ing memory pressure and avoid paging. The system described in [38] dynamically finds the optimal heap size by exploiting phase behavior to balance the GC frequency and collection cost as well as minimize the impact of page faults on performance. Many concurrent collectors also exploit virtual memory support [10, 22, 6, 25], which facilitates mutator conflict detection and exploitation of cache locality [22].

7. Conclusions

We contribute YP, a GC yield predictor that uses virtual page reference bits to accurately estimate the amount of reclaimable space in the heap. We incorporate YP into three state-of-the-art parallel compactors to verify its applicability to canonical heap layouts used by extant collectors. YP is simple and does not require changing the GC algorithm (only its triggering mechanism). YP enables better dynamic control over the space/time trade-off in MREs. We empirically evaluate YP using 3 compactors and 16 programs and find that YP consistently provides good accuracy while imposing low time overhead. In applications with many unproductive GCs, YP significantly improves performance (by 44–59% on average) by skipping most GCs and incurring modest space overhead.

Acknowledgments

References

[1] A. W. Appel and K. Li. Virtual memory primitives for user programs. *ACM SIGPLAN Notices*, 26(4), 1991.

[2] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. PLDI'93.

[3] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric prediction of heap memory requirements. ISMM'08.

[4] D. Buytaert, K. Venstermans, L. Eeckhout, and K. De Bosschere. GCH: Hints for triggering garbage collections. *TH-PEAC*, 1(1), 2007.

[5] P. Cheng and G. Blelloch. A parallel, real-time garbage collector. PLDI'01.

[6] C. Click, G. Tene, and M. Wolf. The pauseless GC algorithm. VEE'05.

[7] J. Cohen and A. Nicolau. Comparison of compacting algorithms for garbage collection. *TOPLAS*, 5(4), 1983.

[8] D. Cohn and S. Singh. Predicting lifetimes in dynamically allocated memory. ANIPS'97.

[9] The DaCapo Benchmark Suite. http://dacapobench.org.

[10] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. ISMM'04.

[11] C. Flood, D. Detlefs, N. Shavit, and C. Zhang. Parallel garbage collection for shared memory multiprocessors. JVM'01.

[12] Open Source Software in Java. http://java-source.net.

[13] C. Grzegorczyk, S. Soman, C. Krintz, and R. Wolski. Isla Vista heap sizing: Using feedback to avoid paging. CGO'07.

[14] M. Hertz, Y. Feng, and E. Berger. Page-level cooperative garbage collection. Univ. Massachusetts, Tech. Report, 2004.

[15] M. Hertz, Y. Feng, and E. D. Berger. Garbage collection without paging. PLDI'05.

[16] A. L. Hosking and J. E. B. Moss. Protection traps and alternatives for memory management of an object-oriented language. SOSP'93.

[17] HotSpot Virtual Machine Garbage Collection. http://java.sun.com/javase/technologies/hotspot/gc.

[18] H. Inoue, D. Stefanović, and S. Forrest. Object lifetime prediction in Java. Univ. New Mexico, Tech. Report, 2003.

[19] R. Jones. Dynamic memory management: Challenges for today and tomorrow. ILC'07.

[20] R. Jones and C. Ryder. Garbage collection should be lifetime aware. ICOOOLPS'06.

[21] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996.

[22] H. Kermany and E. Petrank. The Compressor: Concurrent, incremental and parallel compaction. PLDI'06.

[23] S. Marion, R. Jones, and C. Ryder. Decrypting the Java gene pool. ISMM'07.

[24] Open Source J2SE. http://openjdk.java.net.

[25] Y. Ossia, O. Ben-Yitzhak, and M. Segal. Mostly concurrent compaction for mark-sweep GC. ISMM'04.

[26] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. POPL'88.

[27] N. Sachindran and E. Moss. MarkCopy: Fast copying GC with less space overhead. OOPSLA'03.

[28] M. L. Seidl and B. Zorn. Low cost methods for predicting heap object behavior. WFDO'99.

[29] The SPEC Benchmarks. http://www.spec.org.

[30] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5), 1984.

[31] The VolanoMark Benchmark. http://www.volano.com.

[32] M. Wegiel and C. Krintz. The Mapping Collector: Virtual memory support for generational, parallel, and concurrent compaction. ASPLOS'08.

[33] P. R. Wilson. Uniprocessor garbage collection techniques. Univ. Texas, Tech. Report, 1994.

[34] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. IWMM'95.

[35] F. Xian, W. Srisa-an, and H. Jiang. Microphase: an approach to proactively invoking garbage collection for improved

performance. OOPSLA'07.

[36] T. Yang, E. D. Berger, M. Hertz, S. F. Kaplan, and J. E. B. Moss. Autonomic heap sizing: Taking real memory into account. ISMM'04.

[37] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: Virtual memory support for garbage-collected applications. OSDI'06.

[38] C. Zhang, K. Kelsey, X. Shen, C. Ding, M. Hertz, and M. Ogihara. Program-level adaptive memory management. ISMM'06.

[39] B. Zorn and M. Seidl. Segregating heap objects by reference behavior and lifetime. ASPLOS'98.