MicroPhase: An Approach to Proactively Invoking Garbage Collection for Improved Performance

Feng Xian, Witawas Srisa-an, and Hong Jiang

Department of Computer Science & Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588-0115
{fxian,witty,jiang}@cse.unl.edu

Abstract

To date, the most commonly used criterion for invoking garbage collection (GC) is based on heap usage; that is, garbage collection is invoked when the heap or an area inside the heap is full. This approach can suffer from two performance shortcomings: untimely garbage collection invocations and large volumes of surviving objects. In this work, we explore a new GC triggering approach called *MicroPhase* that exploits two observations: (i) allocation requests occur in phases and (ii) phase boundaries coincide with times when most objects also die. Thus, proactively invoking garbage collection at these phase boundaries can yield high efficiency. We extended the HotSpot virtual machine from Sun Microsystems to support MicroPhase and conducted experiments using 20 benchmarks. The experimental results indicate that our technique can reduce the GC times in 19 applications. The differences in GC overhead range from an increase of 1% to a decrease of 26% when the heap is set to twice the maximum live-size. As a result, MicroPhase can improve the overall performance of 13 benchmarks. The performance differences range from a degradation of 2.5% to an improvement of 14%.

Categories and Subject Descriptors D.3.4 [*Programming Language*]: Processors—Memory management (garbage collection)

General Terms Experimentation, Languages, Performance

1. Introduction

Garbage collection (GC) is a process that automatically reclaims dynamically allocated memory. The benefits of garbage collection include elimination of memory leaks and dangling pointers. It also promotes cleaner code as memory management concerns are not interleaved with execution logics. Categorically, there are two approaches to garbage collection: reference counting and tracing. The latter is the main focus of this paper.

Tracing garbage collectors (e.g. copying generational, mark-sweep, mark-compact) often require two phases: (i) identification of reachable objects, and (ii) reclamation of dead objects *en masse*. Many implementations of these collectors are stop-the-world, which means that all execution threads (except for the garbage collection thread) must be stopped during garbage collection. To date, the most commonly used GC invocation criterion is space-based (we refer to a collector using the space-based criteria as the space-based approach); that is, when the volume of objects in a heap reaches a certain predefined threshold, garbage collection is invoked. It has been well documented that this invocation criterion leads to the following shortcomings:

1. *Untimely invocation*—Garbage collection is usually invoked when applications need to create a large number of objects. In such a scenario, garbage collection often prevents an application from allocating objects freely at its most critical times [18]. Such untimely invocations can greatly affect the usability of mission critical applications.
2. *Large volume of surviving objects*—A consequence of untimely GC invocations is that a large number of objects created just prior to a GC invocation will not have died (we refer to them as surviving new objects or SNOs); thus, spending any effort collecting SNOs is wasteful. In copying generational collectors, the promotion of SNOs reduces the minor collection efficiency and results in a higher frequency of full collection invocations. Two approaches have been introduced to overcome this shortcoming by concentrating the collection effort on older objects and avoiding collecting newly created objects [4, 30]. However, these approaches do not address the

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.

fundamental issue of untimely invocations, which is the root cause of this shortcoming.

This work: We introduce *MicroPhase*, a generalized GC triggering mechanism that *considers efficiency instead of heap usage* as an invocation criterion. MicroPhase leverages two key observations:

1. Allocation requests and reductions of live objects often occur in phases [7, 10, 24].
2. A portion of execution with light or no allocation activities (which appears as an allocation pause) mostly coincides with the time when most objects die [21]. We illustrate this observation in Section 2.

We modified the generational garbage collector in the HotSpot virtual machine to exploit these two observations by monitoring the object allocation behavior and invoking the corresponding garbage collector (minor or full) when an allocation pause is detected. In doing so, MicroPhase avoids invoking GC during intense allocation phases, resulting in a major decrease in the volume of SNOs.

We then compared the performance of MicroPhase against that of the space-based approach. Our experiment is conducted using twenty benchmarks in a setting where the heap size is set to be twice the maximum volume of live objects or *live-size*. This setting allows most applications to have reasonable GC overheads. However, three benchmarks indicate that such a setting makes the heap extremely tight during their executions (i.e. the GC overhead is more than 50% of execution time).

The experimental results show that MicroPhase is more effective than the space-based approach, and thereby, can reduce the garbage collection times of 19 benchmarks. The differences in GC times range from an increase of 1% to a decrease of 26%. A study of *minimum mutator utilization* (MMU) also shows that MicroPhase is less disruptive to mutator execution in 13 benchmarks when the window sizes are small. As a result, MicroPhase can improve the overall performance of 13 benchmarks. The overall performance differences range from a degradation of 2.5% to an improvement of 14% under a tight heap environment.

Outline: The remainder of this paper is organized as follows. Section 2 describes the current issues with space-based triggering and suggests a possible solution based on two validated hypotheses. Section 3 provides an overview of MicroPhase and the implementation details. Section 4 and 5 outline the experimental methodology, evaluate the overhead and accuracy of MicroPhase, and report the experimental results comparing MicroPhase with the HotSpot collector. Section 6 discusses the implications of our work and future challenges. Section 7 summarizes existing efforts in the field related to this work. Section 8 concludes this paper.

2. Motivation

The motivation of our work is to explore an alternative approach to invoking garbage collection that does not solely rely on space usage, but instead relies on garbage collection efficiency. We conjecture that such an alternative exists due to the following reported observations and hypotheses:

1. A study by Dieckmann and Hölzle clearly shows that in many applications, the volume of live objects or *live-size* increases and decreases in phases [10]. Similar behavior is displayed in the work by Shaham *et al.* [24]. In addition, work by Wilson and Moher [33, 34] shows that GC can be opportunistically invoked during and right after long compute-bound phases, when most objects die. These findings imply that it is possible to efficiently collect objects if each GC invocation occurs at a location with a small volume of live objects.
2. Stefanović *et al.* observe that GC is often invoked after a new phase has begun; thus, many objects created in the new phase do not have time to die [30]. This observation is confirmed by Chen *et al.* when they report that the space-based approach often invokes GC after the "phase boundary especially if the new phase starts by allocating many new objects" [8].

These two observations lead us to the following hypotheses:

2.1 Hypothesis 1: Allocation Phases Do Exist

An increase in the volume of live objects (as shown by [10]) indicates a period of intense allocation requests. A sudden decrease in the volume of live objects is likely an indication of periods with much less allocation intensity. Such behavior corresponds to the notion of phase boundaries as reported by [8, 33].

At this time, we wish to introduce two important terms that will be used throughout the paper: *execution pause* and *allocation pause*.

Definition: An *execution pause* is defined as *a time period in which a mutator thread is suspended*. Thus, there are no object allocation activities.

Definition: An *allocation pause* is defined as *a sufficiently long execution period with no allocation requests* made by a mutator thread.

For example, if a mutator thread is suspended due to an I/O request, the detected pause is an execution pause, which has *no effect on heap usage and liveness of objects* because the mutator thread is not active. Therefore, the execution pause is unlikely to be an efficient GC invocation point. On the other hand, an allocation pause occurs when a thread makes two consecutive allocation requests (with no execution interruption in between), and the interval between these two requests is sufficiently long. Within this pause, previously created objects are manipulated and should be dead by

Benchmark	Description	Pause length (bytecodes)			Number of pauses
		average	min	max	
compress	A utility to compress/uncompress large files.	206513	20	8035180	20
db	Performs DB functions on memory resident database.	700	19	2192020	441
jack	A Java parser generator with lexical analyzers.	154	20	2666	11796
javac	JDK 1.0.2 Java compiler.	288	16	24310	12573
jess	A Java expert shell system based on NASA's CLIPS expert shell system.	204	19	7616	13101
mpegaudio	A mpeg-3 audio stream decoder.	236	10	2666	32
mtrt	A dual-threaded program that ray traces an image file.	300	10	1068	13985
raytrace	Works on a scene depicting a dinasaur.	263	23	1056	302
antlr	Parses several gramma files and generates a parser and lexical analyzer for each.	1075	13	234122	141
bloat	Performs a number of optimizations and analysis on Java bytecode files.	346	10	35765	8012
chart	Plots a number of complex line graphs and renders them as PDF.	288	12	34326	7441
eclipse	Executes some of non-GUI jdt performance tests for the Eclipse IDE.	322	12	234122	34012
fop	Formats an XSL-FO file and generates a PDF file.	652	10	12249	2470
hsqldb	Executes a number of transactions against a model of a banking application.	440	10	20077	565
jython	Interprets the pybench.	1145	10	719632	3934
luindex	Uses lucene to index a set of documents.	898	13	108033	739
lusearch	Uses lucene to do a text search of keywords over a corpus of data.	521	11	4800	916
pmd	Analyzes a set of Java classes for a range of source code problems.	5	10	240	2738
xalan	Transforms XML documents into HTML.	12	4	90	405
jbb2000	A Java program emulating a 3-tier system with emphasis on the middle tier.	616	0	12248	10012

Table 1. Allocation pauses occur in every application. However, the length and frequency of pauses can vary significantly among applications.

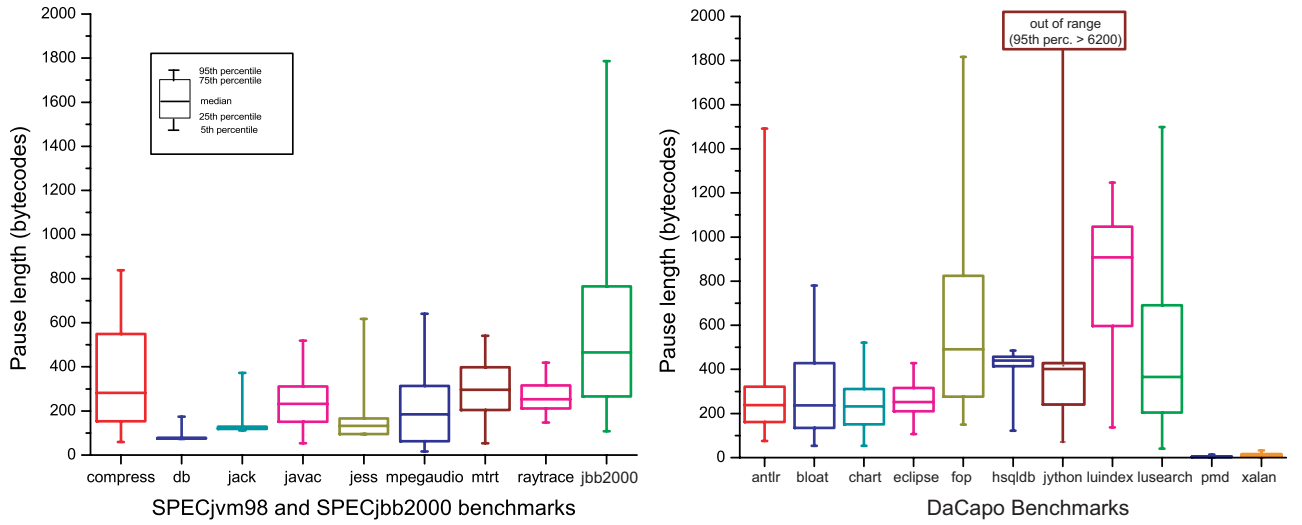


Figure 1. More characterization of allocation pauses in each benchmark

the end of the pause. One major challenge is to segregate the allocation pauses from the execution pauses. A detailed description of our approach to overcome this challenge is provided in Section 3.

We then empirically characterized allocation pauses in all benchmarks. Figure 1 illustrates the boxplot of pause length characteristic of each benchmark. Each boxplot can be interpreted as follows: the box contains the middle 50% of the data from the 75th percentile of the data set (represented by the upper edge of the box) to the 25th percentile (represented by the lower edge); the line in the box represents the median value of the data set; the short dash lines at both ends of the vertical line indicate the 5th percentile and 95th percentile of the data set. Table 1 also reports the mean, maximum, minimum, and number of occurrences of allocation pauses. The result shows that the intervals of the allocation pauses can vary from several bytecodes to a few million bytecodes. In addition, the frequency of these pauses significantly varies among applications. Nevertheless, the table and the figure clearly show that allocation pauses do exist in all applications.

2.2 Hypothesis 2: Strong Relation between Allocation Pauses and Live-Size Reductions

Our hypothesis is based on (i) an intuition that fewer allocation requests are likely the main reason for the decreasing volume of live objects, and (ii) the weak generational hypothesis, which states that most objects die young [17, 32]. Thus, the majority of objects created during an intense allocation phase should die soon after the phase is over.

We utilized the *Merlin Algorithm* [15] to obtain the live-size throughout the execution of every benchmark application. We then conducted experiments to obtain the heap usage of every application over time. We eliminated disruptions due to GC by setting the heap size for each application to be large enough to allow the application to execute without having to invoke GC. At the same time, limiting the maximum heap size to be smaller than the main memory capacity eliminates paging.

We then superimposed the live-size and the heap usage information over the execution time. Figure 2 illustrates the result of *jack*, a benchmark program in SPECjvm98. The figure on the left indicates that there are sixteen major live-size reduction events (appearing as saw tooth). The figure on the right shows a magnified version of the first live-size reduction event. Note that the allocated-size increases throughout the execution. However, there are also short *allocation pauses*, which appear as short horizontal lines in the allocated-size plot.

An interesting observation from the figure to the right is that the major live-size reduction occurs during the allocation pause. In fact, each of the allocation pauses in *jack* coincides with a live-size reduction event. Furthermore, our study indicates that over 90% of allocation pauses coincide with live-size reduction events in 18 out of 20 benchmarks. The magnitude of each live-size reduction event also varies from a few bytes to several megabytes in some applications (as shown in Table 2).

Benchmark	Relation between allocation pauses and live-size reductions (%)	Magnitude of a live-size reduction in a pause (KB)		
		average	min	max
compress	91	31	7	51
db	94	11	0.176	1172
jack	90	15	0.7	562
javac	91	4702	16	24310
jess	96	18	0.3	66
mpegaudio	81	31	7	51
mtrt	92	8	0.048	15
raytrace	86	1	0.048	74
antlr	96	40	0.072	905
bloat	95	19	0.032	1865
chart	96	26	0.07	2012
eclipse	98	102	0.1	34021
fop	96	21	0.072	7391
hsqldb	95	65	0.072	4913
jython	97	16	0.072	396
luindex	96	24	0.072	548
lusearch	93	17	0.032	72
pmd	98	40	0.072	267
xalan	95	6	0.024	347
jbb2000	96	6	0.04	24010

Table 2. Analysis results of correlation between allocation pauses and live-sizes.

However, we also found that many allocation pauses do not coincide with live-size reduction events. Upon further investigations, we discovered that there are many execution events that cause allocation pauses. For example, the "thin-lock" mechanism [2] used by HotSpot can cause an application to spend some of its execution time in busy-waiting loops. Each time a thread enters such a loop, it appears as an allocation pause, but it is not likely to correspond to a live-size reduction. Other execution events that appear as allocation pauses without live-size reductions include iteration of large arrays and initialization of large objects.

Summary: We have shown that our hypotheses are valid in all benchmarks. In the next section, we leverage these insights to construct mechanisms to detect these allocation pauses in order to invoke garbage collection in a timely manner.

3. Introducing MicroPhase

We implemented MicroPhase in *HotSpot*, the flagship Java Virtual Machine (JVM) from Sun Microsystems. HotSpot utilizes generational collector to manage the heap that is divided into three partitions: nursery, mature space and permanent space. The nursery is further subdivided into *eden*, *to* and *from* spaces. The generational collector in HotSpot works as follows. All objects are created in the nursery. When the nursery is full, *minor collection* is invoked to promote surviving objects to the mature space. When the mature

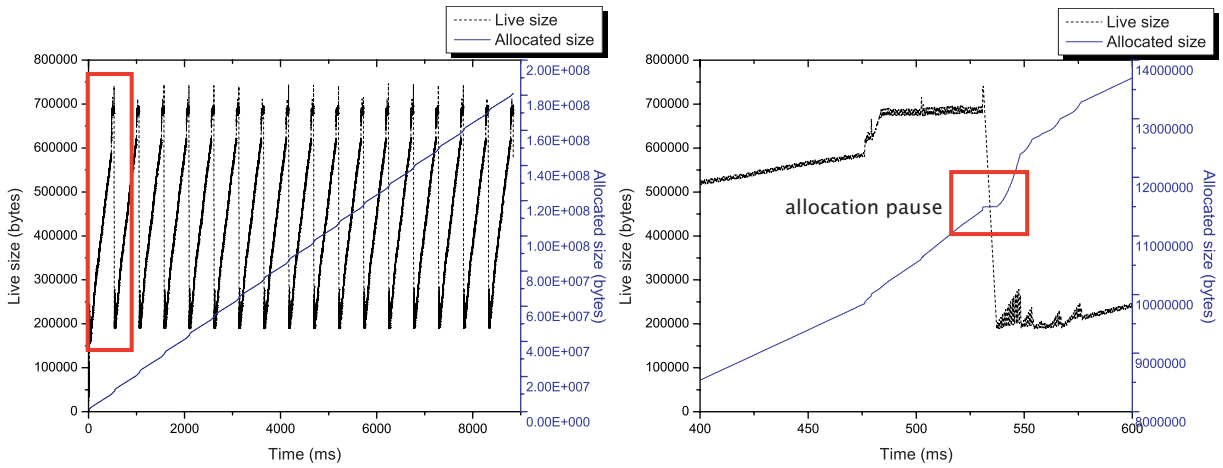


Figure 2. An illustration of a relation between pauses and live-size reductions in *jack*. The graph on the right is the magnified version of the first live-size reduction event in the graph on the left.

space is full, *full collection* is invoked to perform whole heap collection using mark-compact algorithm [17, 31]. Typically, the nursery is configured to be much smaller than the mature space. The permanent space is used to hold objects that live for the duration of the program; thereby, this area is not garbage collected.

We modified HotSpot to include two additional features, *allocation pause detection* (incorporated into the memory allocator) and *dynamic triggering mechanism* (incorporated into the garbage collector), which determines if minor or full collection should be invoked when an allocation pause is detected.

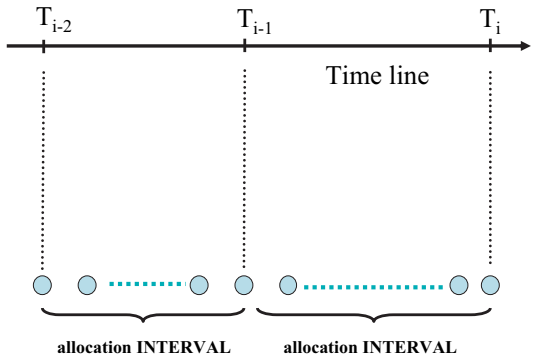


Figure 3. Detecting allocation pauses

3.1 Allocation Pause Detection

The basic notion of the proposed allocation pause detection is quite simple: measure the interval between two consecutive allocation requests by each thread, and if the interval is sufficiently long, consider it an allocation pause. While the notion is straightforward, the actual implementation and its application face two major challenges:

1. *Detect allocation pauses accurately and inexpensively.* Our technique must be able to segregate allocation pauses from execution pauses. In addition, each application may have a distinct allocation pause behavior. MicroPhase must be designed to work effectively in the presence of such variations.
2. *Decide when allocation pauses should be used as a GC triggering criterion.* Frequently occurring allocation pauses can cause GC to be invoked excessively. In order to allow flexible application of the technique, MicroPhase must support multiple modes of operation such as pure pause-based, pure space-based, and a combination of the two.

Allocation Pause Detection Algorithm. We modified the memory allocator in HotSpot to measure the number of CPU cycles needed to satisfy a given number allocation requests (*INTERVAL*). To distinguish between an allocation pause and an execution pause, we created a cooperative profiling environment where the virtual machine, the operating system, and the underlying processors work together to allow detailed profiling of thread execution [14].

Our technique utilizes a per-thread timer, an extension to the Linux kernel, to record the execution cycles that can be obtained through performance-monitoring instructions such as RDTSC in X86 based processors [25]. Each time a thread is scheduled, the value of the cycle counter register (*rdtsc* register in X86 processors) is recorded as $cycle_{start}$ in the thread structure ($task\_struct$ in Linux). When the thread is suspended, the new rdtsc value is read and stored in $cycle_{end}$. The difference between $cycle_{start}$ and $cycle_{end}$ is accumulated in $cycle_{total}$. The value of $cycle_{total}$ is accessible via a system call [22]. The profiling overhead is considered as part of the runtime overhead of MicroPhase. (A study of the runtime overhead is presented in Section

4.1.) With this mechanism, the time that a thread spent on blocking is filtered out.

For example, when *INTERVAL* is set to 2 (the minimum value), each measurement of an allocation pause is taken between two allocation requests. When an allocation request is satisfied, an associated time stamp (T) based on per-thread-timer is recorded for the purpose of calculating the allocation pause. Figure 3 illustrates our allocation pause detection mechanism. Basically, if $\frac{T_i - T_{i-1}}{T_{i-1} - T_{i-2}} > \delta$, where δ **is a predefined threshold indicating an unusually large ratio**, we can assume that the period between T_i and T_{i-1} contains an allocation pause. We used this technique instead of always calculating a pause between two allocation requests to *reduce the overhead of pause calculation and detection*.

```
(1) heapUsage ← Used heap / Heap capacity
(2) if heapUsage > hThreshold then
(3)     if a pause is detected then
(4)         Trigger GC
(5)     endif
        /* Heap has exhuasted but pause is not detected */
(6)     if space_available < requested_size then
(7)         Trigger GC
(8)     endif
(9) endif
```

Figure 4. A generalized algorithm to incorporate allocation pauses as an invocation criterion

If *INTERVAL* is too short (2 for example), the detection overhead will be high. On the other hand, if *INTERVAL* is too long, the system may miss allocation pauses. Similarly, if δ is set too large, smaller allocation pauses may not be detected. On the other hand, if δ is set too small, execution events such as iteration of large arrays or short busy-wait loops can also be detected as the allocation pauses. Our goal is to *identify INTERVAL* and δ *that yield good detection accuracy and low detection overhead across all benchmarks*. In Section 4, we report the results of our experiment to identify these two generalized values applicable to all benchmarks.

When should allocation pauses be used as a criterion? Once these allocation pauses are detected, should they be used as the sole criterion or a supplemental criterion for triggering GC? One option is to use these allocation pauses as the criterion in an *uncontrolled* or pure pause-based mode, meaning that GC is invoked whenever an allocation pause is detected. In this option, one possible shortcoming is excessive GC invocations if allocation pauses are detected frequently. The second option is to use these allocation pauses as a criterion in a *controlled* mode; i.e. invoking GC at each detected allocation pause only after the heap usage has already exceeded a predefined heap usage threshold or $hThreshold$. Prior to this point, the pause detection mechanism is not enabled.

The second option is more advantageous because it strikes a balance between higher efficiency in reclaiming objects and lower frequency of invoking GC. In both cases, the space-based technique is used as a back-up triggering mechanism in a scenario where allocation pauses are not detected prior to heap exhaustion. Figure 4 outlines the generalized algorithm of MicroPhase. An empirical study of these two modes is presented in the next section.

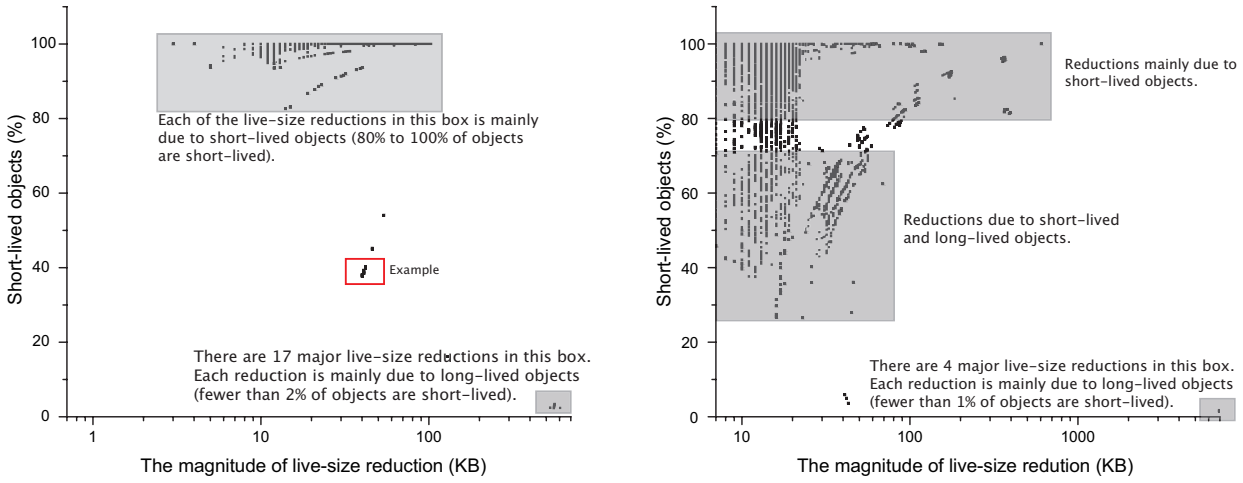
3.2 Triggering Mechanism

Our earlier study (see Section 2) has already shown that the volumes of live-size reduction can vary significantly within an application. Therefore, when MicroPhase is used in a generational collector, one refinement that must be made to the generalized algorithm (Figure 4, line 4) is predicting which area of the heap (nursery or mature) contains a larger volume of dead objects. This prediction allows MicroPhase to invoke the corresponding collector (i.e. minor or full) when an allocation pause is detected. As the first step toward building such a predictor, we conducted experiments to characterize the live-size behaviors.

Study of live-size behavior. For each live-size reduction, we measured the volume of dead objects as well as the ratio between dead objects that were short-lived and dead objects that were long-lived. We defined short-lived objects as objects with lifetimes of less than 20% of the maximum live-size [5]. Figure 5 shows our analyses of *jack* and *javac*, the two benchmarks with distinctive live-size characteristics.

Each dot in the graph represents a live-size reduction. The x-axis represents the reduction volume, and the y-axis indicates the percentage of dead objects that are short-lived. The analysis of *jack* (Figure 5a) indicates that over 98% of the live-size reductions are due to short-lived objects. This suggests that when a pause is detected, most of the dead objects should be in the nursery. On the contrary, the analysis of *javac* (Figure 5b) indicates that both short-lived and long-lived objects are responsible for the live-size reductions; therefore, it becomes less obvious whether minor or full GC should be invoked when a pause is detected.

Figure 5 also indicates that most of the largest live-size reductions are due to long-lived objects. For example, *javac* has four major live-size reductions. Over 99% of the dead objects in these four major reductions have very long lifespans. Similarly, 98% of the dead objects responsible for the seventeen major live-size reductions in *jack* are long-lived. In other words, these objects die in the mature space. The result of our analysis indicates that in most cases, minor collection should be invoked when a pause is detected. However,



(a) jack (b) javac

Figure 5. Investigation of the types of objects responsible for live-size reductions. Each dot represents a livesize reduction event. The x-axis indicates the volume of the reduction in KB, and the y-axis indicates the percentage of objects that are short-lived in a reduction. For example, each reduction event occurring in the box labeled "Example" has the reduction volume of about 40KB, and 40% of objects that die in the reduction event are short-lived.

higher efficiency can be obtained if we can predict instances when it is better to invoke full collection.

When to invoke full collection? Once the *nursery occupancy* reaches $hThreshold$, our triggering algorithm invokes minor collection each time an allocation pause is encountered. As part of each minor collection invocation, its collection efficiency is also calculated (the volume of collected objects/the minor space usage prior to the collection). If the efficiency is higher than a predefined minor collection threshold or th_{minor}, nothing else is done. If the efficiency is below th_{minor}, full collection is invoked. Notice that in the subsequent experiments, we set the value of th_{minor} to be 80%. We selected this value because in all benchmarks, the minor collection efficiencies seldom fall below 80%.

While experimenting with this approach, we made two observations:

1. A large portion of allocation pauses not associated with live-size reductions are originated from a set of method call sites.
2. A large portion of allocation pauses with major live-size reductions are originated from another set of method call sites.

We exploited these two observations to further optimize our triggering mechanism by incorporating *bad-site table* and *major-reduction table* to memorize method call sites (a method call site is represented by the *thread_id, method_id*, and *method_call_depth*) that usually create allocation pauses with poor live-size reductions and allocation pauses with large live-size reductions, respectively. Figure 6 describes the optimized algorithm.

Similar to the algorithm described earlier, the optimized algorithm invokes minor collection upon detecting a pause and calculates the collection efficiency. If the efficiency is less than th_{minor}, full collection is invoked and, once again, its efficiency is calculated. If this efficiency is below the full collection threshold or th_{full}, the method-call site is recorded in the *bad-site* table (line 15 in Figure 6). If the efficiency is higher than th_{full}, the method-call site is recorded in the *major-reduction* table (line 13 in Figure 6). Later, if a site in the bad-site table initiates a pause, no garbage collection is invoked (lines 2 and 3 in Figure 6). On the other hand, if a site in the major-reduction table initiates a pause, full collection is invoked directly (lines 4 and 5 in Figure 6).

In the subsequent experiments, we set the value of th_{full} to be 60%. We selected this value because our preliminary study indicated that the efficiencies of mature collection rarely fall below 60% in all benchmarks. We also set each of the hash tables (i.e. the bad-site table and major-reduction table) to 20KB, which is sufficient for our benchmarks.

4. Evaluation of MicroPhase

In this section, we describe the experimental environment used to evaluate the performance and accuracy of MicroPhase. Our benchmark collection comprised of SPECjvm98 benchmark suite [27], DaCapo benchmark suite (the version as of 10/2006) [3] with the default configuration and SPECjbb2000 [26] with 8 warehouses. These programs were selected for their ability to test the performance of garbage collectors by providing complex heap behaviors and wide ranging memory demands. Table 3 highlights the key characteristics of our benchmarks.

```
(1)  S ← the current method call site
(2)  if S in Bad-Site Table then
(3)     Do not trigger GC at this pause
(4)  else if S in Major-Reduction Table then
(5)     Trigger full collection
(6)  else
(7)     Trigger minor collection
(8)     Eff1 ← Compute the efficiency of the minor collection
(9)     if ( Eff1 < th minor) then
(10)        Trigger a full collection
(11)        Eff2 ← Compute the efficiency of the full collection
(12)        if ( Eff2 > th full ) then
(13)           Insert S into Major-Reduction Table
(14)        else
(15)           Insert S into Bad-Site Table
(16)        endif
(17)    endif
(18) endif
```

Figure 6. Optimized GC triggering algorithm in MicroPhase

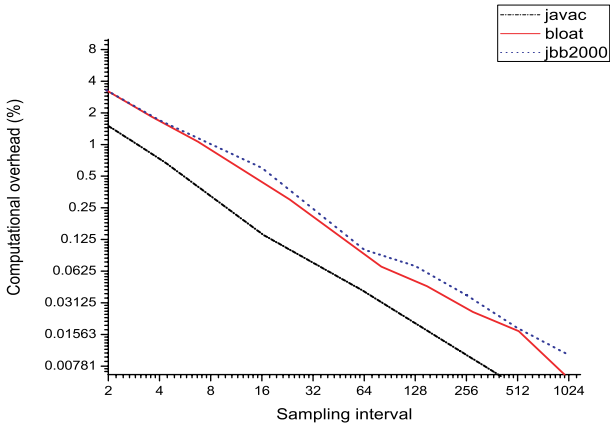


Figure 7. Effect of sampling interval on allocation pause detection overhead

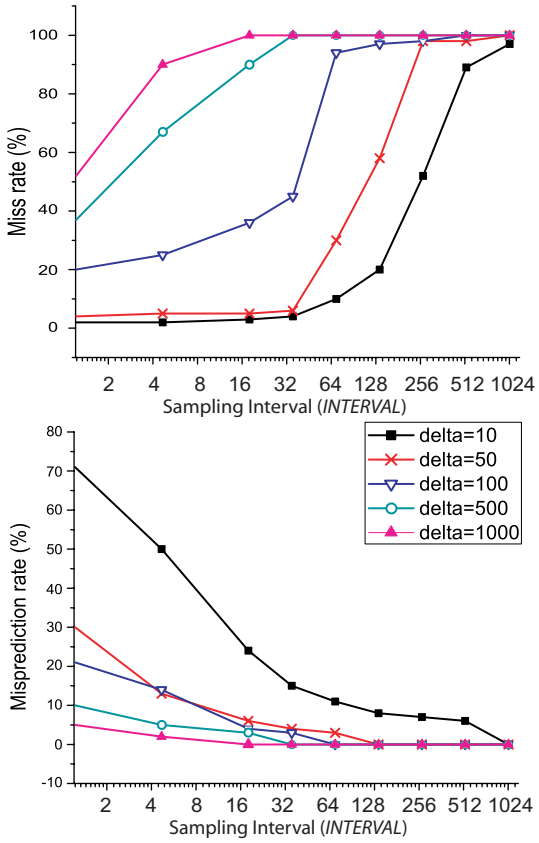


Figure 8. Miss rate and misprediction rate of pause detection algorithm in *javac*

We ran all of the benchmarks in a machine with two AMD Opteron processors and 16GB of physical memory. In all experiments, we used the MicroPhase enabled Hotspot (shipped as part of JKD version 1.4), and the young generation area was set to the default value used by Sun for the 64-bit Opteron, which is 1/3 of the entire heap. *The heap size was set to be twice of the maximum live-size to allow reasonable invocation frequency while still yielding good garbage collection performance* in most applications. In case of multithreaded benchmarks (*mtrt*, *eclipse*, *hsqldb*, *lusearch*, *xalan*, and *SPECjbb2000*), we enabled the *Thread-Local Allocation Buffer* (*TLAB*) feature in HotSpot to reduce the synchronization overhead during object allocations. We executed our benchmarks in a standalone mode with all non-essential daemons and services shut down to minimize the number of other threads competing for the processor time.

4.1 Overhead and Accuracy of MicroPhase

To evaluate the accuracy of MicroPhase, the heap was set to a very large size so that all applications can run without having to invoke GC. The only exception is *javac*, which forces GC four times during its execution.

Because computational overhead of the pause detection algorithm is tightly related to the allocation pressure (i.e. more allocation requests mean higher overhead), we selected three benchmarks with three distinctive memory pressures: *javac* (light pressure), *bloat* (moderate pressure) and *SPECjbb2000* (intense pressure). Figure 7 represents the overhead of MicroPhase (measured in percentage of the overall execution time) of each of the three Java benchmarks over a wide range of sampling intervals. The graph shows an expected trend of decreasing computation overhead with each larger sampling interval (*INTERVAL*). Our experiment shows that the overhead ranges from 3% to 5% of the overall execution time if pause detection is attempted at each allocation request. The overhead can be further reduced to less than 1% if *INTERVAL* is 16 allocations or larger.

We then studied the accuracy of MicroPhase by measuring the miss rate (number of allocation pauses not detected/total number of allocation pauses) and the mispredic-

Benchmark	Maximum live-size (MB)	Allocated objects		Number of threads	GC execution overhead (%) (2x max live size)
		size (MB)	number (x 10^6)		
compress	6.5	110.46	0.01	1	3.7
db	8.5	78.04	3.21	1	2.7
jack	0.75	181.19	5.90	1	10
javac	8.09	212.99	6.31	1	30.1
jess	1.11	297.10	7.94	1	56.1
mpegaudio	0.7	1.16	0.02	1	0.5
mtrt	7.08	139.99	6.64	2	77
raytrace	4.5	159.92	6.37	1	5.1
antlr	1.1	278.02	5.24	1	5.4
bloat	3.51	1092.90	27.69	1	10.6
chart	12.5	787.38	27.19	1	8.4
eclipse	12.7	888.57	47.14	11	13.7
fop	7	54.27	1.22	1	4.6
hsqldb	67.87	134.35	4.43	85	57.1
jython	2.75	1216.95	19.73	1	18.2
luindex	1.4	398.79	11.34	1	11.2
lusearch	12.1	2102.24	16.40	30	21.1
pmd	15.99	810.53	33.88	1	23.9
xalan	20.11	1018.95	14.65	7	24.7
jbb2000 (8 whs.)	231.08	2900.11	91.21	8	19

Table 3. Basic characteristic of SPECjvm98, SPECjbb2000, and DaCapo benchmark suites. For SPECjbb2000, the workload is set to 8 warehouses. For DaCapo, the default workload is used.

tion rate (number of detected allocation pauses with no live-size reductions/number of detected allocation pauses). For the sake of brevity, we only depict the result from *javac*. We chose *javac* because our previous study indicated that (i) the lengths of the allocation pauses are wide ranging, (ii) there are over 12000 instances of allocation pauses (see Table 1), and (iii) there are a large number of allocation pauses with no live-size reductions. Figure 8 shows the miss rate ($missRate$) and misprediction rate ($mispredictRate$) over wide ranging values of *INTERVAL* and δ. Based on Figure 8, we made two observations:

1. Larger δ results in a smaller misprediction rate but a larger miss rate. The contrary occurs when δ is small. Because higher misprediction rates can cause more untimely GC invocations, *the misprediction rate is considered more critical than the miss rate*.
2. Larger *INTERVAL* results in smaller misprediction rate but larger miss rate. The contrary occurs when *INTERVAL* is small.

From the observations, the ideal configuration of *INTERVAL* and δ should yield small miss rate and accurate prediction across all applications. We used the following scoring formula to rank each configuration of C(*INTERVAL*, δ):

$$Score(C) = w \times mispredictRate(C) + (1-w) \times missRate(C)$$

The parameter w is the weight of the misprediction rate, which is considered more critical than the miss rate. We conducted an experiment to identify a good value for w and found that 0.9 works well in all applications. The result shows that when *INTERVAL* is equal to 16 and δ is equal to 50, we can achieve the misprediction rates of about 3% and the miss rates of less than 10% in most applications.

4.2 Uncontrolled versus Controlled Modes

In MicroPhase, when $hThreshold$ is set to zero, the collector operates in the *uncontrolled* mode (i.e. MicroPhase triggers GC each time a pause is detected). On the other hand, when $hThreshold$ is set to 100%, the traditional space-based criterion is used to trigger GC. Any other values of $hThreshold$ between these two extremes set the collector to operate in the *controlled* mode (i.e. pauses are used to trigger GC after the heap usage has surpassed $hThreshold$). We conducted an experiment to identify the optimal $hThreshold$ for every application. Our experiment consisted of running every benchmark with a set of $hThreshold$ values ranging from 0 to 100 in multiples of 10. Every configuration of each benchmark was executed three times, and the value of $hThreshold$ that yielded the shortest GC time was considered the optimal value for that benchmark. The result of our experiment is reported in Table 4.

Our result indicates that the controlled mode is more effective than both the uncontrolled mode and the space-based mode as there are no applications that yield the shortest GC time when $hThreshold$ is equal to 0% or when $hThreshold$ is equal to 100%. In subsequent experiments, we will compare the GC performances of MicroPhase with the optimal $hThreshold$ and fixed $hThreshold$ configurations (we re-

Benchmark	Optimal $hThreshold$ (%)	GC time reduction (%)
compress	60	2.73
db	70	3.83
jack	80	1.96
javac	80	9.94
jess	70	26.47
mpegaudio	70	1.84
mtrt	80	16.99
raytrace	60	15.36
antlr	60	5.12
bloat	80	1.95
chart	60	5.12
eclipse	70	11.64
fop	70	0.11
hsqldb	90	10.94
jython	80	3.17
luindex	70	21.05
lusearch	80	2.07
pmd	60	8.29
xalan	70	-0.59
jbb2000 (8 whs.)	70	11.79

Table 4. Identifying the optimal heap usage threshold ($hThreshold$) for each application

fer to the latter configuration as *fixed-value*) against that of the default space-based collector in HotSpot.

5. Performance Comparisons

So far, we have identified the optimal value of $hThreshold$ for each application. However, obtaining this value in real-world settings may require considerable amount of tuning effort, and thus, may not always be practical. Therefore, we introduced a *fixed-value* configuration as another system under investigation. The fixed-value system uses the same $hThreshold$ across all applications. By examining the optimal values of *hThreshold*, we notice that the majority of them are 70% and 80%. In the fixed-value configuration, $hThreshold$ is set to 70%, which represents the average of all the optimal values. Notice that there are eight applications—db, jess, mpegaudio, eclipse, fop, luindex, xalan, and SPECjbb2000—that also have 70% as the optimal *hThreshold*.

We then conducted experiments to study the performances of MicroPhase with the optimal configuration, MicroPhase with the fixed-value configuration, and the space-based triggering approach. For every technique, we ran each benchmark *three times, and the one with the shortest garbage collection time was used*. We then compared the results with respect to three performance metrics: *overall garbage collection time*, *minimum mutator utilization (MMU)*, and *overall performance*. Note that for the overall performance, we measured execution time in all benchmarks except SPECjbb2000 in which we measured the throughput. Before we discuss the results of our experiments, we wish to report the basic GC behaviors when these three triggering techniques are applied.

5.1 Overall Garbage Collection Time

Table 5 quantifies the reduction in the number of GC invocations and the times spent in minor and full collection when both configurations of MicroPhase are used instead of the space-based technique. Moreover, the table also shows small differences in GC performances between the fixed-value configuration and the optimal configuration. Figure 9 highlights the differences in GC performances between the two configurations of MicroPhase. Notice that the fixed-value configuration shows no more than 5% increase in the overall garbage collection time when compared to the optimal configuration.

In *jess*, MicroPhase invokes eight times more minor collection; however it also invokes 239 times less full collection. Such a magnitude of reduction is achieved because MicroPhase promotes fewer objects to the mature generation. As a result, our technique can reduce the total time that *jess* spent in garbage collection by 26%. Similar results are seen in *javac*, *mtrt*, *chart*, *luindex*, and *pmd* where MicroPhase invokes many times more minor collection, but reduces the number of full collection invocations and the time spent in performing full collection (see Table 5 and Figure 10). In *compress*, *jack*, *mpegaudio*, *SPECjbb2000*, *bloat*, *hsqldb*, *jython*, and *lusearch*, MicroPhase can reduce the minor collection times, full collection times, and overall GC times. In *db* and *raytrace*, full collection is rarely invoked. Thus, the reductions in minor collection times are the factor that reduces the overall GC time.

Xalan is the only application that does not benefit from using MicroPhase. This is because *xalan* has the smallest occurrences of allocation pauses. Moreover, each pause only accounts for a relatively small live-size reduction. *Bloat* also makes a very interesting case study. All three techniques report exactly the same numbers of minor collection (1390) and full collection (1) invocations in *bloat*. We also find that MicroPhase (both configurations) yields slightly shorter times in minor, full, and overall collection. Shorter minor collection time is achieved because MicroPhase promotes fewer objects. Shorter mature collection time is obtained because MicroPhase invokes the only full collection at a time that yields higher efficiency than that of the space-based approach.

Hsqldb is another interesting benchmark. Because it uses memory resident databases, a large number of objects are kept alive for most of the execution. Thus the maximum live-size is nearly 50% of the total allocated objects. The optimal performance is also achieved when $hThreshold$ is set to 90%, meaning that MicroPhase is not considered until the heap is nearly full. Still, the utilization of MicroPhase results in two more minor collection invocations, but with

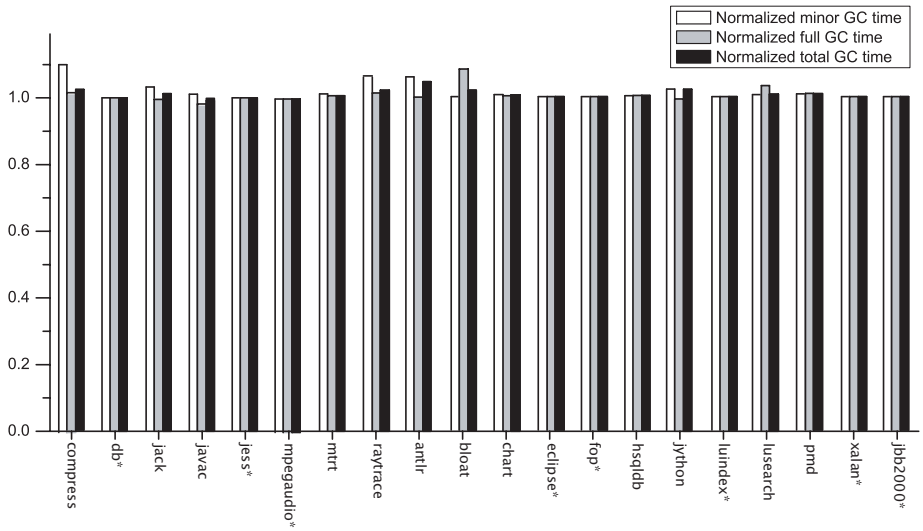


Figure 9. Normalized GC time reductions of MicroPhase with fixed $hThreshold$ (70%) over MicroPhase with optimal $hThreshold$

Benchmark	Space-based scheme				MicroPhase (optimized)				MicroPhase (hThreshold =70%)			
	Minor GCs		Full GCs		Minor GCs		Full GCs		Minor GCs		Full GCs	
	Calls	Seconds)	Calls	Seconds	Calls	Seconds	Calls	Seconds	Calls	Seconds	Calls	Seconds
compress	44	0.02	21	0.17	40	0.02	21	0.17	44	0.02	21	0.17
db*	117	0.15	4	0.16	126	0.14	4	0.16	126	0.14	4	0.16
jack	345	0.24	20	0.27	347	0.23	19	0.27	345	0.24	20	0.27
javac	323	1.07	33	2.55	333	1.17	30	2.09	391	1.16	28	2.12
jess*	566	0.03	551	9.83	574	0.04	212	7.21	574	0.04	212	7.21
mpegaudio*	1	0.004	2	0.02	1	0.003	2	0.02	1	0.003	2	0.02
mtrt	134	0.15	251	15.8	132	0.16	261	13.08	141	0.16	242	13.16
raytrace	304	0.20	2	0.020	304	0.17	2	0.021	307	0.17	2	0.022
antlr	558	0.67	8	0.21	570	0.63	8	0.2	558	0.67	8	0.2
bloat	1390	2	1	0.05	1390	1.97	1	0.04	1391	1.98	1	0.05
chart	445	2.09	8	1.1	428	2.11	6	0.75	448	2.13	6	0.76
eclipse*	2760	10.71	3	0.58	2773	8.88	7	1.09	2773	8.88	7	1.09
fop*	55	0.43	0	0	55	0.43	0	0	55	0.43	0	0
hsqldb	12	1.02	5	2.91	14	1	5	2.5	14	1	5	2.51
jython	1858	8.7	2	0.13	1860	8.42	2	0.13	1887	8.65	2	0.13
luindex*	765	0.67	260	5.22	842	0.74	179	3.91	842	0.74	179	3.91
lusearch	4012	4.61	7	0.22	4026	4.54	7	0.19	4059	4.58	7	0.2
pmd	377	2.54	9	1.32	382	2.56	9	0.98	384	2.59	9	0.99
xalan*	601	2.76	33	2.3	651	2.82	33	2.27	651	2.82	33	2.27
jbb2000*(8 whs)	4798	140.77	284	96.03	4771	128.4	280	80.47	4771	128.4	280	80.47

Table 5. Comparing GC behaviors when MicroPhase (optimal and fixed-ratio configurations) and space-based triggering are used (* indicates application that have the same optimal $hThreshold$ as the fixed-ratio $hThreshold$)

slightly less time spent in each invocation when compared to that of the space-based approach. This is another indication that MicroPhase improves the efficiency of minor collection. As a result, a small GC time reduction is achieved with MicroPhase using the optimal $hThreshold$ value. In the fixed-value approach, the earlier application of MicroPhase (at 70% instead of 90%) also triggers the same numbers of minor collection and full collection invocations as the optimal configuration. However, slightly more time is spent in full collection than that of the optimal configuration.

Raytrace and *eclipse* are the only two benchmarks that spend more time performing full collection when MicroPhase is used. With the space-based approach, these two applications invoke full collection only a few times: two and three invocations for *raytrace* and *eclipse*, respectively. With MicroPhase, *raytrace* still invokes the full collection twice, but at different points in execution than the invocation points in the space-based technique. Unlike *hsqldb*, these two points are not as efficient as the invocation points in the space-based technique. This difference contributes to slightly longer time

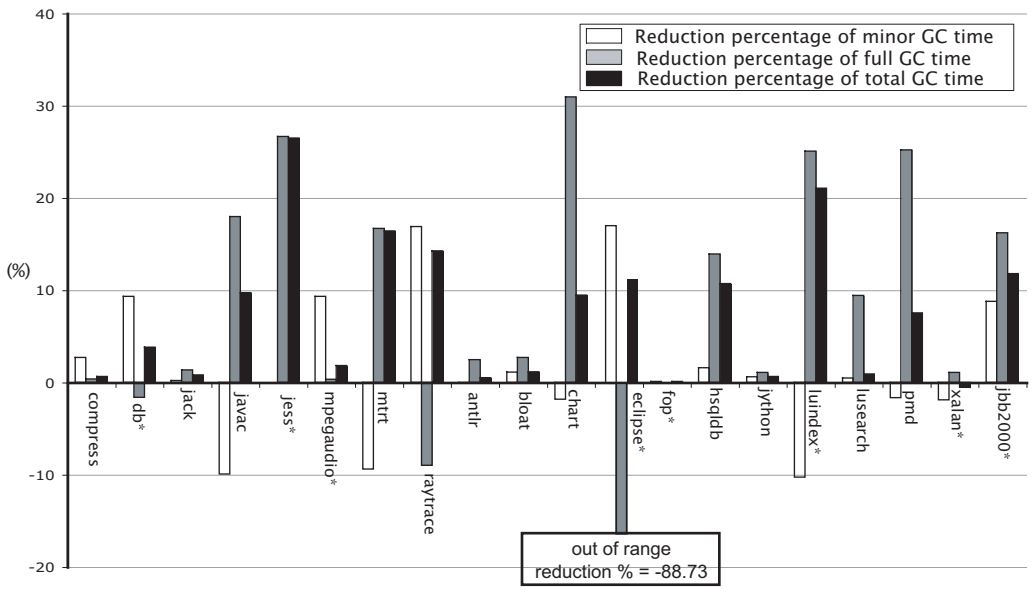


Figure 10. Percentage of GC time reductions when fixed-value configuration of MicroPhase is used.

spent in full collection. For *eclipse*, there are several pauses that yield less than 80% minor collection efficiencies. When this occurs, MicroPhase invokes the full collection. As a result, MicroPhase invokes the full collection four more times than the space-based approach.

MicroPhase also performs well in SPECjbb2000, an application that allocates nearly 3 GB of space. Even though the differences in the number of minor and full collection invocations between MicroPhase and the space-based approach are very small, MicroPhase can reduce the average times spent in each of the minor and full collection invocations. As a result, MicroPhase can achieve a 12% reduction in the overall garbage collection time.

5.2 Garbage Collection Pause

We used MMU (minimum mutator utilization) [9] to measure the pause time and mutator utilization. Mutator utilization is the fraction of the time that an application (or mutator) executes within a given window. For example, given an execution window of 10 ms, within that time the collector runs for 4 ms, and the mutator runs for 6 ms. Thus, the mutator utilization is 60%. The *minimum mutator utilization* (MMU) is the minimum utilization across all execution windows of the same size. For example, an MMU of 40% at 10 ms means that the application will at least execute 4 ms out of every 10 ms. Figure 11 and Figure 12 depict the MMU of every benchmark. The x-intercept indicates the maximum pause time, and the asymptotic y-value indicates the mutator utilization.

In *luindex*, the utilization of MicroPhase is better than the space-based scheme throughout all window sizes. As shown in Figure 12(q), the space-based scheme has the largest x-intercept value of around 0.03s, and its mutator utilization is about 88%. The x-intercept of MicroPhase is 33% smaller (at about 0.02s) due to fewer object promotions, resulting in the mutator utilization (asymptotic y-value) of 92%. The main reason for higher MMU is due to the MicroPhase ability to reduce the minor, full, and overall GC times. MicroPhase also reduces minimum pause times and slightly improves the overall MMU in *jess*, *mtrt*, and *hsqldb*.

For SPECjbb2000, the mutator utilization of MicroPhase is better than space-based scheme when the window size is smaller than 16 seconds (see Figure 11(b)). After that, the utilizations of the two schemes are nearly the same. In *db*, *jack*, *javac*, *bloat*, *chart*, *eclipse*, *jython*, *lusearch*, and *pmd*, we also observed that MicroPhase could improve the MMUs when the window sizes are small.

For *xalan*, the MMU of MicroPhase is nearly the same as the MMU of the space-based scheme. This is because both triggering mechanisms yield nearly the same number of minor and full collection invocations. Moreover, the times spent in minor collection are also nearly the same. As a result, very little differences are observed. The other four benchmarks that yield similar MMU results are *compress*, *mpegaudio*, *raytrace*, and *fop*.

For *antlr*, both configurations of MicroPhase have worse utilization than the space-based technique. Further investigation reveals that *antlr* has very bursty allocation requests, which may be due to the volume of objects needed to store the contents of each file and grammars. Within each burst, there are also small allocation pauses that cause MicroPhase to invoke minor collection in burst. As a result, the mutator utilization when the windows are small is not as high as the space-based approach.

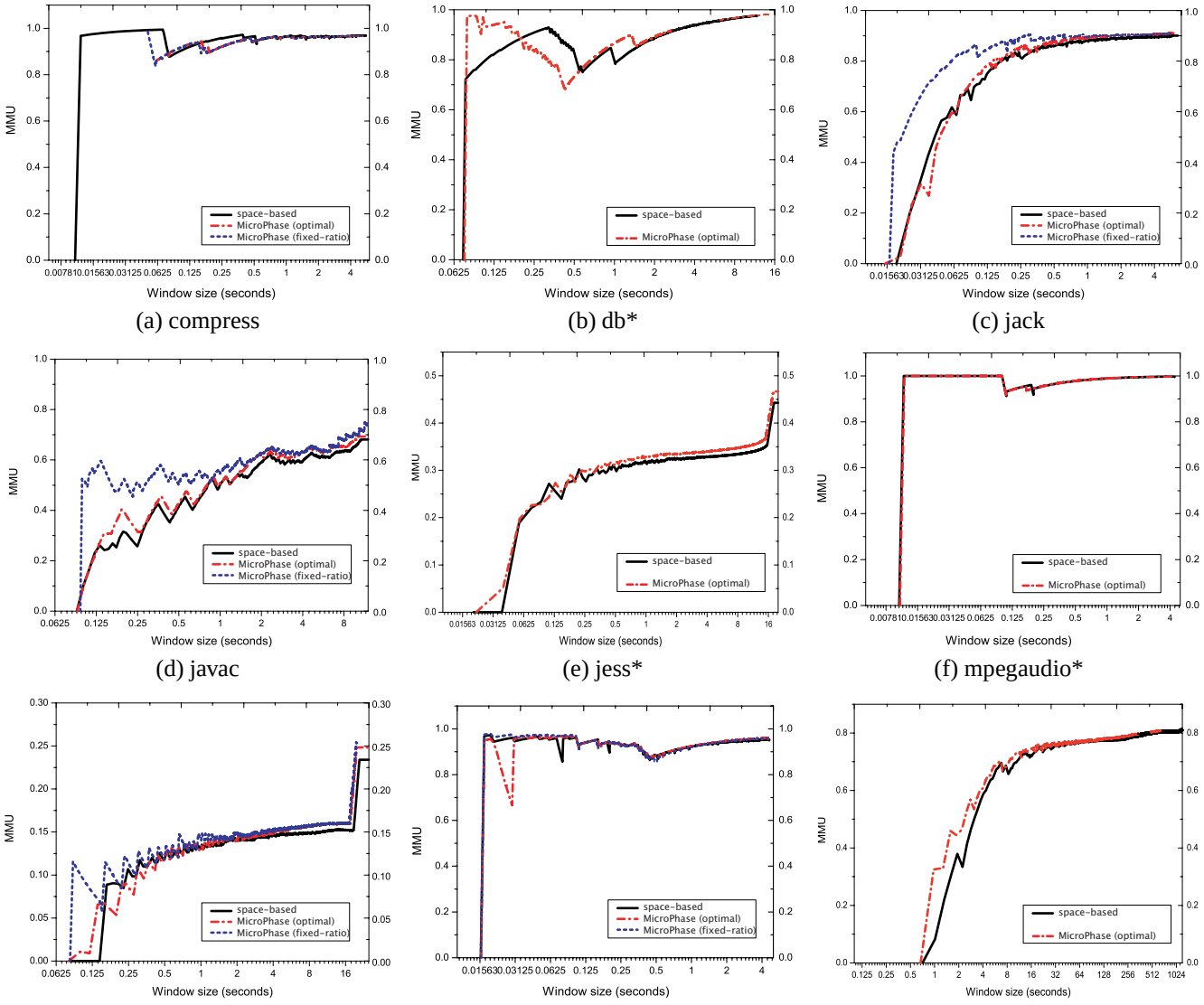


Figure 11. Minimum mutator utilizations (MMUs) of SPECjvm98 benchmarks and SPECjbb2000

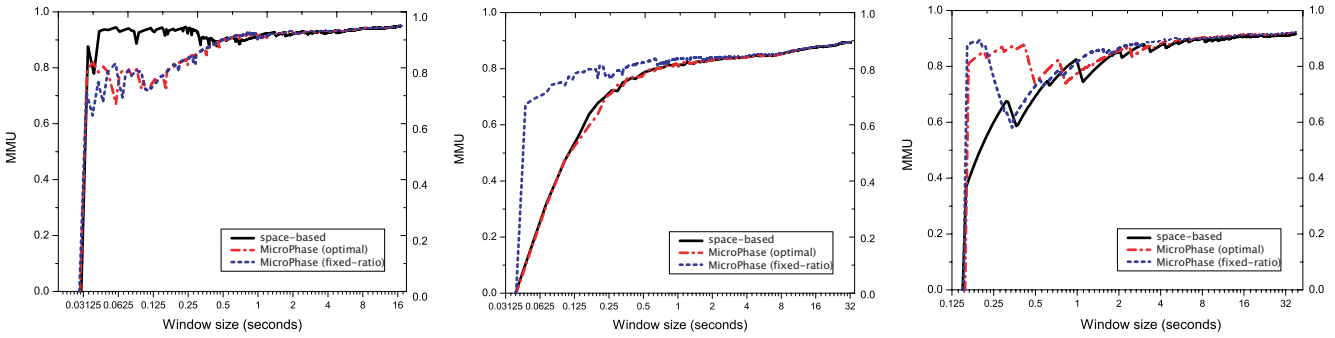
In conclusion, we found that MicroPhase improves mutator utilization in fifteen out of the twenty benchmarks. It also has no positive or negative effects on the other four benchmarks and degrades the MMU of only one benchmark (*antlr*).

5.3 Overall Performance

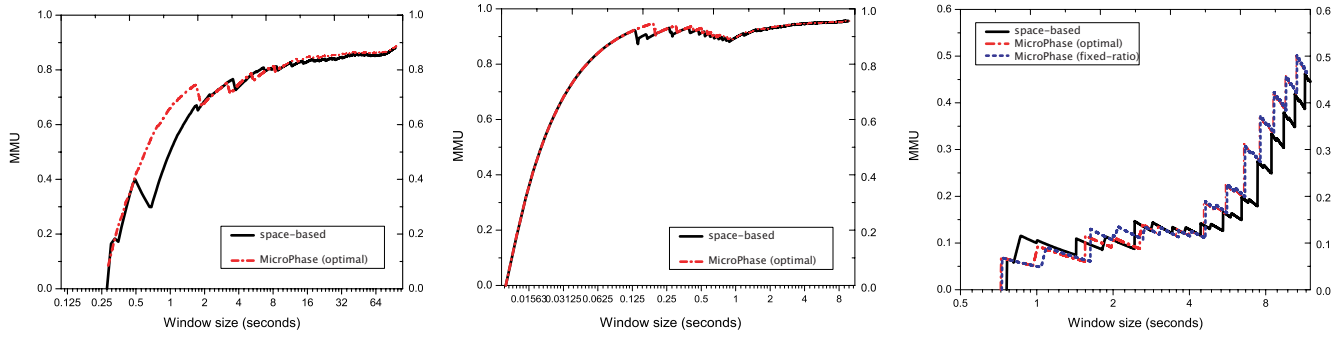
Figure 13 depicts the reduction in the overall execution time of each of the SPECjvm98 and DaCapo benchmarks. The graph shows reductions in the execution times of 12 benchmarks, ranging from 1% to 14%. For SPECjbb2000, we compared the throughput of MicroPhase with that of the space-based approach instead of comparing the overall execution time. As explained earlier, MicroPhase yields higher utilization when the window size is smaller than 16 seconds and very similar utilization to the space-based approach afterward. As a result, MicroPhase improves the throughput by 5%.

For *bloat*, *chart*, *fop*, *jack* and *jython*, MicroPhase can only achieve slight reductions in GC times, and thus, these reductions are offset by the runtime overhead to enable MicroPhase (pause detection and minor or full collection decision), which collectively account for about 2% overhead to the mutator. As a result, the overall execution times of these applications increase by 0.5% to 2%.

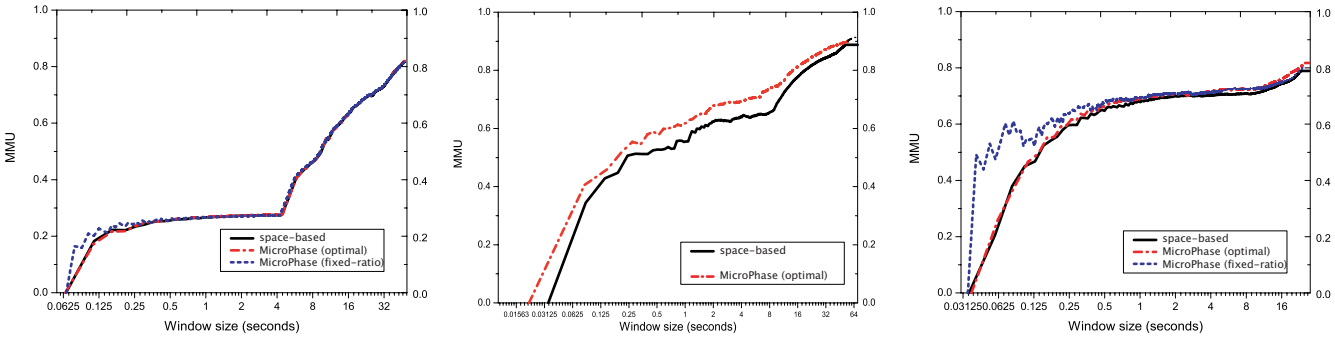
For *lusearch*, MicroPhase cannot significantly reduce the amount of times spent in minor collection and full collection. Thus, the overall execution time is half a percent longer. We suspect that the major reason for MicroPhase inefficiency in *lusearch* is multithreading. (More discussion about apply-



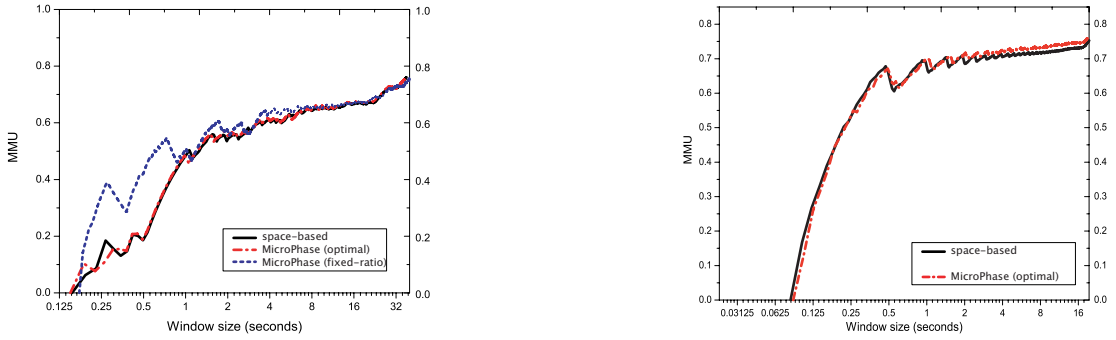
(j) antlr (k) bloat (l) chart



(m) eclipse* (n) fop* (o) hsqldb



(p) jython (q) luindex* (r) lusearch



(s) pmd (t) xalan*

Figure 12. Minimum mutator utilizations (MMUs) of DaCapo benchmarks

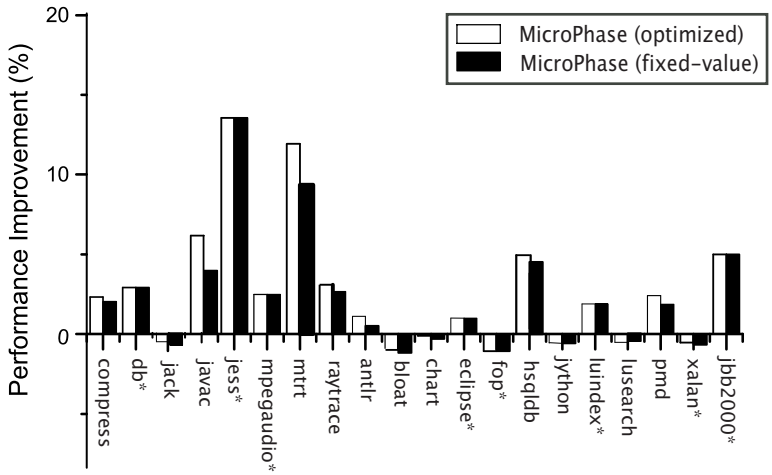


Figure 13. Overall performance improvements in all benchmarks when both configurations of MicroPhase are used. For SPECjbb2000, the improvement reflects the increase in throughput performance. For each of the remaining benchmarks, the improvement represents the percentage of reduction in the overall execution time.

ing MicroPhase in multithreaded environments is discussed in Section 6.) Because our technique detects pauses occurring within each thread, it is likely that a pause detected in one thread does not represent allocation phase boundaries in other threads. While such an issue also exists in SPECjbb2000 and *hsqldb*, it does not manifest itself because:

1. *hsqldb* allocates a large number of long-lived objects. Moreover, GC also dominates the cost of execution. Our technique was able to predict proper locations to invoke the full collection, and therefore, reasonable reductions in the overall GC time and execution time are achieved. However, MicroPhase was less successful in predicting when to efficiently invoke the minor collection, a more difficult task due to different allocation phases in each thread.
2. *SPECjbb2000* spawns eleven threads but only eight are actively used. Moreover, these eight threads are launched sequentially as the tasks become more demanding. As a result, MicroPhase can work efficiently early on when the number of active threads is still small.

We also observe that MicroPhase can also impose positive effects on the mutator performance. For example, its adoption, while incurring some runtime overheads, can also reduce the occurrences of write-barriers and intergenerational pointers because there are fewer surviving new objects (SNOs).

5.4 Heap Size Sensitivity

To investigate how different heap sizes affect MicroPhase, we conducted an experiment utilizing five different heap configurations in all applications: twice (2X), three times (3X), four times (4X), five times (5X), and six times (6X) the maximum live-size. Figure 14 depicts our results. The x-axis is the normalized heap size (relative to the maximum live-size), and y-axis is the overall GC time reduction of MicroPhase compared to the space-based approach operating at the same heap size.

In most benchmarks, our result indicates that MicroPhase has the highest reduction of GC time when the heap configuration is 2X due to more frequent GC invocations, which create more opportunities for savings. In addition, the likelihood that these GCs are triggered in an untimely manner in the space-based approach (with 2X configuration) is much higher than in the larger configurations; as the heap becomes larger, the need to invoke GC lessens.

We also observe that in *xalan*, *pmd*, *jython*, *luindex*, *eclipse*, *bloat*, *chart*, and *SPECjbb2000*, MicroPhase also performs very well when the heap is larger than 2X. The dominating cost of minor collection is the main reason for MicroPhase to perform well. In these applications, minor collection is invoked very frequently while the full collection is rarely called (see Table 5). Enlarging the heap would significantly reduce the number of full collection invocations or eliminate the need to invoke it altogether; however, the minor collection is still periodically invoked. Because MicroPhase often yields higher minor collection efficiencies, it can reduce collection time of each minor GC invocation. In addition, as the time spent in GC becomes smaller with a larger heap size, a slight reduction in the GC time can result in a large percentage of improvement.

6. Future Studies

In this paper, we have shown that MicroPhase can provide an effective alternative to the traditional space-based triggering mechanism. However, these experimental results should not be viewed as the best performance capability of MicroPhase;

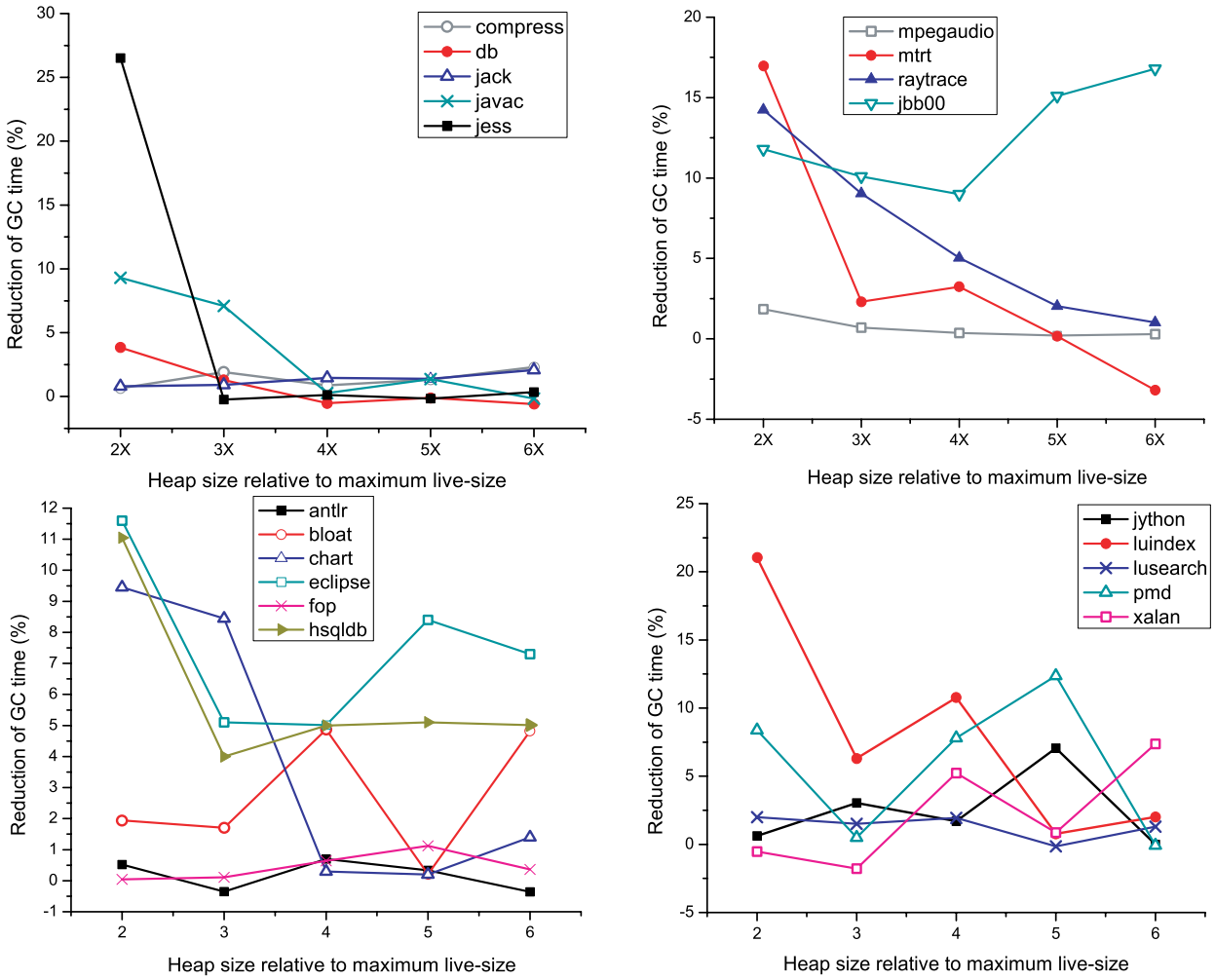


Figure 14. GC time reduction under different heap size for SPECjvm98, jbb2000 (the two upper figures) and DaCapo (the two lower figures).

instead, they represent a glimpse of its potential benefits. Because this is the first attempt to use allocation phases to trigger GC, there is still plenty of room for improvements and further studies.

For example, it is unclear how efficient the proposed technique is on large multithreaded applications running on uniprocessor systems. When a pause is detected in an execution thread, it does not mean that other mutator threads are also experiencing allocation pauses. Thus, triggering GC at this point may be locally optimal based on the allocation behavior of that thread but not globally optimal with respect to the remaining threads. One possible solution is to record allocation status of every threads and invoke GC when the majority is experiencing allocation pauses. We also observed from our past studies [36, 35] that the amount of work done by each thread in large server applications is not uniform. For example, we noticed that 10% of threads in SPECjAppServer2004 contribute to about 80% to 90% of all allocated objects. It may be possible to apply MicroPhase to these threads and then use space-based technique with the remaining threads. We are currently investigating this approach.

Another existing techniques designed for multithreaded programs in multi-processor environments are thread local heaps and thread specific heaps [28, 12]. The main idea is to create one subheap for every thread. Any objects shared by multiple threads are allocated in a shared heap area. Applying MicroPhase to these techniques will make pause detection more straightforward and allow GC to be invoked independently in each subheap.

Recent efforts by Guyer and McKinley [13] and Zee and Rinard [37] have shown the power of flow-sensitive pointer analysis to improve garbage collection performance. Flow-sensitive pointer analysis can also be used to optimize MicroPhase to identify areas with intense and light allocation

requests and target the areas with light requests as possible GC invocation points.

7. Related Work

In a working paper [33], Wilson proposes *opportunistic garbage collection*, a non-disruptive generational technique that hides GC pauses in compute-bound phases of program execution. The rationale behind this approach is that there are two major execution phases in a program: compute-bound and user-interactive. Thus, Wilson proposes two heuristics to hide pauses due to minor and major collection. First, user-oriented heuristics schedule scavenges (i) at the end of non-interactive segments of program execution and (ii) at times when the system is idle. Second, computation-oriented heuristic schedules scavenges at a minimum stack height, where the volume of live objects is small. We think that the computation-oriented heuristic may be an indirect approach to identify allocation pauses [33].

Later on, Wilson and Moher [34] present the design of opportunistic garbage collection. To support the user-oriented heuristics, they attach a small routine to each user-interactive routine to measure the time since the last user interaction. Another routine is used to detect user-interaction pauses. When a pause is detected, the system decides whether to scavenge and how many generations to scavenge. They also suggest that each user-interaction pause can still create a large enough volume of objects to cause minor collection invocations. In such a scenario, the authors claim that the goal of hiding these pauses is still accomplished as these invocations occur during an interaction pause.

Comparatively, each of our allocation pauses is much more fine-grained than each of their user-interaction pauses. When an allocation pause occurs, there are no objects allocations at all while objects are still created during a user-interaction pause. Thus, one major distinction is that our technique attempts to trigger GC at allocation pauses while their approach tries to trigger GC at each user-interaction pause, in which many allocation pauses can still occur.

Another major difference is that our primary objective is to improve the garbage collection performance, while their primary objective is to hide GC pauses. As by-products of these two primary objectives, our approach can yield shorter pauses in some applications, while their approach often yields better GC performance. In terms of implementation, MicroPhase relies on low-overhead runtime monitoring mechanisms and not program annotations to guide the GC triggering decision process.

Work by Qian *et al.* [21] suggests that allocation phases exist in five Java applications, *javac*, *jack*, *jess*, *db*, and *mtrt* from SPECjvm98. They define "activity zones" where a zone represents an execution area with a certain allocation characteristic. They use allocation rate and allocation variation as two criteria in identifying activity zones. They use calculation windows to define the number of sample points needed to calculate the allocation rates. Through simulation, they evaluate the accuracy of their zone detection algorithm and project the improvement in GC efficiency and heap usage when their approach is used. Their results show that in applications with clearly distinct phases (*jack* and *jess*), significant performance gains can be achieved (17% and 30%, respectively). However, their zone detection algorithm fails to detect many if any zones in the remaining three applications, resulting in modest performance improvement.

Comparing to our work, the main motivation is the same; that is there are allocation phases in most Java applications and triggering GC during the execution zones with no allocation activities can improve GC performance. However, we leverage allocation pauses instead of changes in allocation rates as a triggering criterion. The results show that our algorithm can detect execution areas with no allocation activity more accurately than their technique (e.g. our technique can detect many pauses in *mtrt*, while their technique detects none). Another major difference is that our results are based on actual implementation inside a generational collector, while their results are based on simulation of mark-sweep collector.

Work by Buytaert *et al.* [7] uses profile-directed approach called *garbage collection hints* or *GCH* to improve the performance of the Appel collector in Jikes RVM [16]. Their technique uses off-line profiling to identify favorable collection points (FCPs). The collector then dynamically selects whether minor or full collection should be invoked when a favorable collection point is reached. Experimental results (using six SPECjvm98 benchmark) show that GCH can achieve up to 10% reduction in execution time and 29% reduction in garbage collection time.

To identify FCPs, an off-line analysis calculates the live/time function, which represents the volume of live objects (in bytes) over total allocated bytes. The FCPs for each application are then selected from the profiled result; each FCP is represented by a method. They discovered that some benchmarks have as many as three FCPs while others have only one FCP. A cost model is then used to decide if no GC, full GC, or minor GC should be invoked at each FCP. There are three cost components in the model: minor collection cost, full collection cost, and remembered set processing cost.

One major difference between their technique and ours is that we do not use profiling to detect good collection points, instead, we associate these points with allocation pauses. We have yet to study the relationship between our pauses and their FCPs but we hypothesize that they are the same. Because we do not use profiling, we cannot estimate the amount of live-size at each detected point. Instead, we use past performances at these method invocation sites to predict whether to invoke minor or full collection.

Brecht *et al.* [6] suggests that the amount of available memory should be used to regulate heap resizing and in-

voke garbage collection, instead of using heap usage as the main GC invocation criteria. Their technique leverages the heap usage information to select a GC policy from a pool of three policies: if the heap is lightly utilized, enlarge it aggressively; if the heap is heavily utilized, invoke GC aggressively; and if the last collection is not effective, do not collect. We also use heap usage information to regulate when MicroPhase should be utilized (i.e. when heap usage surpasses $hThreshold$). Once MicroPhase is fully engaged, we leverage allocation phase information to invoke GC. Their technique does not consider phase information.

Efforts by Bacon *et al.* and Gestegard *et al.* [1, 23] explore the use of time-based garbage collection to improve real-time performance by using time to control GC triggering. In effect, these garbage collectors become periodic instead of sporadic [23]. Chen *et al.* [8] proposes a technique that proactively invokes GC to improve cache and page localities.

One consequence of using the space-based criterion is a large volume of surviving new objects or SNOs, Work by Stefanović *et al.* proposes *Older-First* algorithm [30, 29], which concentrates its collection effort on older objects. Similarly, work by Blackburn *et al.* proposes the *Beltway* framework [4], in which the heap is partitioned into several belts, and each belt consist of several segments. Beltway uses Older-First algorithm to collect each belt. While Older-First and Beltway can effectively deal with SNOs, they do not address the issue of untimely GC invocation.

Phase is often defined as a period of work (e.g. computation, allocation). In *phase aware* computing, phase information is used to assist with resource reclamation or program optimization. Work by Nagpurkar *et al.* proposes an online framework for phase detection [20, 19]. The phase detection algorithm executes concurrently with the program and detects phase boundaries by computing and evaluating the similarity between two windows of profile elements (e.g. execution events). Our technique can be classified as an instance of their proposed framework.

Work by Ding *et al.* defines a phase as a "unit of recurring behavior, and boundaries of a phase can be uniquely marked in its program" [11]. Their technique called *gated memory control* leverages phase information to monitor and control memory usage. One interesting aspect of their work is the *preventive memory management*, which considers invoking GC at each of the outer most phase instances (loops or functions). Within a phase, the technique lets the heap grow unimpeded. Currently, the relationship between our allocation pauses and their program phases is still unclear. We suspect that pauses resulting in large live-size reductions may be the same as their outer phases. However, their scheme may not detect smaller live-size reductions in the inner phases. It is also unclear how their technique can be integrated with generational collectors (i.e. would their technique invoke minor or full upon entering outer phases).

8. Conclusion

We introduce *MicroPhase*, a GC triggering technique that considers GC efficiency, instead of heap usage, as the main invocation criterion. Thus, MicroPhase can alleviate two existing shortcomings associated with the space-based approach: untimely GC invocations and large volumes of surviving newly created objects. The foundation of MicroPhase is based on two observations commonly found in Java programs: (i) allocation requests occur in phases, and (ii) the phase boundaries coincide with times that most objects also die.

As part of the implementation of MicroPhase, we extended HotSpot to support two additional mechanisms, pause detection and dynamic GC triggering. The pause detection mechanism is sampling-based and can accurately distinguish between actual allocation pauses from other runtime delays. The dynamic triggering mechanism leverages allocation site information to select whether minor collection or full collection should be invoked when a pause is detected. We evaluated the performance of MicroPhase using a collection of 20 Java benchmarks. The experimental results indicate that our technique can reduce the GC times in 19 applications, ranging from an increase of 1% to a decrease of 26% when the heap is set to be twice the maximum live-size. As a result, the overall performance improvement ranges from a degradation of 2.5% to an improvement of 14% in 13 applications.

9. Acknowledgments

This work was sponsored in part by the National Science Foundation through awards CNS-0411043 and CNS-0720757, and by the Army Research Office through DURIP award W911NF-04-1-0104. We also thank the anonymous reviewers for providing insightful comments for the final version of this paper.

References

[1] D. F. Bacon, P. Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *Proceedings of the 2003 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 81–92, San Diego, California, USA, June 2003.

[2] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 258–268, Montreal, Quebec, Canada, June 1998.

[3] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Eliot, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and

analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 169–190, Portland, Oregon, USA, 2006.

[4] S. M. Blackburn, R. E. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceedings of the ACM SIGPLAN Programming Languages Design and Implementation (PLDI)*, pages 153–164, Berlin, Germany, 2002.

[5] S. M. Blackburn, S. Singhai, M. Hertz, K. S. McKinely, and J. E. B. Moss. Pretenuring for Java. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 342–352, Tampa Bay, FL, October 2001.

[6] T. Brecht, E. Arjomandi, C. Li, and H. Pham. Controlling garbage collection and heap growth to reduce the execution time of Java applications. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages, and Applications (OOPSLA)*, pages 353–366, Tampa Bay, FL, USA, 2001.

[7] D. Buytaert, K. Venstermans, L. Eeckhout, and K. De Bosschere. Garbage collection hints. In *Proceedings of the First International Conference on High Performance Embedded Architectures and Compilers (HiPEAC 2005)*, pages 233–248, Barcelona, Spain, 11 2005. Springer Verlag.

[8] W. Chen, S. Bhansali, T. Chilimbi, X. Gao, and W. Chuang. Profile-guided proactive garbage collection for locality optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (OOPSLA)*, pages 332–340, Ottawa, Ontario, Canada, 2006.

[9] P. Cheng and G. E. Blelloch. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 125–136, Snowbird, Utah, USA, 2001.

[10] S. Dieckmann and U. Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 92–115, Lisbon, Portugal, June 1999. Springer Verlag.

[11] C. Ding, C. Zhang, X. Shen, and M. Ogihara. Gated memory control for memory monitoring, leak detection and garbage collection. In *Proceedings of the Workshop on Memory System Performance (MSP)*, pages 62–67, Chicago, Illinois, 2005.

[12] T. Domani, G. Goldshtein, E. K. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald. Thread-local heaps for Java. *SIGPLAN Notices*, 38(2 supplement):76–87, 2003.

[13] S. Z. Guyer, K. S. McKinley, and D. Frampton. Free-me: a static analysis for automatic individual object reclamation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 364–375, Ottawa, Ontario, Canada, 2006.

[14] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: Understanding the behavior of object-oriented applications. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 251–269, Vancouver, British Columbia, Canada, October 2004.

[15] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Error-free garbage collection traces: How to cheat and not get caught. In *Proceedings of the 2002 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 140–151, Marina Del Rey, California, 2002.

[16] IBM. Jikes Research Virtual Machine. http://jikesrvm.sourceforge.net.

[17] R. Jones and R. Lins. *Garbage Collection: Algorithms for automatic Dynamic Memory Management*. John Wiley and Sons, 1998.

[18] P. Mikhalenko. Real-time Java: An introduction. On-Line Article, Last visited: July 2007. http://www.onjava.com/pub/a/onjava/2006/05/10/real-time-java-introduction.html.

[19] P. Nagpurkar and C. Krintz. Visualization and analysis of phased behavior in Java programs. In *Proceedings of the ACM International Conference on the Principles and Practice of Programming in Java (PPPJ)*, Las Vegas, Nevada, USA, 2004.

[20] P. Nagpurkar, C. Krintz, M. Hind, P. F. Sweeney, and V. T. Rajan. Online phase detection algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 111–123, Manhattan, NY, USA, 2006.

[21] Y. Qian, W. Huang, W. Srisa-an, and J. M. Chang. Allocation Pattern and GC Triggering. Technical Report TR-UNL-CSE-2003-0017, University of Nebraska–Lincoln, Lincoln, Nebraska, U.S.A., October 2003. http://lakota.unl.edu//facdb/TechReportArchive/TR-UNL-CSE-2003-0017.pdf.

[22] Y. Qian, W. Srisa-an, T. Skotiniotis, and J. M. Chang. A cycle-accurate per-thread timer for Linux operating system. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 38–44, Tucson, Arizona, USA, November 2001.

[23] S. G. Robertz and R. Henriksson. Time-triggered garbage collection: robust and adaptive real-time GC scheduling for embedded systems. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 93–102, San Diego, California, USA, 2003.

[24] R. Shaham, E. K. Kolodner, and M. Sagiv. On effectiveness of GC in Java. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 12–17, Minneapolis, Minnesota, United States, 2000.

[25] J. Shemitz. Using RDTSC for benchmarking code on Pentium computers, Last visited: July 2007. http://www.midnightbeach.com/jon/pubs/rdtsc.htm.

[26] Standard Performance Evaluation Corporation. SPECjbb2000. White Paper, Last visited: July 2007. http://www.spec.org/osg/jbb2000/docs/whitepaper.html.

[27] Standard Performance Evaluation Corporation. SPECjvm98 benchmarks, Last visited: July 2007. http://www.spec.org/osg/jvm98.

[28] B. Steensgaard. Thread-specific heaps for multi-threaded programs. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 18–24, Minneapolis, Minnesota, United States, 2000.

[29] D. Stefanović, M. Hertz, S. M. Blackburn, K. S. McKinley, and J. E. B. Moss. Older-first garbage collection in practice: Evaluation in a Java virtual machine. *SIGPLAN Notices*, 38(2 supplement):25–36, 2003.

[30] D. Stefanović, K. S. McKinley, and J. E. B. Moss. Age-based garbage collection. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 370–381, Denver, Colorado, United States, November 1999.

[31] Sun Microsystems. Java technology is everywhere, surpasses 1.5 billion devices worldwide. Press Release, February 2004. http://www.sun.com/smi/Press/sunflash/2004-02/sunflash.20040219.1.html.

[32] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, 1984.

[33] P. R. Wilson. Opportunistic garbage collection. *ACM SIGPLAN Notices*, 23(12):98–102, 1988.

[34] P. R. Wilson and T. G. Moher. Design of the opportunistic garbage collector. *ACM SIGPLAN Notices*, 24:23–35, 1989.

[35] F. Xian, W. Srisa-an, C. Jia, and H. Jiang. AS-GC: An efficient generational garbage collector for Java application servers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 126–150, Berlin, Germany, July 2007.

[36] F. Xian, W. Srisa-an, and H. Jiang. Investigating the throughput degradation behavior of Java application servers: A view from inside the virtual machine. In *Proceedings of the 4th International Conference on Principles and Practices of Programming in Java (PPPJ)*, pages 40–49, Mannheim, Germany, 2006.

[37] K. Zee and M. Rinard. Write barrier removal by static analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 191–210, Seattle, Washington, USA, 2002.