Controlling Garbage Collection and Heap Growth to Reduce the Execution Time of Java Applications

Tim Brecht, Eshrat Arjomandi, Chang Li, and Hang Pham

Technical Report CS-2000-04

October 2000

Department of Computer Science

4700 Keele Street North York, Ontario M3J 1P3 Canada

Controlling Garbage Collection and Heap Growth to Reduce the Execution Time of Java Applications [*]

Tim Brecht
Department of Computer Science
University of Waterloo,
Waterloo, Ontario N2L 3G1
brecht@cs.uwaterloo.ca

Eshrat Arjomandi, Chang Li, and Hang Pham
Department of Computer Science
York University,
Toronto, Ontario M3J 1P3
{eshrat,changli,hangp}@cs.yorku.ca

Abstract

In systems that support garbage collection a tension exists between collecting garbage too frequently and not collecting garbage frequently enough. Garbage collection that occurs too frequently may introduce unnecessary overheads at the risk of not collecting much garbage during each cycle. On the other hand, collecting garbage too infrequently can result in applications that execute with a large amount of virtual memory (i.e., with a large footprint) and suffer from increased execution times due to paging.

In this paper we use a large collection of Java applications and the highly tuned and widely used Boehm-Demers-Weiser conservative garbage collector to experimentally examine the extent to which the frequency of garbage collection impacts an application's execution time, footprint, and pause times. We use these results to devise some guidelines for controlling garbage collection and heap growth in a conservative garbage collector in order to minimize application execution times. Then we describe new strategies for controlling garbage collection and heap growth that impact not only the frequency with which garbage collection occurs but also the points at which garbage collection occurs. Experimental results demonstrate that when compared with the existing approach our new strategy can significantly reduce application execution times.

1 Introduction

In many programming languages (e.g., Pascal, C, and C++) dynamically allocated memory must not only be tracked by the programmer but it must also be freed when it is no longer needed. Tracking and freeing dynamically allocated memory is time consuming and error-prone. In Java and other languages (e.g., Lisp, Smalltalk, ML, Self, Modula-3, and Eiffel) the run-time system keeps track of memory (objects) that has been dynamically allocated and periodically frees that memory which is no longer being used (i.e., it automatically performs garbage collection). In most existing Java implementations garbage collection is performed synchronously. That is, the executing program is suspended for a period of time while garbage collection is performed. Alternatively, some approaches to garbage collection attempt to simultaneously execute the garbage collector code and the main application by using a separate thread of control for garbage collection. However, the suspension of the main application or even uncontrolled delays due to thread switching can cause serious problems for users or other programs attempting to interact with the application.

Time spent reclaiming memory that is no longer in use typically delays the execution of the application and as a result can increase the execution time of the application. A tension is therefore created between collecting garbage too frequently and not collecting garbage frequently enough. Garbage collection that occurs too frequently may introduce significant and unnecessary overheads at the risk of not collecting much garbage during each collection. On the other hand, collecting garbage too infrequently can lead to larger heap sizes and increased execution times due to paging.

A memory allocation and garbage collection subsystem is faced with a number of fundamental decisions:

1. When allocating memory, what algorithm should be used?
2. If garbage collection is performed, which algorithm should be used?
3. When should garbage collection be performed?
4. When should the heap be expanded and by how much should it expand?
5. If the heap is being compacted, when should it be compacted?

[*]This work was partially supported by the IBM Toronto Labs Center for Advanced Studies and by NSERC.

Some of these decisions can have a significant impact on the frequency with which garbage collection occurs and on the overhead incurred in performing garbage collection. Much research has examined questions 1 and 2 above. For surveys see Wilson [16], Wilson *et al.* [17], and Jones and Lin [11]. Recent work [9] suggests that no one garbage collector is best suited for all applications. In this paper we concentrate on questions 3 and 4 and evaluate their impact on application performance in the context of the highly tuned and widely used Boehm-Demers-Weiser (BDW) [7] [6] [4] [3] conservative, mark-sweep collector. Our goal is to gain a better understanding of the impact of these decisions on application behavior and to examine techniques for scheduling garbage collection and heap growth. Although we are not aware of any published work that specifically studies question 5 above, we don't consider this question here because the BDW implementation used is not a compacting collector.

Garbage Collector Performance

Three main metrics that arise naturally from how garbage collection impacts an application and its execution are: the overall execution time of the application; the pause times introduced due to garbage collection (typically the measures of interest are the total, average, and maximum pause times); and the footprint of the application.

In this paper we concentrate on minimizing the execution time of an application. Execution time in some ways includes components of the other two metrics because pause times that are large will increase application execution times and applications with large footprints are more likely to incur overheads due to paging. While we don't believe that this is the only metric of importance, we believe that it is an important metric to a large number of users and that it represents an important starting point when optimizing garbage collector performance.

In this work we also focus on applications executing in isolation. We believe that it is first necessary to understand how these decisions impact a single application in order to develop and study techniques designed for environments where multiple applications execute simultaneously (which we also plan to study in future work).

2 Experimental Environment

All experiments were conducted on a 400 MHz Pentium II with 16 KB of level 1 instruction and data cache and 512 KB of unified level 2 cache. The operating system is NT Version 4.0 service pack 3 which uses a 4 KB page size. We use IBM's High Performance Java (HPJ) which translates Java-byte codes of whole programs into native machine instructions and provides the run-time system (including the garbage collector). Although the system we used contains 256 MB of memory, we configure the amount of memory used by the system at boot time. Since many of the Java benchmarks do not consume large amounts of memory this permits us to shrink the amount of memory in the system in order to place higher demands on the virtual memory subsystem.

2.1 The Applications

The Java applications used in our experiments were obtained from several sources including SPECJVM98 benchmarks (we exclude `_200_check` which is a synthetic benchmark designed to check features of the JVM).[1] They cover a wide range of application areas including: virtual machine benchmark programs, language processors, database utilities, compression utilities, artificial intelligence systems, multimedia, graphics, and object broker applications. In our experiments all explicit requests to `java.lang.System.gc()` are ignored in order to ensure that garbage collections are only scheduled by the algorithm being tested. Fitzgerald and Tarditi [9] report that the SPECJVM98 benchmarks run faster in their environment when they disregard these calls. Table 1 provides a brief description of each of the applications used in our experiments. (The SPECJVM98 applications have been studied and described in detail in other work [12] [14] [9].)

We use this relatively large collection of Java applications to evaluate the original BDW collector (using a variety of configurations) and compare their execution time to our new approach to controlling garbage collection and heap growth. At this point we have made no effort to eliminate applications that behave similarly or that are not impacted by garbage collection.

3 The BDW Collector

In this paper we use version 4.11 of the highly tuned and widely used Boehm-Demers-Weiser (BDW) conservative, mark-and-sweep garbage collector to study how garbage collection frequency impacts the execution of several applications. This collector was originally designed for use with C and C++ programs where information regarding pointer locations is not known by the collector at runtime. As a result, any reachable location in memory that contains a bit pattern that could be interpreted as a pointer to heap memory must conservatively be considered a pointer

[1]Two data set sizes are include for each application `-s10` and `-s100`, which are denoted by appending `.10` or `.100` to the application name.

Application	Description
compress *	A data compression utility that implements a modified version of a compression technique known as LZW (`_201_compress`)
db *	Performs multiple database functions on a memory resident database (`_209_db`)
espresso	A compiler that translates Java programs using a subset of the language into byte code
fred	Application frame work editor
jack *	A Java parser generator based on Purdue Compiler Construction Tool Set (PCCTS) (`_228_jack`)
jacorb	An object broker system based on OMG's Common Object Request Broker Architecture
javac *	Common Java compiler JDK1.0.2 (`_213_javac`)
javacup	A parser generator which generates parser code in Java
javalex	A lexical analyzer generator for Java
javaparser	A parser generator for Java
jaxnell	Generates tokenizers from regular expressions, and parser generator that generates recursive descent parsers from LL(1) grammars.
jess *	An expert shell system based on NASA'a CLIP expert shell system (`_202_jess`)
jgl	A Java virtual machine benchmark that performs array operations and sorting to test the performance of Java virtual machine.
jobe	A Java Obfuscation tool that scrambles Java Byte code to prevent the reverse engineering of the byte code
jolt	A Java byte code to C translator
mpegaudio *	This application decompresses audio files that conform to the ISO MPEG Layer-3 audio specifications (`_222_mpegaudio`)
mtrt *	A ray tracer that works on a dinosaur scene (`_227_mtrt`)
netrexx	A new programming language written in Java
toba	Translates Java class files to C

Table 1: List of benchmark Java programs used in our experiments, * denotes SPECJVM98 application.

to reachable memory. Additionally, heap compaction is not supported.

The BDW collector has been used to form the basis for Geodesic's REMIDI product [10]; integrated with the Apache web servers running Amazon.com; and used in a number of Java environments including the GNU Java compiler (gcj) and IBM's HPJ environment used in this study. We now briefly describe those aspects of this garbage collector that are relevant to our study.

The marking phase starts by marking all memory that can be accessed (reached) by the application. The algorithm begins with objects in registers, on the stack and in static variables and then recursively marks all objects that can be reached from the original (root) set of objects. Upon completion of the marking phase, unmarked objects that can not be reached are considered garbage and are reclaimed during the sweep phase. The system supports a distinction between atomic objects (those not containing pointers) and composite objects (those containing pointers) and only composite objects are traced during the marking phase. Our implementation is able to distinguish composite and atomic objects. Further, in order to reduce pause times, an initial sweeping reclaims only blocks consisting completely of unmarked objects. A lazy sweep technique is used during allocation to incrementally sweep remaining objects as they are needed. As a result, garbage collection times should be correlated with the size of the set of reachable composite objects and not the size of the heap (we've found this to be true in our experiments).

In the BDW collector the decision regarding whether or not to garbage collect is significantly influenced by a statically defined variable, called the *free space divisor* (FSD). Figure 1 shows a simplified version of the algorithm used in the BDW collector to decide whether to collect garbage or to grow the heap (i.e., the algorithm used to schedule garbage collections). This portion of the code is invoked when the memory allocator fails to find a suitable chunk of memory for the object (being allocated by the application). A check is made to determine what portion of the current heap is being used. Namely, if the portion of the heap that is used by the application is sufficiently large when compared with the reciprocal of the FSD value, then the garbage collector is invoked. For example, if the FSD is 2 then garbage is collected if more than roughly one half of the heap is used and if the FSD is 4 then garbage is collected if more than roughly one quarter of the heap is used. If the portion of the heap that is being used by the application is lower than the threshold determined by the FSD value then the heap is grown.

Note that the FSD is also used when the heap is grown. In this case it is used to determine how much to grow the heap by. So modifications to the FSD impact two decision points: whether to garbage collect or not and how much to grow the heap by. The amount by which the heap is grown also impacts garbage collection frequency since growing the heap by a large amount can postpone the need for garbage collection.

```
if (mem_used >= (heap / FSD)) {
  collect_garbage()
} else {
  grow_heap_by((heap / FSD) +
               request_size)
}
```

Figure 1: Simplified pseudo-code for the BDW collector that impacts scheduling.

3.1 BDW Experiments

Table 2 illustrates the impact of garbage collection frequency (including turning garbage collection off) on the execution of three Java applications. This subset of applications was chosen from our larger set of Java applications in order to illustrate the variety of affects that garbage collection frequency can have on the application. Each application is described in Section 2.1 and we consider the full set of applications later in the paper.

The experiments were conducted using 64 MB of memory so that some of the applications are using a reasonable portion of memory. Once the operating system and associated applications are loaded there is roughly 45-50 MB of memory available for the application. Each experiment was run 15 times and Table 2 shows observed averages.

Using the BDW conservative garbage collector we change the frequency with which garbage is collected by modifying the statically defined free space divisor (FSD). The first column shows the algorithm used to control garbage collection and heap growth (when garbage collection is off we grow the heap as though an FSD value of 4 is used). The remaining columns show the: execution time including 90% confidence intervals (Runtime); number of garbage collections (GCs); total time spent in the garbage collector (GC time); average time spent per garbage collection (Avg Pause); maximum time spent on one garbage collection (Max Pause); average footprint (Avg Foot)[2]; the total number of page faults (Faults); and the number of page faults that occurred during garbage collection (GC-faults). Although in all cases at least one garbage collection is reported, the first call is for initialization purposes only and no garbage is collected.

[2]This is obtained by post processing the amount of un-reclaimed memory over 10,000 points during the application's execution and taking the average over those points. Points of execution are determined based on the number of bytes allocated to ensure that samples are taken at the same points in the application's execution no matter which algorithm is used and how execution time is impacted.

As can be seen in Table 2 the `fred` application executes fastest when no garbage collections are performed. As the frequency of garbage collection increases the execution times and total garbage collection times increase significantly. With an FSD value of 16 the application runs slower than without garbage collection by a factor of about 3.8. In this case the 41 garbage collections take a total of 4471 milliseconds. Adding this to the execution time of the application without garbage collection (1650) nearly completely accounts for the extra execution time. With an FSD value of 4, the default configuration for the BDW code, it executes 1.8 times slower than without garbage collection (again the extra time spent in the garbage collector nearly completely accounts for the difference).

Interestingly, the average (and to a lesser extent the maximum) garbage collection times increase as the frequency of garbage collection increases. This is contrary to the notion that more frequent garbage collections might result in less garbage being collected during each collection and would therefore result in faster garbage collections. This is because the amount of reachable composite data in this application grows during execution; in the BDW collector tracing reachable composite objects accounts for the significant portion garbage collection time. This is accomplished by differentiating atomic objects from composite objects and only tracing composite objects and by utilizing a lazy-sweep technique that efficiently sweeps objects during allocation (when they are next allocated).

The `fred` application executes fastest when garbage collection is turned off because this application can execute within the memory available in the system (with garbage collection off the heap grows to 33 MB). However, as can be seen for applications with larger memory requirements like `db.100` and `javac.100`, turning garbage collection off can significantly degrade performance. Both applications execute slowest when garbage collection is turned off. In the case of `db.100` the slowest execution time is about 1.2 times slower than the fastest execution time which is obtained when an FSD value of 2 is used. In the case of `javac.100` the slowest execution time is about a factor of 7 times slower than the fastest execution time (which is obtained when an FSD value of 8 is used).

Unlike the `fred` application, in which average garbage collection times grow as the frequency of garbage collection increases, the maximum and average pause times for `db.100` are relatively unaffected by garbage collection frequency. This is because in `db.100` the total size of the reachable composite objects is relatively stable throughout the execution of the program (around 800 KB during all but the first few collections). Therefore, increasing the

Alg	Runtime			GCs	GC time	Avg Pause	Max Pause	Avg Foot	Faults	GCfaults
fred										
Off	1650	+/-	4.5	1	1	1	1	12199	7547	24
FSD 1	1661	+/-	6.1	1	0	0	0	12199	7547	24
FSD 2	2170	+/-	3.0	7	501	71	237	3456	5756	33
FSD 4	3030	+/-	5.0	16	1334	83	302	1978	5249	35
FSD 8	4220	+/-	4.7	26	2510	96	323	1457	5155	35
FSD 16	6236	+/-	3.4	41	4471	108	320	1138	5051	35
db.100										
Off	67619	+/-	854.2	1	17	17	17	56246	35353	24
FSD 1	68968	+/-	1385.2	1	19	19	19	56246	35272	24
FSD 2	55471	+/-	6.1	14	2267	161	224	6862	7682	42
FSD 4	57039	+/-	5.9	24	3926	163	220	4330	6355	45
FSD 8	62719	+/-	6.5	52	9509	182	223	2709	5462	45
FSD 16	69181	+/-	11.8	89	15979	179	220	2170	5171	45
javac.100										
Off	430507	+/-	41188.8	1	19	19	19	137306	171688	24
FSD 1	431476	+/-	27102.9	2	16526	8262	16502	88601	155440	3099
FSD 2	310998	+/-	13403.7	17	87263	5022	49631	16921	75594	15898
FSD 4	68399	+/-	6883.6	37	21731	591	7253	8660	20056	2063
FSD 8	61553	+/-	281.6	65	27404	415	1126	5931	13632	276
FSD 16	86362	+/-	1769.7	120	54401	450	944	4363	11576	63

Table 2: Impact of garbage collection frequency in BDW with 64 MB system; times are in milliseconds and sizes are in KB

frequency of garbage collections will increase the average pause time (until collections are so frequent that the asymptote is reached). The `db.100` application executes fastest when an FSD value of 2 is used. Here a `sweet spot` is obtained. Garbage collection is frequent enough that paging overheads are relatively low but not so frequent that overheads due to collection would negatively impact execution time. Without garbage collection the heap grows to about 124 MB, while the 14 garbage collections performed when FSD = 2 limits the heap growth to about 21 MB.

When executing `javac.100`, average garbage collection times decrease significantly as garbage collection frequency increases (up to FSD = 8) even though the size of the reachable set of composite objects is mainly increasing during execution. In this case, garbage needs to be collected frequently enough to permit the application to execute within the amount of memory available. More frequent collection keeps the footprint smaller and reduces the number of page faults that are incurred both during the execution of the program and during garbage collection (for FSD = 2, 4, 8, and 16). Note however, that once the footprint of the application is reduced to the point where it fits within the amount of memory available, which occurs when FSD = 8, more frequent collections increase execution time (when FSD = 16). It is worth pointing out that this is also the only application that incurs a real garbage collection when FSD = 1 (besides the one that is incurred to initialize the collector). Since the first collection is really only a quick initialization phase and only two collections are performed, the reported average pause time in this case (8262 ms) is a bit misleading. The actual pause time for the one real collection is 16502 ms (as can be seen in the column labelled Max Pause).

The results in Table 2 demonstrate that for the Java applications shown, the frequency with which garbage is collected can have a substantial impact on their execution and that a sweet spot exists in terms of minimizing execution times. Additionally, we see that for the BDW collector no one FSD value works best for all applications and that increasing the frequency of garbage collection does not appear to reduce the time spent in garbage collection for some applications.

Table 3 shows the results of the same experiments conducted on a system with 128 MB of memory (rather than 64 MB as in the previous experiments). Because the algorithm being used does not take into account the memory available in the system (it is based on the size of the heap), the garbage collection frequency is unchanged when compared with the 64 MB case. Consequently, the results for `fred` are unchanged, since it can easily execute within the available memory even without garbage collection. In

Alg	Runtime			GCs	GC time	Avg Pause	Max Pause	Avg Foot	Faults	GCfaults
fred										
Off	1633	+/-	5.6	1	5	5	5	12199	7546	24
FSD 1	1662	+/-	11.8	1	5	5	5	12199	7546	24
FSD 2	2180	+/-	14.5	7	498	70	236	3456	5755	33
FSD 4	3022	+/-	8.2	16	1329	82	302	1978	5249	35
FSD 8	4220	+/-	7.2	26	2500	96	322	1457	5155	35
FSD 16	6224	+/-	6.8	41	4470	108	320	1137	5050	35
db.100										
Off	56011	+/-	174.8	1	32	32	32	56246	31749	24
FSD 1	55924	+/-	124.5	1	24	24	24	56246	31772	24
FSD 2	55345	+/-	4.3	14	2259	161	223	6862	7681	42
FSD 4	56916	+/-	4.1	24	3918	163	219	4330	6354	45
FSD 8	62588	+/-	3.9	52	9501	182	225	2709	5461	45
FSD 16	69033	+/-	5.1	89	15949	179	219	2170	5170	45
javac.100										
Off	399141	+/-	7286.2	1	24	24	24	137320	190045	24
FSD 1	264869	+/-	3400.5	2	439	219	402	88588	133386	42
FSD 2	37959	+/-	170.7	17	6937	391	1022	15788	19328	67
FSD 4	45374	+/-	112.0	37	14284	385	1001	8958	16787	62
FSD 8	58524	+/-	240.0	65	27196	416	975	5968	13369	72
FSD 16	86473	+/-	1647.9	120	54664	451	945	4366	11634	69

Table 3: Impact of garbage collection frequency in BDW with 128 MB system; times are in milliseconds and sizes are in KB

the case of the `db.100` application all measured aspects of the application are unchanged relative to the 64 MB case (within confidence intervals), except the execution time of the application. While the sweet spot is still observed to occur when FSD = 2, we see that the execution time is only slightly better than when garbage collection is turned off.

Fairly significant and important differences are seen in the execution of the `javac.100` application. When compared with the 64 MB case, the execution time is substantially reduced in many of the cases. However, the application now executes fastest when an FSD value of 2 is used (38.0 seconds), as compared with the 64 MB case when a best execution time of 61.6 seconds is obtained using FSD = 8.

Interestingly, when FSD = 1 is used, the overhead incurred by the one real garbage collection is significantly lower in this case than when executed on a system with 64 MB of memory. In both cases the collection is triggered when the heap size is 64 MB (recall that about 45-50 MB is available for the application) so the heap has exceeded the amount of memory available in the system. In this case the reachable composite objects can be traced without incurring many page faults (42 faults are incurred during collections) while a total of 3099 faults are incurred during collection in the case when a 64 MB system is used.

In a system with 128 MB of memory the differences in maximum and average garbage collection times as the collection frequency increases for FSD = 2, 4, 8, and 16 are not nearly as dramatic as in the 64 MB case. In fact when garbage collection is less frequent (but not so infrequent as to cause paging) average and maximum pause times are actually equal to or lower than when collection is more frequent. For this reason the algorithm we develop in the next section is able to postpone garbage collection without incurring substantial costs (provided it isn't deferred too long).

When comparing the results in Table 2 with those in Table 3 we see that for some applications the best FSD value changes with the amount of memory available in the system. This motivates us to develop a technique that considers the memory available in the system in order to attempt to execute each application at its sweet spot.

4 A New Approach

In analyzing the results obtained from the experiments conducted in the previous section combined with lessons learned from experiments in which we attempted to produce an improved algorithm, we have developed some guidelines that we use in our new scheduling algorithm:

1. If there is sufficient memory available, garbage should not be collected and the heap should be grown quite aggressively.
2. As the amount of available memory becomes low we attempt to keep some memory available in order to avoid paging if possible. This is done by more aggressively (i.e., more frequently) collecting garbage and less aggressively growing the heap (i.e., growing by smaller amounts).
3. When the amount of available memory is low the initiation of garbage collection can become too aggressive. Therefore, methods are required for ensuring that frequency is tempered. We accomplish this by tracking the amount of memory reclaimed on recent collections and not collecting if recent collections do not reclaim a sufficient amount of memory.

As mentioned earlier, a significant problem with using the FSD to control garbage collection and heap growth (and approaches used in other garbage collectors) is that the amount of memory available in the system is not considered. Our new approach utilizes thresholds that are based on and determined relative to the amount of available memory.

When the memory allocator is unable to find a suitable block of memory in the existing heap for a new request, it must either garbage collect or grow the heap. When making this decision our modified runtime system determines the amount of memory available in the system and makes the decision based on: the amount of memory available; whether or not a threshold has been exceeded since the last garbage collection; and the amount of garbage collected during the recent garbage collections (to ensure that collections are not too frequent and that they are actually reclaiming memory).

Until the amount of memory used by the application exceeds the first threshold, garbage collection is effectively off. When any threshold is exceeded for the first time garbage is always collected. During a collection caused by exceeding threshold T_i the amount of memory reclaimed is calculated (R_i) and is used in deciding whether or not to collect garbage the next time threshold T_i is exceeded (or any other thresholds crossed by R_i). Garbage will be collected if a sufficient amount of garbage was collected during recent collections. If the amount of memory reclaimed crosses two or more thresholds, we collect the next time each of those thresholds is reached. In our current implementation we define a sufficient amount of memory that should be collected when crossing threshold T_i as follows:

$$S_1 = T_2 - T_1 \quad \text{for} \quad T_1 \text{ and}$$
$$S_i = T_i - T_{i-1} \quad \text{for} \quad T_i \text{ where } i > 1.$$

We have also added another decision point to our modified runtime system. This decision point considers whether or not garbage should be collected even if there is a considerable amount of free memory available in the current heap. This is considered important because in the BDW collector the heap size is never reduced and once a heap grows, all decisions are made with respect to that new heap size. Using the original FSD-based approach to controlling garbage collection and heap growth, if an application allocated a large amount of data (growing the heap to a point beyond available memory), even if a subsequent garbage collection reclaimed substantial amounts of memory, garbage collection would not be invoked again until an allocation request could not be satisfied from the existing heap. This can potentially result in paging when it might not be necessary.

This new decision point is carefully added to the allocator so as to limit its impact on the already highly optimized allocation code. We track the memory used by the application and when a threshold is passed we invoke the garbage collector if a recent collection reclaimed a sufficient amount of memory. Although this adds a few instructions to the allocation path it does not seem to impact the execution time of our applications in a noticeable way. In fact, this extra decision point is not included in the BDW version of our experiments and we have observed that we are typically able to obtain application execution times that are as low or lower than those obtained with the BDW version.

Finally, when growing the heap we grow quite aggressively, targeting a doubling of the heap size on each growth, until the heap size reaches the first threshold (care is taken to grow only up to the first threshold). Subsequent heap growths are done by growing to the next threshold. We propagate these targeted growth sizes through the original BDW code which ensures that the heap is grown by at least a minimum increment and no more than a maximum increment (256 KB and 16 MB respectively, as defined in the original version of our BDW implementation).

Figure 2 shows an example of how thresholds are used to control both garbage collection and heap growth. In this example each decision point is marked. A circle denotes that the heap was grown, a square denotes that garbage was collected and a diamond denotes that the decision was to do neither. The program starts with an initial heap (in our experiments we use the default initial size used in the BDW collector, 256 KB). As the program allocates memory the heap is grown as shown by points A and B. Once the amount of memory used by the application has reached threshold T_1 (at point C), the garbage collector will be invoked because the allocator is not able to find an appropriate block in the heap to satisfy the request. At point C garbage collection reclaims memory to point D.

The amount of memory collected from point C to D, R_1, is not considered large enough for garbage to be collected the next time T_1 is reached (because R_1 is less than S_1). Therefore, the next time the allocator fails to satisfy the current request (at point E) the heap is expanded up to the next threshold, T_2.

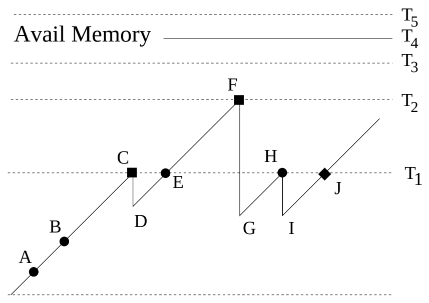


Figure 2: Example operation using thresholds

As the application continues to use memory, the collector will be invoked at point F because a new threshold, T_2, has been reached. This time a substantial amount of garbage is collected, reducing the amount of memory considered live to point G. This is considered to be a good collection because it collected more than $S_2 = T_2 - T_1$ bytes of memory. Therefore, the next time T_1 or T_2 are reached the garbage collector will be invoked. From this point onward the heap does not need to be expanded for some time because the heap size is at T_2 and it is never reduced (compaction is not implemented in the BDW collector).

As the program continues to allocate objects, the amount of memory being used will grow until point H is reached (again at T_1). Since our approach is to try to keep the amount of memory below each threshold (provided the recent collection reclaimed a sufficient amount of memory), garbage collection will be invoked at point H. Since not a lot of garbage is collected to reach point I, the next time T_1 is reached (at point J) the collector is not invoked.

In the example described above we differentiate garbage collection points C and F as being initiated as a result of the heap becoming sufficiently utilized and collection point H as being initiated by the allocator passing a threshold that we would like to avoid exceeding.

As can be seen in the example in Figure 2, we have chosen thresholds so that as less memory is available the thresholds are closer together. This permits us to collect garbage more aggressively (i.e., more frequently) and to grow the heap less aggressively (i.e., by smaller amounts) as the amount of memory available to the application decreases. Both of these actions are designed to avoid collection and heap growth overheads when there is an abundance of available memory. Moreover, they are designed to try to free up unused memory and limit heap growth in order to attempt to execute the application within the amount of memory available in the system. Bouncing above and below thresholds is prevented by ensuring that a sufficient amount of memory is reclaimed by the previous collection.

Naturally the choice of the number of thresholds and their locations can greatly influence application performance. While we would prefer a technique that did not require parameter tuning, we didn't find it difficult to choose a set of thresholds that works quite well across the set of applications used in our experiments. Our current implementation defines logical thresholds relative to the amount of memory available in the system at the time of collection. For the experiments conducted with 64 MB of memory we used logical thresholds of:

0.40, 0.55, 0.70, 0.85, 0.92, 1.00, 1.15, and 30.00.

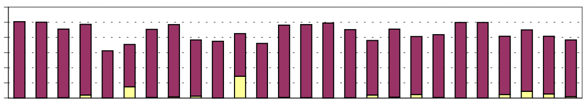
For experiments conducted with 128 MB of memory we used logical thresholds of:

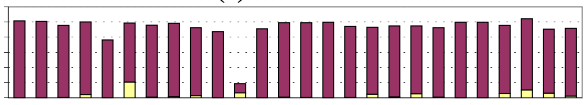
0.80, 0.85, 0.90, 0.95, 1.00, 1.05, and 10.00.

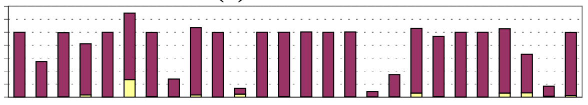
Note that each set of thresholds includes one threshold at the point equal to available memory and one slightly above. Again these are designed to attempt to collect enough garbage so that the program will execute with some memory available. However, we've set the next threshold to a point well beyond available memory (this point is not reached in any of our experiments). Preliminary experiments we have conducted suggest (and it has been pointed out in other work [12]) that it might be best not to garbage collect when the amount of reachable data is large (since the virtual memory subsystem will page out inactive data). We hope to conduct further research into this issue in the future.

4.1 Threshold-based Experiments

One of the goals of this work is to develop an approach to scheduling that results in faster application execution times than achieved by the existing approaches. Therefore, we begin by using an environment with 64 MB of memory and comparing our threshold-based approach with the original BDW algorithm with FSD = 4, FSD = 2, and with garbage collection turned off. These environments are examined explicitly because the majority of the applications used in our experiments execute very efficiently in one of these three scenarios. The results of these experiments are shown in Figure 3(a), 3(b), and 3(c), respectively. Additionally, in Figure 3(d) we compare the execution times obtained using our threshold based approach with the minimum execution times obtained across all FSD values.


(a) FSD = 4


(b) FSD = 2


(c) GC Off

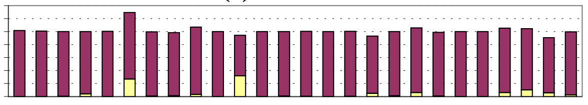

(d) Best times over all FSD values

Figure 3: Comparing threshold-based algorithm with BDW using different FSD values on a 64 MB system

In all graphs, the execution time obtained with our scheduling algorithm averaged over 15 runs is normalized with respect to the mean execution time over 15 runs of the application when executed using the standard BDW environment (for the FSD value used). The values are shown using the taller, dark colored bars. We use light colored bars to show the amount of time spent paused during garbage collection using the threshold-based approach, as a percentage of the execution time obtained using the threshold-based approach.

Figure 3(a) shows that the set of Java applications used executes quite well using the threshold-based approach when compared with the standard BDW approach using FSD = 4 (the default value). Using the threshold-based algorithm, many of the applications (10 of 26) execute in roughly 80% or less of the time required when using FSD = 4. Interestingly, one of these applications (`espresso`) benefits significantly and executes in roughly 60% or less of the time required when FSD = 4 is used. Moreover, none of the 26 applications runs slower using the threshold-based approach.

It is interesting to note that when using the threshold-based algorithm and executing with 64 MB of memory, overheads due to garbage collection are relatively small for all applications except `javac.100` (where it is about 33%) and `fred` (where it is about 21%). For `javac.100`, execution time is significantly improved especially when compared with FSD = 2 (Figure 3(b)) or turning garbage collection off (Figure 3(c)). We discuss the `fred` application in detail shortly.

The results observed in Figure 3(b), which compare the threshold-based algorithm with FSD = 2, are similar to those seen in Figure 3(a) (for FSD = 4) but they are not as dramatic. In this case, for all executions except `espresso` and `javac.100` the threshold-based algorithm and FSD = 2 yield execution times that are within approximately 5–10% of each other. However, we point out that when using the threshold-based approach `javac.100` executes in less than 20% of the time required to execute with FSD = 2. We also point out that when comparing the graphs in Figure 3(a) and 3(b), one could conclude that a default FSD value of 2 would better serve more of the applications in our test suite than the value of 4 used in the current BDW distribution. However, the value of 4 appears to be more conservative and prevents any one of the applications tested from suffering from very poor performance.

Figure 3(c) compares the execution times of the threshold-based algorithms with those obtained when garbage collection is off. By comparing these results with those in Figure 3(d) we can see that in our environment a number of applications execute most effectively when garbage collections do not interfere with the execution of the application. For almost all of these applications the threshold-based algorithm is able to ensure that the number of garbage collections is kept to a minimum and we see that execution times are as good as when no garbage collections occur (and in fact in many cases the garbage collector is not invoked after its initialization phase). However, for 8 of the 26 applications, the threshold-based approach provides significant reductions in execution time because without garbage collection these applications incur considerable overheads due to paging.

Unfortunately, one of the applications (`fred`) executes slower by a factor of 1.3 when using the threshold-based algorithm than when garbage collection is turned off (which is also when `fred` executes fastest). This is because `fred` is a very short running program and the two garbage collections initiated in the threshold-based algorithm take a total of 447 ms and inflate execution time by a factor close to 1.3. One possible remedy for this situation would be to take into account how long an application has been executing. The idea would be to further delay collections until an application has executed for a sufficient length of time (assuming that its rate of memory allocation is not too high). This would ensure that the cost of garbage collection is amortized over a longer period of time, rather than only over the amount of memory allocated as is currently done. We have not tried this approach yet.

As mentioned previously, one of the goals of the threshold-based approach is to perform at least as well as the best possible FSD value across all applications. That is, to perform enough garbage collections to avoid paging for those applications for which that is a problem and to avoid collecting garbage too frequently for those applications whose execution would be negatively impacted. Figure 3(d) compares the mean response time of each application with the mean response time obtained with the best standard BDW collection method (i.e., the minimum mean execution time obtained with garbage collection off and using FSD values of 1, 2, 4, 8, and 16).[3] This graph shows that our approach to controlling garbage collection and heap growth is roughly as good or slightly better than the best FSD value for all but one of the applications (`fred`).

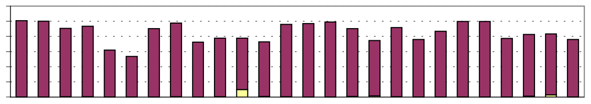
To demonstrate that our approach also works with different amounts of physical memory, we conduct the same set of experiments with 128 MB of memory and present a similar series of graphs in Figure 4. The observations here are similar to those made when executing using 64 MB, except that the threshold-based approach appears to be slightly more consistent than in the 64 MB case. In the 128 MB case all applications execute in time equal to or slightly better than the best possible FSD value (Figure 4(d)). Additionally, the portion of the execution time spent performing garbage collection is negligible in all applications except `javac.100` where it is now about 13%.

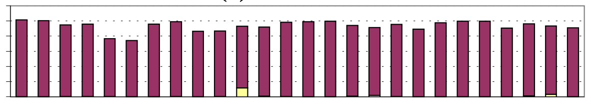
5 Related Work

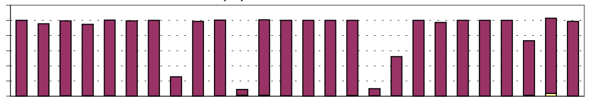
Moon [13] points out that users of some early Lisp machines found that garbage collection made interactive response time so poor that users preferred to turn garbage collection off and reboot once the virtual address space was consumed. He also demonstrates that some applications execute fastest with garbage collection turned off.

Appel [2] and Cooper, Nettles, and Subramanian [8] describe techniques for determining heap sizes in copying collectors. Unfortunately, neither study specifically examines the impact that their decisions have on application performance. Smith and Morrisett [15] describe a mostly-copying collector that collects garbage whenever the heap

[3]Of the applications tested none of the execution times decreased with FSD values of 32 or higher.


(a) FSD = 4


(b) FSD = 2


(c) GC Off

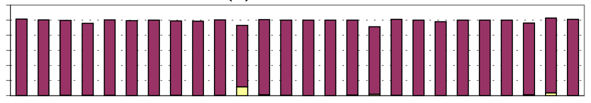

(d) Best times over all FSD values

Figure 4: Comparing threshold-based algorithm with BDW using different FSD values on a 128 MB system

is two-thirds full. This roughly corresponds to using an FSD value of 1.5 in the BDW collector and will suffer from the drawbacks described in Section 3.1.

Zorn [18] [19] points out that the efficiency of conservative garbage collection can be improved if more garbage can be collected during each collection phase and suggests that one way to achieve this is to wait longer between collections. However, he also warns that there is a tradeoff between the efficiency of collection and program address space. In addition, he describes a policy for scheduling garbage collection that is based on an “allocation threshold”. Namely, the collector runs only after a fixed amount of memory has been allocated (e.g., after every 2 MB of memory have been allocated).

A set of experiments conducted in our environment using Zorn’s allocation threshold approach yielded results that are similar to those obtained using the different FSD values for the BDW collector.[4] Namely, no one allocation threshold was suitable for all applications. Because the

[4]To be fair, Zorn devised this algorithm to fairly compare two different garbage collection algorithms while ensuring that the scheduling of garbage collections was done identically in each case and not to optimize any performance metric.

amount of memory available is not considered when initiating garbage collection, some applications exceed the amount of memory available and overheads due to paging significantly increase execution time. While a smaller threshold might reduce the execution time of such an application it can increase the execution time of other applications where the overhead due to frequent collections is significant.

Alonso and Appel [1] implement an "advice server" that is used to determine how to take maximum advantage of memory resources available to a generational copying garbage collector for ML. After each garbage collection the application contacts the advisor process to determine how it should adjust its heap size. The advisor process uses `vmstat` output to monitor the number of free pages and page fault rates in order to tell each application how to adjust its heap size. Although this is not explicitly stated in their paper, we believe that garbage collections only occur when the free space portion of the heap is exhausted (based on the ML implementation we believe they used [2]). As a result, control over garbage collection only occurs by modifying the size of the heap. Unfortunately this approach can not be deployed in the BDW collector, since the BDW implementation does not support shrinking the heap. Although it might be possible to modify the BDW collector to contact an advisor process when making decisions regarding garbage collection and heap growth, we believe that our approach obtains significant benefits by occasionally deciding to garbage collect even when there is sufficient memory available in the heap to satisfy a request. This is accomplished by carefully adding a simple and efficient check that occurs during allocation. We believe that the overhead incurred in contacting an external process to perform such a check would be highly detrimental to the performance of most applications. However, their work does demonstrate that garbage collection can be controlled in a number of applications executing simultaneously to provide reduced executions times.

Recently, Kim and Hsu [12] have analyzed the memory system behavior of several Java programs from the SPECJVM98 benchmark suite. One of the observations made in their work is that the default heap configuration used in IBM JDK 1.1.6 results in frequent garbage collection and the inefficient execution of applications. Although the direct overheads due to garbage collection in their environment appear to be more costly than in ours (because the entire heap is swept on each collection and because heap compaction is used), we believe that their results also demonstrate the need to improve techniques for controlling garbage collection and heap growth. They also point out that although the direct costs of garbage collection decrease as the available heap size is increased, there exists an optimal heap size which minimizes total execution time (due to interaction with the virtual memory subsystem).

One of the main differences between our work in this paper and previous work is that we specifically study and devise techniques for controlling garbage collection and heap growth and directly consider the tradeoff between execution time and application footprint.

6 Discussion

Decreased garbage collection times, whether they are achieved by improvements to the implementation we've used (e.g., by reducing cache misses during collections [5]) or by utilizing a different collector, may mean garbage collection can be invoked more frequently without negatively impacting the application's execution time. While we believe that this would make it easier to obtain a sweet spot in terms of collection frequency, we suspect that it would still be necessary to prevent collections from occurring too frequently.

Moreover, we've observed that in some cases the benefits obtained from using our threshold-based approach are a result not so much of controlling how frequently collections occur but of controlling the point at which they occur. For example, when executing `javac.100` using our threshold-based approach on a 64 MB system, an average of 24 collections are performed and a mean execution time of 58 seconds is observed. In contrast, with FSD = 2, 4 and 8 the collector is invoked an average of 17, 37 and 65 times respectively, while mean execution times are 311, 68 and 62 seconds respectively. Although the number of collections invoked using the threshold-based approach falls on a spectrum somewhere between FSD = 2 and FSD = 4, the execution time does not lie in the same spectrum. In fact it is slightly better than that obtained with the 65 collections performed when FSD = 8.

It also seems that garbage collection overheads are still sufficiently large that our approach might prove useful in combination with other garbage collectors. Recent experiments conducted by Fitzgerald and Tarditi [9] show that for one application (`cn2`) garbage collections account for a minimum of 30% of the total execution time across three different collectors used in their experiments. Additionally, garbage collections account for roughly 15–25% of the execution time for several combinations of applications and collectors. They also point out that for some of their applications reducing the number of garbage collections by half roughly halves the time spent in garbage collection.

Of course, our technique may not work with all garbage collectors. An unstated but underlying assumption is that delaying garbage collection will not significantly increase the time spent in garbage collection. While this is true in the BDW collector used in our experiments, this is not true for all garbage collectors. For example, the mark-and-sweep copying collector reported on by Kim and Hsu [12] incurs overheads proportional to the amount of garbage being collected. It is unclear what impact delaying garbage collection would have in such environments.

As discussed in more detail in Section 5, Alonso and Appel [1] demonstrate that garbage collection can be effectively controlled in a number of simultaneously executing applications. Although we intentionally focus on understanding how to minimize the run time of one application executing in isolation, we have tried to keep multiprogrammed environments in mind. In such cases the amount of available memory will be reduced and presumably thresholds will be reached sooner. However, such an environment might require us to dynamically adjust our thresholds because we have found that for some applications it is important to be more aggressive about collecting garbage when less memory is initially available. This is reflected in differences in thresholds used for 64 MB and 128 MB systems (*cf.* Section 4).

7 Conclusions

In this work we evaluate the performance of, compare, and design algorithms specifically to control the scheduling of garbage collection and heap expansion. Collectively we refer to these problems as the garbage collection scheduling problem.

We have conducted a detailed study of the impact of these scheduling decisions on the execution of several Java applications (26 different executions using 19 different applications) while using the highly-tuned and widely used conservative Boehm-Demers-Weiser (BDW) memory allocator and garbage collector. Using this environment we observe that:

- The execution times of many of the applications we tested varies significantly with the scheduling algorithm used for garbage collection.
- No one configuration of the BDW collector results in the fastest execution time for all applications. That is, no one FSD value can be used to obtain the fastest execution time for all applications. In fact choosing the wrong FSD value can significantly and unnecessarily increase the execution time of the application.
- The best scheduling algorithm for an application (the one which results in the fastest execution of an application) also varies with the amount of memory available in the machine on which the application is executing.
- Making decisions about whether to garbage collect or grow the heap based primarily on how much of the current heap is used is not a good choice when the heap does not shrink (as is the case in the BDW collector). We argue that in order to minimize the execution time of an individual application it is better to base such a decision on how much memory is currently available.

We also design, implement, and experimentally evaluate a threshold-based algorithm for controlling garbage collection and heap growth. We demonstrate that by using our new approach we are able to significantly reduce the execution time of many applications when compared with the method used by the standard BDW implementation. We believe that these benefits are obtained by taking into account the amount of memory available to the application when determining whether to collect garbage or to grow the heap and by considering the amount of memory reclaimed in previous collections.

We observe that with our threshold-based approach the execution times of the applications tested are (except for one case) at least as fast as the execution times obtained using the best possible scheduling in the standard BDW collector (i.e., over all FSD values).

In the future we plan to test our approach using different garbage collectors, and to consider multiple applications executing simultaneously, dynamic threshold values, and techniques that do not require tuning.

References

[1] R. Alonso and Andrew W. Appel. Advisor for flexible working sets. In *Proceedings of the 1990 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems. Boulder, May 22–25*, pages 153–162. ACM Press, 1990.

[2] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[3] Hans-Juergen Boehm. A garbage collector for C and C++. Hans Boehm's web page for his garbage collector, http://www.hpl.hp.com/-personal/Hans_Boehm/gc/.

[4] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, pages 197–206, Albuquerque, NM, June 1993. ACM Press.

[5] Hans-Juergen Boehm. Reducing garbage collector cache misses. In *International Symposium on Memory Management*, pages 59–64. ACM Press, October 2000.

[6] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

[7] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[8] Eric Cooper, Scott Nettles, and Indira Subramanian. Improving the performance of SML garbage collection using application-specific virtual memory management. In *Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming*, pages 43–52, San Francisco, CA, June 1992. ACM Press.

[9] R. Fitzgerald and D. Tarditi. The case for profile-directed selction of garbage collectors. In *International Symposium on Memory Management*, pages 111–120. ACM Press, October 2000.

[10] Geodesic Systems Inc. REMIDI. http://www.-geodesic.com/solutions/remidi.html.

[11] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[12] J. Kim and Y. Hsu. Memory system behavior of java programs: Methodology and analysis. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 264–274. ACM Press, June 2000.

[13] David A. Moon. Garbage collection in a large LISP system. In Guy L. Steele, editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–245, Austin, TX, August 1984. ACM Press.

[14] R. Shaham, E.K. Kolodner, and M. Sagiv. On the effectiveness of GC in Java. In *International Symposium on Memory Management*, pages 12–17. ACM Press, October 2000.

[15] Frederick Smith and Greg Morrisett. Comparing mostly-copying and mark-sweep conservative collection. In Richard Jones, editor, *Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 68–78, Vancouver, October 1998. ACM Press.

[16] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994.

[17] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Henry Baker, editor, *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, September 1995. Springer-Verlag.

[18] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Conference Record of the 1990 ACM Symposium on Lisp and Functional Programming*, Nice, France, June 1990. ACM Press.

[19] Benjamin Zorn. The measured cost of conservative garbage collection. *Software Practice and Experience*, 23:733–756, 1993.