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

TIM BRECHT
University of Waterloo
ESHRAT ARJOMANDI
York University
CHANG LI
IBM Toronto
and
HANG PHAM
University of Toronto

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 set of Java applications and the highly tuned and widely used Boehm-Demers-Weiser (BDW) conservative mark-and-sweep 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 used in the standard BDW collector, our new strategy can significantly reduce application execution times.

Our goal is to obtain a better understanding of how to control garbage collection and heap growth for an individual application executing in isolation. These results can be applied in a number of high-performance computing and server environments, in addition to some single-user environments. This work should also provide insights into how to make better decisions that impact garbage collection in multi-programmed environments.

A shorter version of this paper appeared in the Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 353-366, October, 2001. Tim Brecht: School of Computer Science, University of Waterloo, `brecht@cs.uwaterloo.ca`. Eshrat Arjomandi: Department of Computer Science and Engineering, York University, `eshrat@cs.yorku.ca`. Chang Li: Toronto Lab, IBM Canada Ltd., `changl@ca.ibm.com`. Hang Pham: Department of Computer Science, University of Toronto, `hangp@cs.utoronto.ca`. This work started while all four authors were at York University and some of it was conducted while Tim Brecht was employed by Hewlett-Packard Labs.

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

General Terms: Algorithms, Experimentation, Languages, Measurement, Performance

Additional Key Words and Phrases: garbage collection, heap growth, implementation, Java, memory management, performance measurement, programming languages,

1. INTRODUCTION

In many programming languages (e.g., Pascal, C, and C++), dynamically allocated memory must not only be tracked by the programmer but must also be freed when it is no longer needed. Tracking and freeing dynamically allocated memory is a time-consuming task performed by programmers who are error-prone. In Lisp and other languages (e.g., Java, Smalltalk, ML, Self, Modula-3, and Eiffel) the run-time system keeps track of memory (objects) that has been dynamically allocated and periodically frees the memory that is no longer being used (i.e., it automatically performs garbage collection). In some of these language implementations, and in particular several Java implementations (which is the focus of our work), 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 [Domani et al. 2000; Bacon et al. 2001; Ossia et al. 2002; Barabash et al. 2003; Azatchi et al. 2003]. However, the suspension of the main application 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 by 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 increased cache and TLB misses and paging.

A number of fundamental decisions must be made when implementing a memory-allocation and garbage-collection subsystem:

(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?

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 been focused on questions 1 and 2 above. For surveys of research related to this and other aspects of garbage collection see Wilson [1994], [Wilson et al. 1995], and Jones and Lins [1996]. Some research [Fitzgerald and Tarditi 2000; Attanasio et al. 2001] suggests that no one garbage collector is best suited for all applications. Additional work, Printezis [2001] proposes and studies a technique for hot-swapping between a mark-and-sweep and a mark-and-compact garbage collector. The simple heuristic proposed for deciding when to switch between the two collectors in effect considers a form of question 5 above.

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) [Boehm and Weiser 1988; Boehm 2004] conservative, mark-and-sweep garbage 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.

Garbage-Collector Performance

Three main metrics arise naturally from how garbage collection impacts an application and its execution: the overall execution time of the application; the pause times introduced due to garbage collection (typically, the measures of interest are the total, average, maximum, and distribution of 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 do not 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 for 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 memory-allocation and garbage-collection decisions impact a single application in order to develop and study techniques designed for environments where multiple applications execute simultaneously (which we plan to study in future work).

2. EXPERIMENTAL ENVIRONMENT

All experiments are conducted on a 400 MHz Pentium II system with a 16 KB level 1 instruction cache, a 16 KB level 1 data cache, and with 512 KB of unified level 2 cache. The operating system is Windows NT Version 4.0, service pack 3, which uses a 4 KB page size. We use IBM's High Performance Java (HPJ), which compiles 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. In practice, most JVMs are run with enough memory to avoid paging.

Table I. List of benchmark Java programs used in our experiments, where * denotes a SPECJVM98 application.

Application	Description
compress *	A data compression utility that implements a modified version of a compression technique known as LZW (`_201_compress`)
db *	A database utility that 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	An application framework editor
jack *	A Java parser generator based on the Purdue Compiler Construction Tool Set (PCCTS) (`_228_jack`)
jacorb	An object broker system based on OMG's CORBA (Common Object Request Broker Architecture)
javac *	Common Java compiler JDK1.0.2 (`_213_javac`)
javacup	A parser generator that generates parser code in Java
javalex	A lexical analyzer generator for Java
javaparser	A parser generator for Java
jaxnjell	A parser generator for Java that generates tokenizers from regular expressions and recursive descent parsers from LL(1) grammars
jess *	An expert system shell based on NASA's CLIP (`_202_jess`)
jgl	A Java virtual machine benchmark that performs array operations and sorting to test the performance of a 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 *	An application that 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	A Java class files to C translator

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 Java virtual machine).[1] Our collection of programs covers 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 scheduled only by the algorithm being tested. Fitzgerald and Tarditi [2000] report that the SPECJVM98 benchmarks run faster in their environment when they disregard these calls. Table I 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 [Kim et al. 2000; Shaham et al. 2000; Fitzgerald and Tarditi 2000].)

We use this relatively large collection of Java applications to evaluate the original

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

BDW collector (using a variety of configurations) and to compare each application's execution time to that obtained with our new approach to controlling garbage collection and heap growth. We have made no effort to eliminate applications that behave similarly or that are not impacted by garbage collection.

3. THE BDW COLLECTOR

We use version 4.11 of the Boehm-Demers-Weiser (BDW) garbage collector and memory allocator. The BDW collector was originally designed for use with C and C++ programs where information regarding pointer locations is not known by the collector at run time. 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 to reachable memory. Additionally, heap compaction is not supported.

The BDW collector has been used to form the basis for Geodesic's REMIDI product [Geodesic Systems Inc. 2002], 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. Because HPJ is used to compile Java to native machine code prior to executing the program, we believe that its performance is comparable to modern just in-time compilers. We now briefly describe those aspects of the BDW garbage collector that are relevant to our study.

The marking phase starts with the marking of all objects in memory that can be accessed (reached) by the application. The algorithm begins with root objects in registers, on the stack, and in static variables. It then recursively marks all objects that can be reached from the roots. Upon completion of the marking phase, unmarked objects that cannot 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 traces composite objects 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).

For the BDW collector, the decision regarding whether or not to collect garbage 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 decision is made when the memory allocator fails to find a suitable chunk of memory for the object being allocated by the application. A test is performed to see if the amount of memory allocated since the last garbage collection is greater than a portion of the heap. The spirit of this approach is to amortize the cost of a garbage collection by ensuring that the number of bytes allocated since the last collection is large enough to warrant a garbage collection. For example, if the FSD is 2, then garbage is collected if more than roughly half of the heap was allocated since the last garbage collection; if the

```
if (cannot_alloc_memory(request_size)) {
  if (alloced_since_last_GC $\geq$ (heap / FSD)) {
    collect_garbage()
  } else {
    grow_heap_by((heap / FSD) + request_size)
  }
}
```

Fig. 1. Simplified pseudo-code for the portion of the BDW collector that impacts scheduling.

FSD is 4, then garbage is collected if more than roughly one quarter of the heap was allocated since the last garbage collection. If the amount of memory allocated since the last garbage collection 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 or not to collect garbage and by how much to grow the heap. The amount by which the heap is grown also impacts garbage collection frequency since growing the heap by a large amount can influence the need for garbage collection.

3.1 BDW Experiments

Table II illustrates the impact that garbage collection frequency (including turning garbage collection off) has 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 effects that garbage collection frequency can have on an application. Each application is described in Table I 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. This is done by rebooting the machine so that is it configured to use the specified amount of memory. Once the operating system and associated applications are loaded, there is roughly 45 - 50 MB of memory available for the application. Note that the variations in available memory occur only occasionally and are due to the Windows NT implementation. Initially we found larger variations that occurred more frequently but we were able to reduce and control the variation by running an application between each experiment that allocates and uses substantially more memory than is physically available in the machine. This forced the operating system to try to take pages from other processes and allowed us to begin each experiment from approximately the same amount of available memory. In order to ensure that statistically significant differences in execution times are due to changes in the garbage collector (and not other factors like differences in the amount of available memory at the start of the execution) we use multiple executions and compute 90% confidence intervals on the execution times. Table II shows averages and 90% confidence intervals obtained over 19 runs. We executed twenty runs but one of the experiments for one run was tainted when someone used the machine.

Using the BDW collector, we change the frequency with which garbage is collected by modifying the statically defined FSD. For comparison, we also include results

Table II. Impact of garbage collection frequency using BDW with 64 MB system; times are in milliseconds and sizes are in MB. The row labelled "Tholds" shows the results obtained with our new algorithm. This algorithm and the results for it are discussed in Section 4.

Alg	Time	90%	GC	Total	Avg	Max	HG	Foot	pf	gcpf
					fred					
Off	1654	4	0	0	0	0	8	11.9	7547	0
FSD 1	1671	9	0	0	0	0	8	11.9	7547	0
FSD 2	2174	4	6	495	82	237	11	3.4	5756	9
FSD 4	3035	6	15	1331	88	302	17	1.9	5248	11
FSD 8	4227	6	25	2506	100	323	25	1.4	5155	11
FSD 16	6240	5	40	4472	111	320	38	1.1	5051	11
Tholds	2200	67	1	477	292	314	8	8.0	6150	9
					db.100					
Off	68823	1146	0	0	0	0	13	54.9	35709	0
FSD 1	71294	2111	0	0	0	0	13	54.9	35653	0
FSD 2	55480	9	13	2260	173	224	9	6.7	7682	18
FSD 4	57044	6	23	3922	170	220	13	4.2	6355	21
FSD 8	62727	8	51	9508	186	223	20	2.6	5462	21
FSD 16	69201	23	88	15985	181	221	30	2.1	5170	21
Tholds	55443	5	10	2250	225	231	7	6.7	7265	18
					javac.100					
Off	463472	40771	0	0	0	0	23	134.1	183910	0
FSD 1	452722	26595	1	16012	16012	16012	20	86.5	162761	2965
FSD 2	323418	14149	16	92065	5642	53188	14	16.7	77897	16446
FSD 4	74770	7268	35	24162	677	10012	21	8.6	21387	2513
FSD 8	61744	265	64	27548	424	1126	33	5.8	13677	253
FSD 16	87443	1607	120	55475	459	943	52	4.2	11525	39
Tholds	64130	4696	24	23145	958	3501	12	8.6	17535	1871

obtained using our new algorithm (labelled "Tholds"). The algorithm and these results are described and discussed in Section 4. In this section, we concentrate on understanding the impact that different FSD values have on program execution. We have chosen to run experiments with FSD values of 1, 2, 4, 8, and 16 because we found that these gaps resulted in a good range in the frequency of garbage collections. The FSD value is not required to be a power of two and is not required to be an integer.

The first column in Table II 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 mean execution time, including 90% confidence intervals (Time); number of garbage collections (GC); total time spent in the garbage collector (Total); average time spent per garbage collection (Avg); maximum time spent on one garbage collection (Max); number of times the heap was grown (HG); average footprint (Foot);[2] total page faults (pf); and page faults that occurred during garbage collection (gcpf).

[2]The average footprint is obtained by using a graph of the amount of live memory (based on how much memory the conservative collector can not reclaim) versus the amount of total memory allocated and taking an average over 10,000 equally spaced points during the application's execution. These points 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 or how execution time is impacted. Numbers reported are in MB.

As can be seen in Table II, 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 more slowly than without garbage collection by a factor of 3.8. In this case, the 40 garbage collections take a total of 4472 milliseconds. Adding this to the execution time of the application without garbage collection (1654) accounts for almost all of the extra execution time. With an FSD value of 4, the default BDW configuration, `fred` executes 1.8 times more slowly than without garbage collection (again, the extra time spent collecting garbage nearly completely accounts for the difference).

When comparing different FSD values, the average (and to a lesser extent the maximum) garbage collection times increase as the frequency of garbage collection increases. This is because, in this application, the amount of reachable composite data grows during execution; in the BDW collector, tracing reachable composite objects accounts for the significant portion of garbage collection time. The cost of tracing composite objects dominates other costs because the BDW collector differentiates atomic objects from composite objects, only traces composite objects, and utilizes a lazy-sweep technique that efficiently sweeps objects during allocation (if and when that space is needed).

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 32 MB). However, as can be seen for applications with larger memory requirements such as `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 1.3 times slower than the fastest execution time that is obtained when an FSD value of 2 is used. In the case of `javac.100`, the slowest execution time is a factor of 7.5 times slower than the fastest execution time (that 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). On the other hand, the size of the reachable composite objects in the `fred` application grows during the execution. Therefore, increasing the frequency of garbage collections increases 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. When examining the heap size statistics gathered during execution with garbage collection turned off (not shown in the table), we found the heap grows beyond the size of physical memory to 121 MB. In the FSD = 2 case the 13 garbage collections performed limit 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 garbage collection when FSD = 1. As can be seen in the column labelled Max, the pause time for the one collection is 16012 ms.

The results in Table II 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 on garbage collection for some applications.

Table III 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 original algorithm 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 obtained using different FSD values for `fred` are unchanged, since it can easily execute within the available memory even without garbage collection. In 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 for the execution time of the application and the number of page faults, when garbage collection is turned off and when using FSD = 1. 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 with 128 MB of memory when compared with the 64 MB case. The execution time is significantly reduced in the 128 MB case for all FSD values except FSD = 16. However, the application now executes fastest when an FSD value of 2 is used (38 seconds), as compared with the 64 MB case when a best execution time of 62 seconds is obtained using FSD = 8.

Interestingly, when FSD = 1 is used in the 128 MB case, the overhead incurred by the one real garbage collection is significantly lower than when the application is executed on a system with 64 MB of memory. In both cases, the heap size is about 64 MB when collection is triggered (recall that about 45 to 50 MB is available for the application), so the heap has exceeded the amount of physical memory available in the 64 MB system. In the 128 MB case, the reachable composite objects can be traced without incurring many page faults (18 faults are incurred during collection), while a total of 2965 faults are incurred during collection in the 64 MB case.

In a system with 128 MB of memory, as the collection frequency increases (for FSD = 2, 4, 8, and 16) the differences in maximum and average garbage collection times are not nearly as dramatic as in the 64 MB case. In fact when garbage

Table III. Impact of garbage collection frequency using BDW with 128 MB system; times are in milliseconds and sizes are in MB. The row labelled "Tholds" shows the results obtained with our new algorithm. This algorithm and the results for it are discussed in Section 4.

Alg	Time	90%	GC	Total	Avg	Max	HG	Foot	pf	gcpf
					fred					
Off	1646	11	0	0	0	0	8	11.9	7546	0
FSD 1	1681	18	0	0	0	0	8	11.9	7546	0
FSD 2	2205	22	6	494	82	236	11	3.4	5755	9
FSD 4	3034	12	15	1328	88	302	17	1.9	5249	11
FSD 8	4228	8	25	2500	100	322	25	1.4	5154	11
FSD 16	6233	9	40	4463	111	320	38	1.1	5050	11
Tholds	1627	6	0	0	0	0	8	11.9	7546	0
					db.100					
Off	56198	197	0	0	0	0	13	54.9	31755	0
FSD 1	56138	198	0	0	0	0	13	54.9	31788	0
FSD 2	55349	5	13	2254	173	224	9	6.7	7681	18
FSD 4	56921	5	23	3916	170	219	13	4.2	6354	21
FSD 8	62597	7	51	9500	186	226	20	2.6	5461	21
FSD 16	69053	22	88	15959	181	219	30	2.1	5169	21
Tholds	53114	8	1	269	269	269	11	37.2	25103	18
					javac.100					
Off	405112	7328	0	0	0	0	23	134.1	190956	0
FSD 1	268804	4031	1	401	401	401	20	86.5	133641	18
FSD 2	38039	147	16	7004	418	1013	14	15.3	19248	41
FSD 4	45687	272	36	14596	401	1000	21	8.6	16346	41
FSD 8	58679	230	64	27337	424	976	34	5.8	13376	50
FSD 16	87488	1499	121	55667	459	944	52	4.2	11570	44
Tholds	35419	187	3	4563	1156	1380	12	34.3	25060	39

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 present in the next section is able to postpone garbage collection without incurring substantial costs (provided garbage collection isn't deferred too long).

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

4. A NEW APPROACH

After 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 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.[3]

[3]We currently do not consider the size of the caches and the impact on the TLB.

(2) As the amount of available memory becomes low some should be kept 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 the heap 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. This can be accomplished 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 perform a garbage collection or grow the heap. Our modified run-time system makes this 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 recent garbage collections (to ensure that collections are not too frequent and that they are actually reclaiming memory).

Garbage collection is triggered for the first time when the amount of memory used by the application exceeds the first threshold. When this or any threshold is exceeded for the first time, garbage is always collected. During a collection caused by a memory allocation that results in 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. Garbage will be collected if a sufficient amount of garbage was collected during recent collections. If the amount of memory reclaimed results in two or more thresholds being crossed, 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:

$$\begin{array}{lll} S_1 = T_2 - T_1 & \text{for} & T_1 \text{ and} \\ S_i = T_i - T_{i-1} & \text{for} & T_i \text{ where } i > 1. \end{array}$$

Note that this approach ties the amount of memory that should be reclaimed in order to be considered sufficient to thresholds. Although in this paper we have done this intentionally for simplicity, this is not required and the two could be separated.

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.

To try to ameliorate this situation we have added another decision point to our modified run-time 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. That is, even if the requested amount of memory could be allocated. 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 an amount of memory is allocated that grows the total to a point that crosses a threshold, 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 (to check if the current threshold is being crossed) it does not seem to impact the execution time of our applications in a noticeable way. As will be seen in Section 4.2 our new approach performs quite well when compared with the original approach.

Finally, when growing the heap, we grow it 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 the heap only up to the first threshold). Subsequent heap growths are done by growing the heap 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 (points A, B, and E), a square denotes that garbage was collected, (points C, F, and H) and a diamond denotes that the decision was to do neither (point J). The program starts with an initial heap (in our experiments, we used 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 $S_1 = T_2 - T_1$). 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.

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

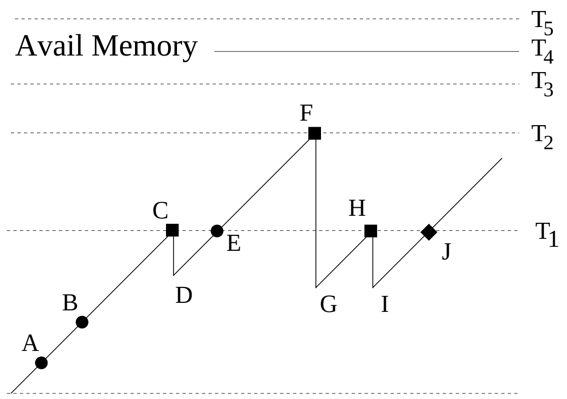


Fig. 2. Example heap growth and garbage collection operation using thresholds.

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 initiate garbage collection 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.

The main differences between our approach, the original approach used in the BDW collector, and other approaches (see Section 5 on related work for details) is that we dynamically control the garbage collection frequency and heap growth in an attempt to adapt to changes in available resources. We avoid garbage collections and aggressively growing the heap while memory is available, and then more aggressively initiate garbage collections and less aggressively grow the heap as memory becomes more utilized.

4.1 The Implementation

The new algorithm was implemented within the context of the existing JVM and garbage collector (as used in the previous experiments), taking care to only modify the portion of the code that makes decisions about when to perform garbage collection, when to grow the heap, and how much to grow the heap by.

We used the Win32 call `GlobalMemoryStatus()` to determine how much physical memory was in the system and how much memory is available. Other operating systems typically provide support for applications to obtain such information. For example, in Linux this information is available through the proc file system in `/proc/meminfo`; in a number of other versions of UNIX, it can be read from `/dev/kmem`.

For the purposes of our current experiments, we make an initial call to determine the amount of physical and available memory and compute static thresholds based on these values. As the program executes, we track the amount of memory that has not been reclaimed (i.e., the memory the run-time system considers live) and use this to determine where the program is executing relative to the thresholds. This approach circumvents issues related to accurately determining the amount of available memory during execution. For example, when all physical memory has been used most operating systems report that the amount of available memory is near or slightly above 0 bytes (typically, an attempt is made to keep a few MBs free). Unfortunately, this currently limits our approach to environments where the application is executing in isolation (e.g., many high-performance computing and server environments). We plan to explore techniques that will work with multiple applications in the future, for example, using information about available memory and page fault rates as has been done by Alonso and Appel [1990] (see Section 5 on related work).

Naturally, the choice of the number of thresholds and their locations can greatly influence application performance. Although we would prefer a technique that does not require parameter tuning, we did not 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.* For the experiments conducted with 64 MB of memory, we used these logical thresholds, defined as a fraction of the system memory:

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 these logical thresholds, defined as a fraction of the system memory:

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

The 64 MB thresholds start at 0.40 and were chosen at 15% intervals (a little under 8 MB) with an additional threshold added roughly half way between the 0.85 and 1.00 thresholds (at 0.92). Without this additional threshold, the last collection before reaching the physical memory threshold would occur with roughly 8 MB of memory available, which we found was a little too late for some applications.

The 128 MB thresholds start at 0.80 and were chosen at 5% intervals (around 5 MB). In this case, we did not see a need for an extra interval just before reaching the physical memory threshold because the threshold comes about 5 MB before the physical threshold. In both cases, we started with thresholds at 1.00 and picked the remaining thresholds based on intervals around the physical memory threshold. Originally, the thresholds for the 64 MB case did not extend as low as those currently used, but we found that the higher starting threshold was not quite aggressive enough for some applications.

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 set the next threshold (the last threshold) to a point well beyond available memory (this point was not reached in any of our experiments). In both the 64 MB and 128 MB case the last threshold has been chosen simply to ensure that it is large enough so that it no garbage collections would be triggered after passing the second to last threshold. An interesting question we hope to study in the future is how to schedule garbage collection in programs where the footprint far exceeds the physical memory in the system.

4.2 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. Tables II and III compare details of three applications executing using our threshold-based approach (the row labelled "Tholds") with the execution of the same applications using different FSD values. In the 64 MB case, which is shown in Table II, the run time obtained using thresholds compares quite favorably when compared with any single FSD value. Since the FSD is static there is no one FSD value that could be chosen that does as well over all of the applications shown as the threshold-based approach. Interestingly, the run time obtained using our approach is in fact quite close to the best run time obtained over all FSD values (except for the case of `fred`, which is discussed in more detail later).

In the 128 MB case, (Table III), the results are more dramatic. The run time obtained using the threshold-based approach is significantly better than any single FSD value and surprisingly performs as well or better than the best run time obtained over all different FSD values.

Note that when executing `javac.100` using our threshold-based approach on a 64 MB system, an average of 24 collections were performed and a mean execution time of 64 seconds was observed. In contrast, with FSD = 2, 4, and 8 the collector was invoked an average of 16, 35, and 64 times respectively, while the mean execution times were 323, 75, 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 (it lies between FSD = 4 and FSD = 8). As a result, we conclude that in some cases the benefits obtained from using our threshold-based approach are a result of controlling the point at which collections occur more so than controlling the frequency.

We now expand our comparisons with a wider variety of programs. 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 garbage collection turned off. These environments are examined explicitly because the majority of the applications used in our experiments execute very efficiently using one of these three scenarios. The results of these experiments are shown in Figure 3(a), 3(b), and 3(c), respectively. Figure 3(d) compares the execution times obtained using our threshold-based approach with the minimum execution times obtained across all FSD values. Additionally, we include detailed statistics regarding the execution of all applications in the appendix.

In all graphs in Figure 3, the execution time obtained with our scheduling algorithm averaged over 19 runs is normalized with respect to the mean execution time over 19 runs of the application when executed using the standard BDW environment (for the FSD value used). The execution times of our algorithm are shown using the taller, dark-colored bars. The light portion of each bar indicates the portion of the normalized execution time that the threshold-based approach spent on garbage collection. As can be seen in Figure 3, in most cases garbage collection times are negligible.

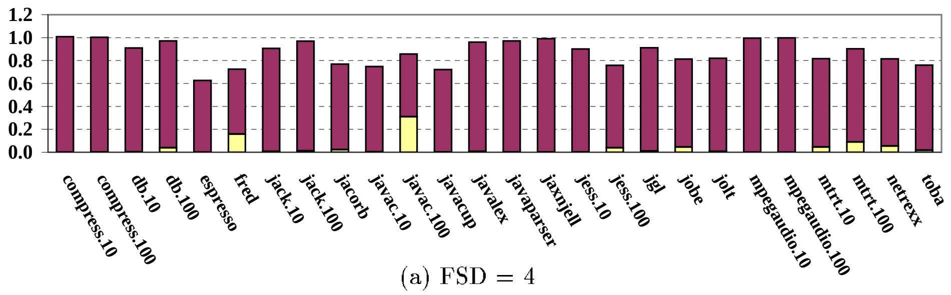
Figure 3(a) shows that the Java applications used execute quite well using the threshold-based approach when compared with the standard BDW approach using FSD = 4 (the default value). Using the threshold-based algorithm, many of the applications (11 of 26) execute in roughly 80% or less of the time required 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 more slowly using the threshold-based approach.

Note that the results for our entire application set are shown even though when executing with 64 MB, 6 of these 26 applications (compress.10, javalex, javaparser, jaxnjell, mpegaudio.10, and mpegaudio.100) execute in the same amount of time whether garbage collection is turned off, FSD = 4, FSD = 2, or our threshold-based algorithm is used. In these cases, garbage collection is typically invoked only a few times and collection overheads are very small relative to the total execution time. That is, they might not be considered to be very good garbage collection benchmarks for a system with 64 MB but we felt that it would be interesting to include them since some of them are from the SPECJVM98 benchmark suite.

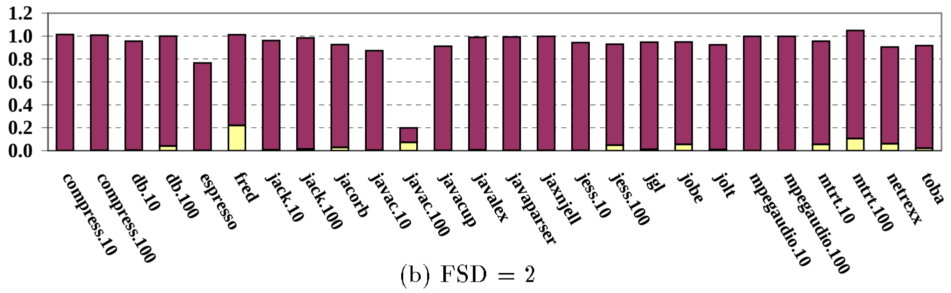
We also see from Figure 3(c) that 13 of the 26 applications execute fastest with no garbage collections. We considered using a smaller amount of memory than the 64 MB used for these experiments but when no garbage collections are performed 8 of the applications execute with a total heap size of 8 MB or less, 3 execute with 15-18 MB of heap, and only one requires 33 MB.

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 36% of the run time) `fred` (where it is about 22%), and `mtrt.100` (where it is about 10%). 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.

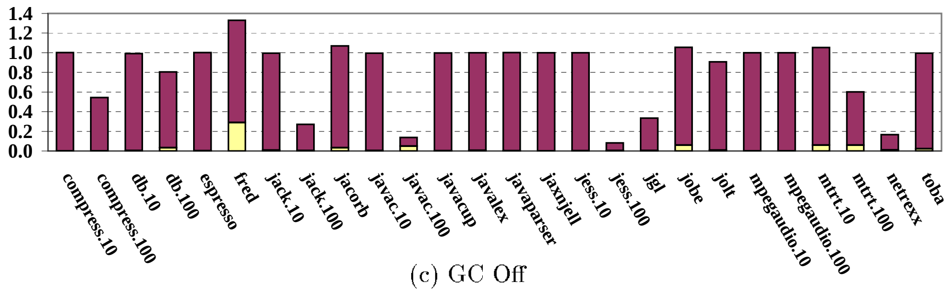
The results observed in Figure 3(b), which compare the threshold-based algorithm with FSD = 2, are similar to those seen in Figure 3(a) (for FSD = 4), but they are not as dramatic. For all executions except `espresso` and `javac.100`, the threshold-based algorithm and FSD = 2 yield execution times that are within approximately 5 to 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 it with FSD = 2 (a factor of 5 improvement). 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



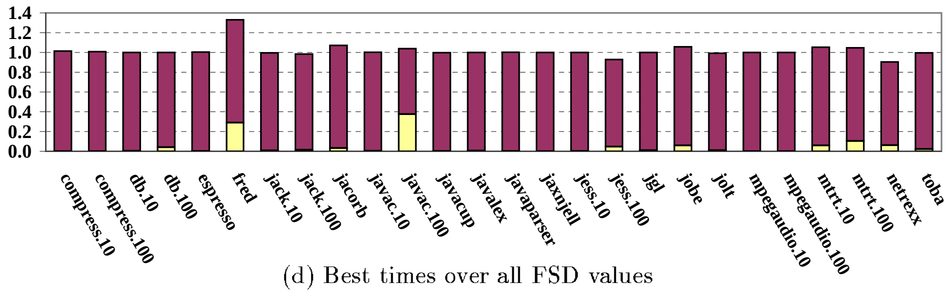
(a) FSD = 4



(b) FSD = 2



(c) GC Off



(d) Best times over all FSD values

Fig. 3. Comparing the threshold-based algorithm with the BDW algorithm using different FSD values on a 64 MB system. These graphs show the execution time of each application when run using the threshold-based algorithm, normalized with respect to the execution time obtained when run using the specified FSD value.

value of 4 appears to be more conservative and prevents any one of the applications tested from suffering from very poor performance.

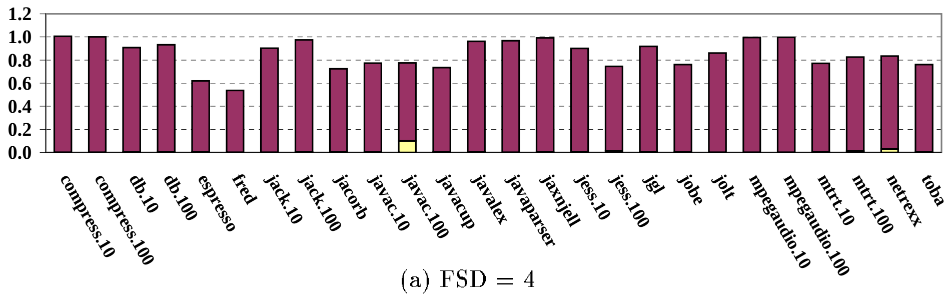
Figure 3(c) compares the execution times of the threshold-based algorithms with those obtained when garbage collection is turned off. By comparing these results with those in Figure 3(d), we can see that in our environment a number of applications execute most effectively when garbage collections do 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 more slowly by a factor of 1.3 when using the threshold-based algorithm than when garbage collection is turned off (which is when `fred` executes fastest). This is because `fred` is a very short-running program, and the one garbage collection initiated in the threshold-based algorithm takes a total of 477 ms and inflates 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.

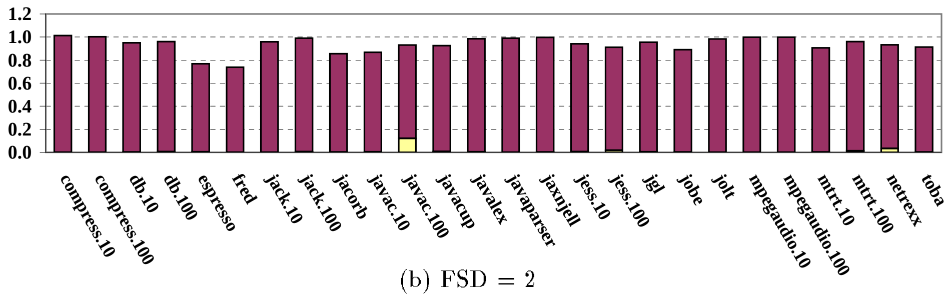
As mentioned previously, when designing the threshold-based approach it was our hope that it would perform as well as the best possible FSD value across all applications, that is, to perform enough garbage collections to avoid paging for those applications that use a large amount of memory and to avoid collecting garbage too frequently for those applications whose execution would be negatively impacted. Figure 3(d) compares the mean execution time of each application with the mean execution 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).[4]

This graph shows that with the exception of the `fred` application our approach to controlling garbage collection and heap growth is reasonably close to the best FSD value. Most of the applications execute in the same time under both approaches while a few others execute about 5% faster or slower using the threshold-based approach. It is worth re-emphasizing that because the FSD value is static, obtaining such performance using the original BDW implementation is not possible. The main comparison of interest is the threshold-based approach versus the default BDW configuration with FSD = 4 (arguably a good choice). In this case significant performance improvements are obtained for several applications using the threshold-based approach.

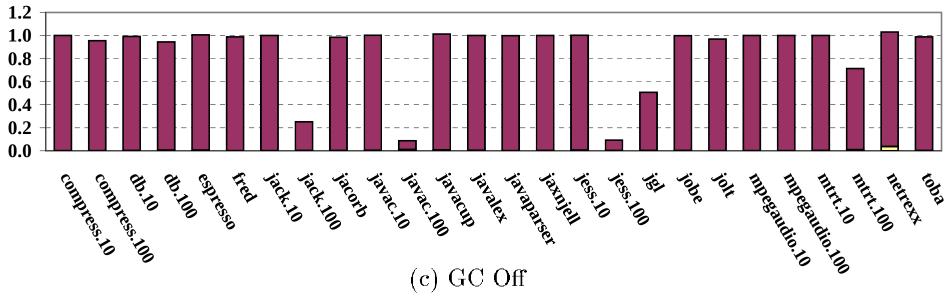
[4]For the applications tested, none of the execution times decreased with FSD values of 32 or higher.



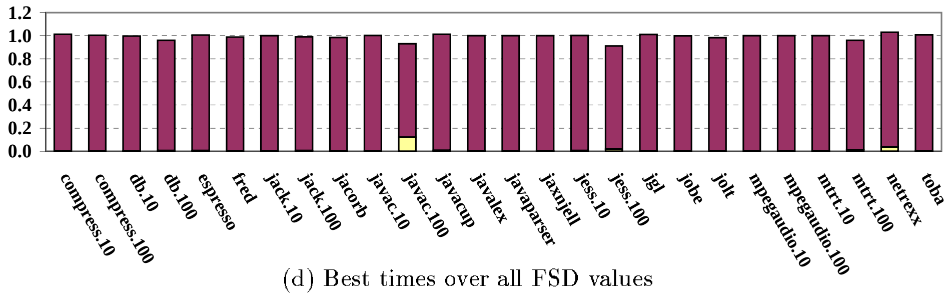
(a) FSD = 4



(b) FSD = 2



(c) GC Off



(d) Best times over all FSD values

Fig. 4. Comparing the threshold-based algorithm with the BDW algorithm using different FSD values on a 128 MB system. These graphs show the execution time of each application when run using the threshold-based algorithm, normalized with respect to the execution time obtained when run using the specified FSD value.

To demonstrate that our approach also works with different amounts of physical memory, we conducted the same set of experiments with 128 MB of memory and in Figure 4 present a series of graphs similar to those in Figure 3. The observations in Figure 4 are similar to those made when executing the applications using 64 MB of memory except that the improvements from the threshold-based approach appear to be slightly more consistent with 128 MB of memory than with 64 MB. Again we include detailed statistics obtained for all applications in the appendix. In the 128 MB case, all applications execute in times 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 about 13%.

5. RELATED WORK

Moon [1984] 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.

Ungar and Jackson [1988; 1992] conduct a simulation study to examine the impact that tenuring decisions have on the pause times in a generation-based scavenging garbage collector. They first show that when using a fixed-age tenuring policy low tenure thresholds (based on the amount of time an object has survived) produce the most tenured garbage and the shortest pause times. They then introduce feedback-mediated tenuring in which they base future tenuring decisions upon the amount of surviving data in the youngest generation. Their work is able to reduce pause times in their simulated environment which can provide significant benefits in an interactive environment. However, they do not consider the cost of later collecting the increased amount of memory that has been tenured, nor the impact of page faults. Our work is concerned with the execution times of applications and as a result we are willing to tolerate longer pause times if they lead to a decrease in the total execution time. Although our technique does not attempt to control pause times directly, our experimental results show that average and maximum pause times in almost all cases are as low or lower than the alternative approaches considered.

Cooper et al. [1992] show how the performance of Standard ML can be improved by applying optimizations to a simple generational collector introduced by Appel [1989]. In addition to utilizing Mach's support for sparse address spaces and external pagers, they propose and study a modification to Appel's algorithm for deciding how to grow the heap. For each of the three applications studied they use a brute force approach to determine optimal values (i.e., those that produce the fastest execution time) for the two parameters used by Appel's algorithm and the three parameters used by their algorithm. By modifying the algorithm for growing the heap they are able to significantly reduce the number of page faults and the execution time of two of the three applications studied (the performance of the third was not changed significantly). Although this aspect was not the focus of their study, it is interesting to see that the set of optimal parameters is different for each application and that it varies with the other approaches used to reduce paging.

Smith and Morrisett [1998] describe a mostly copying collector that collects garbage whenever the heap is two-thirds full. This roughly corresponds to using an FSD value of 1.5 in the BDW collector and will suffer from the same drawbacks that a fixed FSD value were shown to have in Section 3.1. Namely, no one fixed value works best for different applications and no one fixed value works best for different physical memory sizes.

Zorn [1990; 1993] 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 trade-off 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 has 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.[5] Namely, no one allocation threshold was suitable for all applications. Because the amount of memory available is not considered when initiating garbage collection, some applications exceeded the amount of memory available and overheads due to paging significantly increased 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 [1990] 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 the application 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. Garbage collections occur only when the free space portion of the heap is exhausted [Appel 2003]. As a result, control over garbage collection occurs only by modifying the size of the heap. Unfortunately, this approach cannot be deployed in the BDW collector because 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 collect garbage 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, the work of Alonso and Appel does demonstrate that garbage collection can be controlled in a number of applications executing simultaneously to provide reduced execution times.

[5]To be fair, Zorn devised this algorithm to compare fairly 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.

Kim et al. [2000] analyze 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 point out that although the direct costs of garbage collection decrease as the available heap size is increased, there exists an optimal heap size that minimizes total execution time (due to interaction with the virtual memory subsystem).

One of the main differences between our work in this paper and the work described above is that we specifically study and devise techniques for controlling garbage collection and heap growth that consider how much memory is available in the system and we directly consider the trade-off between execution time and application footprint.

Dimpsey et al. [2000] describe the IBM DK version 1.1.7 for Windows. This is derived from a Sun reference JVM implementation and changes were made in order to improve performance of applications executing in server environments. Their approach also considers the amount of physical memory in the system. They set the default initial and maximum heap size to values that are proportional to the amount of physical memory in the system. However, they do not explain what values are used or how they were chosen.

They also make modifications to reduce the number of heap growths, because they are quite costly in their environment. If the amount of space available after a garbage collection is less than 25% of physical memory or if the ratio of time spent collecting garbage to time spent executing the application exceeds 13%, the heap is grown by 17% of the heap size. They report that the ratio based heap growth was disabled if the heap approached 75% of the size of physical memory but they do not explain what was done in this case. They report that when starting with an initial heap size of 2 MB this approach increases throughput by 28% on the VolanoMark and pBOB benchmarks.

Recent versions of the Sun Java virtual machine include a large collection of command line options designed to permit users to adjust parameters that impact heap growth and garbage collection. As well, documents are provided [Sun Microsystems 2005] that explain that these parameters need to be adjusted for each application and describe how to read garbage collection trace data in order to tune these parameters.

Our work in this paper concentrates more on trying to understand what should influence the decision about when to garbage collect and by how much to grow the heap. We provide full details of how decisions are made and how they are made relative to the amount of available memory. In addition, we study a variety of benchmark applications and demonstrate that we have been able to obtain good performance across a variety of applications.

6. DISCUSSION

Decreased garbage collection times, whether they are achieved by improving the implementation used (e.g., by reducing cache misses during collections [Boehm 2000]) or by utilizing a different collector, may mean garbage collection can be invoked more frequently without negatively impacting an 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.

Garbage collection overheads that have been reported in the literature are still sufficiently large that our approach might prove useful if used with other garbage collectors. Experiments conducted by Fitzgerald and Tarditi [2000] 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 to 25% of the execution time for several combinations of applications and collectors they studied. They point out that for some of their applications reducing the number of garbage collections by half roughly halves the time spent in garbage collection. While this does provide evidence that our techniques might work in conjunction with other garbage collectors these results and the results from our paper were both obtained using Java to native instruction compilers. It is possible that slower execution of the Java application code inside the context of a virtual machine might render garbage collection times inconsequential. However, just-in-time hot-spot technologies should compete well with compiled code. In the future we hope to examine our technique in the context of an optimized virtual machine that supports different garbage collectors.

Of course, our technique may not work with all garbage collectors. A benefit of the mark and lazy-sweep approach that differentiates atomic and composite objects used in the BDW collector is that delaying garbage collection does not appear to significantly increase the time spent in garbage collection (because pause times are correlated with the size of the composite object set size). While this is true for 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 et al. [2000] incurs overheads proportional to the amount of garbage being collected. It is unclear what impact delaying garbage collection would have in such environments.

Several previous studies have pointed out the importance of, studied, and improved the cache and TLB hit rates within the context of garbage collected systems [Zorn 1991; Wilson et al. 1991; 1992; Grunwald et al. 1993; Chilimbi and Larus 1998; Smith and Morrisett 1998; Boehm 2000; Shuf et al. 2001]. Our results described in this paper do not consider or contain information regarding the impact on cache and TLB hit rates. In future work it would be interesting to determine both the impact our approach has on caches and the TLB and to examine if adding new thresholds corresponding to the size of the level two or greater caches could be used to improve execution times (especially for systems with large caches).

As discussed in Section 5, Alonso and Appel [1990] 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 multi-programmed environments in mind. In such environments the amount of available memory is reduced and presumably thresholds are reached sooner. Such an environment might also require us to adjust our thresholds dynamically 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 the differences in thresholds used for 64 MB and 128 MB systems (see Section 4).

Lastly, the choice of thresholds is very important in our experiments. The values used in this paper were chosen through trial and error and yield good results for all applications when compared with results obtained using different FSD values. It is worth noting that we do not expect that the set of thresholds used here will always work so well for all applications. This combined with the fact that we tuned the thresholds differently for each memory configuration and that different memory configurations would likely also require tuning, point out the need for techniques that either do not require tuning or that tune themselves. Some work has shown some potential in this area [Andreasson et al. 2002] by deploying machine learning techniques in order to determine when to garbage collect.

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.

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 mark-and-sweep Boehm-Demers-Weiser (BDW) garbage collector and memory allocator. Our goal is to obtain a better understanding of how to control garbage collection and heap growth for an individual application executing in isolation. These results can be applied in a number of high-performance computing and server environments, in addition to some single user environments. This work should also provide insights into how to make better decisions that impact garbage collection in multi-programmed environments.

From the experiments conducted in our environment we observed the following:

—The execution times of many of the applications we tested vary 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 static 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 an application relative to the best FSD value.

—The best scheduling algorithm for an application (the one that 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 perform garbage collection 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 use these observations to design, implement, and experimentally evaluate a threshold-based algorithm for controlling garbage collection and heap growth. Our experiments demonstrate that when compared with the execution time obtained with the method used by the standard BDW implementation (i.e., when FSD = 4), our new approach can match or significantly reduce the execution time of every application. Additionally, our approach is preferable to any other single FSD value and interestingly, the run times compare favorably (in all but one case) with the best run time obtained across all FSD values. We believe that this benefit is obtained by taking into account the amount of memory available to the application when determining whether to collect garbage or to grow the heap, carefully controlling heap growth when memory resources become scarce, and by considering the amount of memory reclaimed in previous collections.

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.

ACKNOWLEDGMENTS

We thank Richard Paige, Natalie Polikashov and Szymon Stachniak at York University for their fruitful discussions and Ben Korvemaker and the anonymous referees for their comments which have helped to improve this paper.

We thank the Natural Sciences and Engineering Research Council (NSERC) of Canada for grants and scholarships that partially supported this research. Additionally, we are grateful for support received from the IBM Center for Advanced Studies (CAS) and for the efforts of and interactions with members of CAS and the developers of HPJ. In particular we thank, Patrick Gallop, Josée Lajoie, Bill O'Farrell, Albert Rothenstein, Ven Seshadri, Clark Verbrugge and Dean Williams.

APPENDIX

This section contains a number of the statistics that were collected while executing all 26 versions of our applications. These tables compare the impact of garbage collection frequency using the BDW algorithms and our new threshold-based algorithm (the rows labelled Tholds). The first set of results were obtained using a 64 MB system and are shown in Tables IV, V, and VI. The second set of results were recorded using a 128 MB system and are shown in Tables VII, VIII, and IX. All times are reported in milliseconds and sizes are reported in kilobytes.

The results shown are the average observed over 19 runs. The columns shown in each table are the algorithm used (Alg), application execution time (Time) and 90% confidence interval for the execution time (90%), the number of garbage collections (GCs), the total time spent garbage collecting per run (Total), the average time spent garbage collecting per collection (Avg), the maximum time spent garbage collecting per run (Max), the average memory footprint of the application (Foot), the total number of page faults that occurred during execution (pfs) and the total number of page faults that occurred during garbage collections (GC pfs). Although in all cases at least one garbage collection is reported, the first call is for initialization purposes only and no garbage is collected.

Table IV. 64 MB

Alg	Time	90%	GC	Total	Avg	Max	HG	Foot	pf	gcpf
compress.10										
Off	6992	5	0	0	0	0	6	4.4	4028	0
FSD 2	6901	6	2	16	8	11	7	3.9	3972	18
FSD 4	6940	4	6	44	7	8	8	3.0	3856	21
Tholds	6999	6	0	0	0	0	6	4.4	4028	0
compress.100										
Off	139741	1326	0	0	0	0	11	52.7	41554	0
FSD 2	75313	30	7	59	8	10	6	13.2	9847	18
FSD 4	75759	15	12	96	8	9	7	11.0	8290	21
Tholds	75956	57	7	64	8	13	6	12.3	8962	18
db.10										
Off	1992	9	0	0	0	0	6	2.7	4288	0
FSD 2	2066	7	3	65	21	43	7	1.6	3832	18
FSD 4	2173	6	6	160	26	70	11	1.2	3818	21
Tholds	1975	6	0	0	0	0	6	2.7	4288	0
db.100										
Off	68823	1146	0	0	0	0	13	54.9	35709	0
FSD 2	55480	9	13	2260	173	224	9	6.7	7682	18
FSD 4	57044	6	23	3922	170	220	13	4.2	6355	21
Tholds	55443	5	10	2250	225	231	7	6.7	7265	18
espresso										
Off	786	5	0	0	0	0	7	4.7	3661	0
FSD 2	1029	5	5	237	47	119	8	1.9	2969	10
FSD 4	1258	5	10	470	47	119	12	1.2	2721	10
Tholds	788	6	0	0	0	0	7	4.7	3661	0
fred										
Off	1654	4	0	0	0	0	8	11.9	7547	0
FSD 2	2174	4	6	495	82	237	11	3.4	5756	9
FSD 4	3035	6	15	1331	88	302	17	1.9	5248	11
Tholds	2200	67	1	477	292	314	8	8.0	6150	9
jack.10										
Off	4158	6	0	0	0	0	8	14.7	9672	0
FSD 2	4310	4	13	247	19	32	8	1.8	3428	18
FSD 4	4564	4	30	515	16	28	10	0.9	2874	21
Tholds	4137	5	1	31	31	31	8	8.3	7130	18
jack.100										
Off	122918	5240	0	0	0	0	22	122.4	89167	0
FSD 2	33781	25	36	916	24	38	9	4.7	5029	18
FSD 4	34278	23	72	1530	21	35	13	2.8	3974	21
Tholds	33254	22	13	451	34	44	8	9.4	7097	18
jacorb										
Off	1874	16	0	0	0	0	8	10.4	6834	0
FSD 2	2166	14	9	307	34	49	8	1.5	2745	9
FSD 4	2607	20	21	698	33	49	11	0.8	2496	9
Tholds	2006	21	1	58	58	57	8	8.4	6201	9

Table V. 64 MB

Alg	Time	90%	GC	Total	Avg	Max	HG	Foot	pf	gcpf
				javac.10						
Off	1274	6	0	0	0	0	7	5.8	5004	0
FSD 2	1455	5	6	188	31	59	8	1.5	3433	18
FSD 4	1698	19	14	430	29	55	10	0.8	2863	21
Tholds	1269	6	0	0	0	0	7	5.8	5004	0
				javac.100						
Off	463472	40771	0	0	0	0	23	134.1	183910	0
FSD 2	323418	14149	16	92065	5642	53188	14	16.7	77897	16446
FSD 4	74770	7268	35	24162	677	10012	21	8.6	21387	2513
Tholds	64130	4696	24	23145	958	3501	12	8.6	17535	1871
				javacup						
Off	536	4	0	0	0	0	6	3.3	3039	0
FSD 2	587	3	5	63	12	27	8	0.9	2312	9
FSD 4	742	4	14	214	15	35	9	0.5	2048	12
Tholds	535	3	0	0	0	0	6	3.3	3039	0
				javalex						
Off	713	3	0	0	0	0	5	1.2	1795	0
FSD 2	721	3	3	16	5	11	4	0.4	1446	10
FSD 4	742	4	8	39	4	7	3	0.3	1338	10
Tholds	713	2	0	0	0	0	5	1.2	1795	0
				javaparser						
Off	1717	2	0	0	0	0	6	2.7	2490	0
FSD 2	1734	2	7	37	5	12	4	0.6	1333	9
FSD 4	1773	3	15	72	4	7	3	0.4	1231	9
Tholds	1721	3	0	0	0	0	6	2.7	2490	0
				jaxnjell						
Off	4343	4	0	0	0	0	4	0.9	1517	0
FSD 2	4354	3	2	20	10	13	5	0.4	1433	9
FSD 4	4380	2	5	45	9	15	6	0.3	1390	9
Tholds	4341	4	0	0	0	0	4	0.9	1517	0
				jess.10						
Off	1381	5	0	0	0	0	6	2.9	3390	0
FSD 2	1466	3	5	94	18	29	6	0.8	2533	17
FSD 4	1534	5	10	166	16	28	7	0.5	2371	21
Tholds	1382	3	0	0	0	0	6	2.9	3390	0
				jess.100						
Off	310286	19105	0	0	0	0	29	181.3	294003	0
FSD 2	27205	9	77	3443	44	53	9	2.9	4244	17
FSD 4	33321	54	233	9610	41	49	11	1.1	3208	21
Tholds	25274	8	21	1280	60	66	8	8.7	7184	17
				jgl						
Off	42722	1564	0	0	0	0	17	82.5	48635	0
FSD 2	15118	4	148	1056	7	14	5	0.7	1490	9
FSD 4	15716	4	258	1723	6	14	6	0.5	1459	9
Tholds	14323	5	8	145	18	23	8	9.6	6318	9

Table VI. 64 MB

Alg	Time	90%	GC	Total	Avg	Max	HG	Foot	pf	gcpf
					jobe					
Off	3207	6	0	0	0	0	8	10.4	6580	0
FSD 2	3572	10	6	369	61	178	11	3.3	5009	11
FSD 4	4171	5	15	971	64	173	15	1.7	4272	11
Tholds	3387	7	1	189	189	189	8	9.2	6229	11
					jolt					
Off	5919	214	0	0	0	0	8	14.2	8690	0
FSD 2	5812	44	14	339	24	38	8	1.3	2603	10
FSD 4	6555	150	42	977	23	37	9	0.6	1890	12
Tholds	5373	49	1	45	45	44	8	7.9	6104	10
					mpegaudio.10					
Off	14502	6	0	0	0	0	3	0.4	2096	0
FSD 2	14530	6	1	8	8	8	4	0.3	2098	18
FSD 4	14557	5	3	23	7	10	4	0.2	2094	21
Tholds	14502	4	0	0	0	0	3	0.4	2096	0
					mpegaudio.100					
Off	131137	22	0	0	0	0	3	0.3	2049	0
FSD 2	131328	25	1	8	8	9	3	0.2	2056	18
FSD 4	131427	45	2	14	7	8	3	0.2	2056	21
Tholds	131128	13	0	0	0	0	3	0.3	2049	0
					mtrt.10					
Off	3529	3	0	0	0	0	8	13.2	8840	0
FSD 2	3889	5	7	381	54	192	11	3.9	6826	18
FSD 4	4549	6	17	1033	57	202	16	2.3	5711	21
Tholds	3714	4	1	209	209	208	8	8.6	7083	18
					mtrt.100					
Off	65330	5740	0	0	0	0	19	103.9	64117	0
FSD 2	37562	27	16	2103	130	213	12	10.2	9668	18
FSD 4	43623	71	55	8016	145	204	16	3.8	5817	21
Tholds	39358	133	20	3944	193	210	8	6.5	7160	18
					netrexx					
Off	33209	858	0	0	0	0	12	45.3	32858	0
FSD 2	6098	23	19	961	50	72	10	4.3	6015	12
FSD 4	6765	17	35	1665	47	67	13	2.4	4546	15
Tholds	5510	18	5	369	73	84	9	9.0	7759	12
					toba					
Off	1943	7	0	0	0	0	8	13.2	8112	0
FSD 2	2113	7	15	256	17	22	7	1.1	2092	10
FSD 4	2546	29	39	634	16	23	7	0.6	1591	10
Tholds	1936	8	1	36	36	35	8	8.1	6251	9

Table VII. 128 MB

Alg	Time	90%	GC	Total	Avg	Max	HG	Foot	pf	gcpf
compress.10										
Off	6989	3	0	0	0	0	6	4.4	4027	0
FSD 2	6903	4	2	15	7	9	7	3.9	3971	18
FSD 4	6947	4	6	44	7	9	8	3.0	3855	21
Tholds	6990	3	0	0	0	0	6	4.4	4027	0
compress.100										
Off	79048	97	0	0	0	0	11	52.7	29205	0
FSD 2	75365	11	7	59	8	12	6	13.1	9781	18
FSD 4	75501	19	12	96	7	10	7	10.9	8290	21
Tholds	75548	10	1	10	10	10	9	37.1	23722	22
db.10										
Off	1976	8	0	0	0	0	6	2.7	4286	0
FSD 2	2065	5	3	65	21	43	7	1.6	3831	18
FSD 4	2161	6	6	159	26	70	11	1.2	3817	21
Tholds	1961	6	0	0	0	0	6	2.7	4287	0
db.100										
Off	56198	197	0	0	0	0	13	54.9	31755	0
FSD 2	55349	5	13	2254	173	224	9	6.7	7681	18
FSD 4	56921	5	23	3916	170	219	13	4.2	6354	21
Tholds	53114	8	1	269	269	269	11	37.2	25103	18
espresso										
Off	767	3	0	0	0	0	7	4.7	3660	0
FSD 2	1005	3	5	241	48	120	8	1.9	2968	10
FSD 4	1245	3	10	472	47	119	12	1.2	2721	10
Tholds	771	4	0	0	0	0	7	4.7	3660	0
fred										
Off	1646	11	0	0	0	0	8	11.9	7546	0
FSD 2	2205	22	6	494	82	236	11	3.4	5755	9
FSD 4	3034	12	15	1328	88	302	17	1.9	5249	11
Tholds	1627	6	0	0	0	0	8	11.9	7546	0
jack.10										
Off	4104	3	0	0	0	0	8	14.7	9671	0
FSD 2	4286	3	13	246	18	32	8	1.8	3428	18
FSD 4	4551	5	30	510	16	28	10	0.9	2873	21
Tholds	4105	3	0	0	0	0	8	14.7	9671	0
jack.100										
Off	132019	1579	0	0	0	0	22	122.4	101951	0
FSD 2	33615	25	34	873	24	38	9	5.0	5277	18
FSD 4	34163	26	72	1517	20	34	13	2.8	3946	21
Tholds	33292	21	2	147	73	73	12	41.3	25134	25
jacorb										
Off	1923	26	0	0	0	0	8	10.4	6833	0
FSD 2	2212	42	9	308	34	49	8	1.5	2744	9
FSD 4	2617	27	21	697	33	50	11	0.8	2496	9
Tholds	1894	20	0	0	0	0	8	10.4	6833	0

Table VIII. 128 MB

Alg	Time	90%	GC	Total	Avg	Max	HG	Foot	pf	gcpf
javac.10										
Off	1247	4	0	0	0	0	7	5.8	5003	0
FSD 2	1441	5	6	188	31	59	8	1.5	3432	18
FSD 4	1617	10	13	363	27	55	10	0.9	2906	21
Tholds	1250	3	0	0	0	0	7	5.8	5003	0
javac.100										
Off	405112	7328	0	0	0	0	23	134.1	190956	0
FSD 2	38039	147	16	7004	418	1013	14	15.3	19248	41
FSD 4	45687	272	36	14596	401	1000	21	8.6	16346	41
Tholds	35419	187	3	4563	1156	1380	12	34.3	25060	39
javacup										
Off	533	3	0	0	0	0	6	3.3	3038	0
FSD 2	584	4	5	63	12	27	8	0.9	2312	9
FSD 4	735	4	14	213	15	35	9	0.5	2047	12
Tholds	540	6	0	0	0	0	6	3.3	3038	0
javalex										
Off	708	2	0	0	0	0	5	1.2	1794	0
FSD 2	720	2	3	15	5	9	4	0.4	1445	10
FSD 4	737	2	8	38	4	11	3	0.3	1337	10
Tholds	709	3	0	0	0	0	5	1.2	1794	0
javaparser										
Off	1712	2	0	0	0	0	6	2.7	2489	0
FSD 2	1727	2	7	37	5	7	4	0.6	1332	9
FSD 4	1768	2	15	71	4	9	3	0.4	1230	9
Tholds	1711	2	0	0	0	0	6	2.7	2489	0
jaxnjell										
Off	4340	3	0	0	0	0	4	0.9	1516	0
FSD 2	4352	2	2	19	9	13	5	0.4	1432	9
FSD 4	4379	3	5	44	8	13	6	0.3	1389	9
Tholds	4342	3	0	0	0	0	4	0.9	1516	0
jess.10										
Off	1370	2	0	0	0	0	6	2.9	3389	0
FSD 2	1457	2	5	92	18	29	6	0.8	2532	17
FSD 4	1523	4	10	165	16	28	7	0.5	2370	21
Tholds	1372	2	0	0	0	0	6	2.9	3389	0
jess.100										
Off	267384	13254	0	0	0	0	29	181.3	304567	0
FSD 2	27035	3	77	3425	44	53	9	2.9	4243	17
FSD 4	33126	57	233	9555	40	48	11	1.1	3207	21
Tholds	24658	8	4	449	112	119	12	40.5	25120	17
jgl										
Off	28340	721	0	0	0	0	17	82.5	48304	0
FSD 2	15043	4	148	1036	7	14	5	0.7	1489	9
FSD 4	15645	4	258	1688	6	15	6	0.5	1458	9
Tholds	14364	5	1	58	58	58	12	41.5	24241	9

Table IX. 128 MB

Alg	Time	90%	GC	Total	Avg	Max	HG	Foot	pf	gcpf
					jobe					
Off	3160	13	0	0	0	0	8	10.4	6579	0
FSD 2	3549	10	6	369	61	178	11	3.3	5021	11
FSD 4	4153	6	15	978	65	175	15	1.7	4273	12
Tholds	3158	4	0	0	0	0	8	10.4	6579	0
					jolt					
Off	5661	67	0	0	0	0	8	14.2	8695	0
FSD 2	5580	15	14	339	24	38	8	1.3	2602	10
FSD 4	6377	99	42	977	23	37	9	0.6	1889	12
Tholds	5484	45	0	0	0	0	8	14.2	8701	0
					mpegaudio.10					
Off	14494	4	0	0	0	0	3	0.4	2095	0
FSD 2	14524	5	1	8	8	8	4	0.3	2097	18
FSD 4	14557	5	3	23	7	11	4	0.2	2093	21
Tholds	14497	3	0	0	0	0	3	0.4	2095	0
					mpegaudio.100					
Off	131142	19	0	0	0	0	3	0.3	2048	0
FSD 2	131306	24	1	8	8	9	3	0.2	2055	18
FSD 4	131439	35	2	14	7	10	3	0.2	2055	21
Tholds	131147	21	0	0	0	0	3	0.3	2049	0
					mtrt.10					
Off	3497	3	0	0	0	0	8	13.2	8839	0
FSD 2	3861	3	7	377	53	189	11	3.9	6822	19
FSD 4	4536	11	18	1040	57	202	16	2.3	5710	22
Tholds	3500	3	0	0	0	0	8	13.2	8839	0
					mtrt.100					
Off	50485	1542	0	0	0	0	19	103.9	61691	0
FSD 2	37472	24	16	2138	131	213	12	10.2	9667	19
FSD 4	43657	134	55	8146	146	203	16	3.8	5816	22
Tholds	36019	15	2	454	227	254	12	37.8	25108	18
					netrexx					
Off	5430	14	0	0	0	0	12	45.3	26132	0
FSD 2	5996	25	18	939	49	80	10	4.4	6012	12
FSD 4	6704	19	35	1658	47	66	14	2.4	4577	15
Tholds	5592	15	1	193	193	193	12	40.9	24902	11
					toba					
Off	1983	26	0	0	0	0	8	13.2	8111	0
FSD 2	2144	19	15	254	16	23	7	1.1	2091	10
FSD 4	2573	27	39	630	16	22	7	0.6	1590	10
Tholds	1959	25	0	0	0	0	8	13.2	8111	0

REFERENCES

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

Andreasson, E., Hoffmann, F., and Lindholm, O. 2002. Memory management through machine learning: To collect or not to collect? In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '02)*. San Francisco, CA.

Appel, A. 2003. Personal communication.

Appel, A. W. 1989. Simple generational garbage collection and fast allocation. *Software Practice and Experience 19,* 2, 171–183.

Attanasio, C. R., Bacon, D. F., Cocchi, A., and Smith, S. 2001. A comparative evaluation of parallel garbage collectors. In *Fourteenth Annual Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science. Springer-Verlag, Cumberland Falls, KT.

Azatchi, H., Levanoni, Y., Paz, H., and Petrank, E. 2003. An on-the-fly mark and sweep garbage collector based on sliding view. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*. ACM SIGPLAN Notices. ACM Press, Anaheim, CA.

Bacon, D. F., Attanasio, C. R., Lee, H. B., Rajan, V. T., and Smith, S. 2001. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*. ACM SIGPLAN Notices. ACM Press, Snowbird, Utah.

Barabash, K., Ossia, Y., and Petrank, E. 2003. Mostly concurrent garbage collection revisited. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*. ACM SIGPLAN Notices. ACM Press, Anaheim, CA.

Boehm, H.-J. 2000. Reducing garbage collector cache misses. In *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, T. Hosking, Ed. ACM SIGPLAN Notices, vol. 36(1). ACM Press, Minneapolis, MN.

Boehm, H.-J. 2004. 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/.

Boehm, H.-J. and Weiser, M. 1988. Garbage collection in an uncooperative environment. *Software Practice and Experience 18,* 9, 807–820.

Chilimbi, T. M. and Larus, J. R. 1998. Using generational garbage collection to implement cache-conscious data placement. In *ISMM'98 Proceedings of the First International Symposium on Memory Management*, R. Jones, Ed. ACM SIGPLAN Notices, vol. 34(3). ACM Press, Vancouver, 37–48.

Cooper, E., Nettles, S., and Subramanian, I. 1992. 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*. ACM Press, San Francisco, CA, 43–52.

Dimpsey, R., Arora, R., and Kuiper, K. 2000. Java server performance: A case study of building efficient, scalable JVMs. *IBM Systems Journal 39,* 1, 151–174.

Domani, T., Kolodner, E., and Petrank, E. 2000. A generational on-the-fly garbage collector for Java. In *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation*. ACM SIGPLAN Notices. ACM Press, Vancouver.

Fitzgerald, R. and Tarditi, D. 2000. The case for profile-directed selection of garbage collectors. In *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, T. Hosking, Ed. ACM SIGPLAN Notices, vol. 36(1). ACM Press, Minneapolis, MN.

Geodesic Systems Inc. 2002. *REMIDI*. Geodesic Systems Inc. http://www.geodesic.com/-solutions/remidi.html.

Grunwald, D., Zorn, B., and Henderson, R. 1993. Improving the cache locality of memory allocation. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*. ACM SIGPLAN Notices, vol. 28(6). ACM Press, Albuquerque, NM, 177–186.

Jones, R. E. and Lins, R. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley.

KIM, T., CHANG, N., AND SHIN, H. 2000. Bounding worst case garbage collection time for embedded real-time systems. In *Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS 2000).*

MOON, D. A. 1984. Garbage collection in a large LISP system. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, G. L. Steele, Ed. ACM Press, Austin, TX, 235–245.

OSSIA, Y., BEN-YITZHAK, O., GOFT, I., KOLODNER, E. K., LEIKEHMAN, V., AND OWSHANKO, A. 2002. A parallel, incremental and concurrent GC for servers. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation.* ACM SIGPLAN Notices. ACM Press, Berlin, 129–140.

PRINTEZIS, T. 2001. Hot-swapping between a mark & sweep and a mark & compact garbage collector in a generational environment. In *Proceedings of the Java Virtual Machine Research and Technology Symposium.* USENIX.

SHAHAM, R., KOLODNER, E. K., AND SAGIV, M. 2000. On the effectiveness of GC in Java. In *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, T. Hosking, Ed. ACM SIGPLAN Notices, vol. 36(1). ACM Press, Minneapolis, MN.

SHUF, Y., SERRANO, M., GUPTA, M., AND SINGH, J. P. 2001. Characterizing the memory behavior of Java workloads: A structured view and opportunities for optimizations. In *SIGMETRICS'01.*

SUN MICROSYSTEMS. 2005. Tuning garbage collection with the 5.0 Java virtual machine. http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html.

SMITH, F. AND MORRISETT, G. 1998. Comparing mostly-copying and mark-sweep conservative collection. In *ISMM'98 Proceedings of the First International Symposium on Memory Management*, R. Jones, Ed. ACM SIGPLAN Notices, vol. 34(3). ACM Press, Vancouver, 68–78.

UNGAR, D. M. AND JACKSON, F. 1988. Tenuring policies for generation-based storage reclamation. *ACM SIGPLAN Notices 23,* 11, 1–17.

UNGAR, D. M. AND JACKSON, F. 1992. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems 14,* 1, 1–27.

WILSON, P. R. 1994. Uniprocessor garbage collection techniques. Tech. rep., University of Texas. Jan.

WILSON, P. R., JOHNSTONE, M. S., NEELY, M., AND BOLES, D. 1995. Dynamic storage allocation: A survey and critical review. In *Proceedings of International Workshop on Memory Management*, H. Baker, Ed. Lecture Notes in Computer Science, vol. 986. Springer-Verlag, Kinross, Scotland.

WILSON, P. R., LAM, M. S., AND MOHER, T. G. 1991. Effective static-graph reorganization to improve locality in garbage collected systems. *ACM SIGPLAN Notices 26,* 6, 177–191.

WILSON, P. R., LAM, M. S., AND MOHER, T. G. 1992. Caching considerations for generational garbage collection. In *Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming.* ACM Press, San Francisco, CA, 32–42.

ZORN, B. 1990. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Conference Record of the 1990 ACM Symposium on Lisp and Functional Programming.* ACM Press, Nice, France.

ZORN, B. 1991. The effect of garbage collection on cache performance. Tech. Rep. CU–CS–528–91, University of Colorado at Boulder. May.

ZORN, B. 1993. The measured cost of conservative garbage collection. *Software Practice and Experience 23*, 733–756.

Received June 2002, revised May 2004, accepted March 2005.