GCH: Hints for Triggering Garbage Collections*

Dries Buytaert, Kris Venstermans, Lieven Eeckhout, Koen De Bosschere

ELIS Department, Ghent University – HiPEAC member
St.-Pietersnieuwstraat 41, B-9000 Gent, Belgium
{dbuytaer,kvenster,leeckhou,kdb}@elis.UGent.be

Abstract. This paper shows that Appel-style garbage collectors often make suboptimal decisions both in terms of *when* and *how* to collect. We argue that garbage collection should be done when the amount of live bytes is low (in order to minimize the collection cost) and when the amount of dead objects is high (in order to maximize the available heap size after collection). In addition, we observe that Appel-style collectors sometimes trigger a nursery collection in cases where a full-heap collection would have been better.
Based on these observations, we propose *garbage collection hints (GCH)* which is a profile-directed method for guiding garbage collection. Offline profiling is used to identify favorable collection points in the program code. In those favorable collection points, the garbage collector dynamically chooses between nursery and full-heap collections based on an analytical garbage collector cost-benefit model. By doing so, GCH guides the collector in terms of *when* and *how* to collect. Experimental results using the SPECjvm98 benchmarks and two generational garbage collectors show that substantial reductions can be obtained in garbage collection time (up to 29X) and that the overall execution time can be reduced by more than 10%. In addition, we also show that GCH reduces the maximum pause times and outperforms user-inserted forced garbage collections.

1 Introduction

Garbage collection (GC) is an important subject of research as many of today's programming language systems employ automated memory management. Popular examples are Java and C#. Before discussing the contributions of this paper, we revisit some garbage collection background and terminology.

1.1 Garbage collection

An Appel-style generational copying collector divides the heap into two generations [2], a variable-size *nursery* and a *mature generation*. Objects are allocated from the nursery. When the nursery fills up, a *nursery collection* is triggered and the surviving objects are copied into the mature generation. When the objects

* The first two authors contributed equally to this paper.

are copied, the size of the mature generation is grown and the size of the nursery is reduced accordingly. Because the nursery size decreases, the time between consecutive collections also decreases and objects have less time to die. When the nursery size drops below a given threshold, a *full-heap collection* is triggered. After a full-heap collection all free space is returned to the nursery.

In this paper we consider two flavors of generational copying collectors, namely *GenMS* and *GenCopy* from JMTk [3]. GenMS collects the mature generation using the mark-sweep garbage collection strategy. The GenCopy collector on the other hand, employs a semi-space strategy to manage its mature generation. The semi-space collector copies scanned objects, whereas the mark-sweep collector does not. These Appel-style garbage collectors are widely used.

To partition the heap into generations, the collector has to keep track of references between different generations. Whenever an object in the nursery is assigned to an object in the mature generation—*i.e.*, there is a reference from an object in the mature space to an object in the nursery space—this information is tracked by using a so-called *remembered set*. When a nursery collection is triggered the remembered set must be processed to avoid erroneously collecting nursery objects that are referenced only from the mature generation.

1.2 Paper contributions

While implicit garbage collection offers many benefits, for some applications the time spent reclaiming memory can account for a significant portion of the total execution time [1]. Although garbage collection research has been a hot research topic for many years, little research has been done to decide *when* and *how* garbage collectors should collect.

With Appel-style collectors, garbage is collected when either the heap or a generation is full. However, to reduce the time spent in GC, the heap is best collected when the live ratio is low: the fewer live objects, the fewer objects need to be scanned and/or copied, the more memory there is to be reclaimed, and the longer we can postpone the next garbage collection run. In this paper, we show that collecting at points where the live ratio is low, can yield reductions in GC time.

In addition, when using an Appel-style collector with two generations, a decision needs to be made whether to trigger a full-heap or nursery collection. We found that triggering nursery collections until the nursery size drops below a certain threshold is sometimes suboptimal. In this paper, we show how to trade off full-heap collections and nursery collections so that performance improves.

The approach presented in this paper to decide *when* and *how* to collect, is called *garbage collection hints (GCH)* and works as follows. GCH first determines *favorable collection points (FCPs)* for a given application through offline profiling. A favorable collection point is a location in the application code where the cost of a collection is relatively cheap. During program execution a cost function is then computed in each FCP to determine the best GC strategy: postpone GC, perform a nursery GC, or perform a full-heap GC. Our experimental results using the SPECjvm98 benchmarks and two generational collectors show that GCH

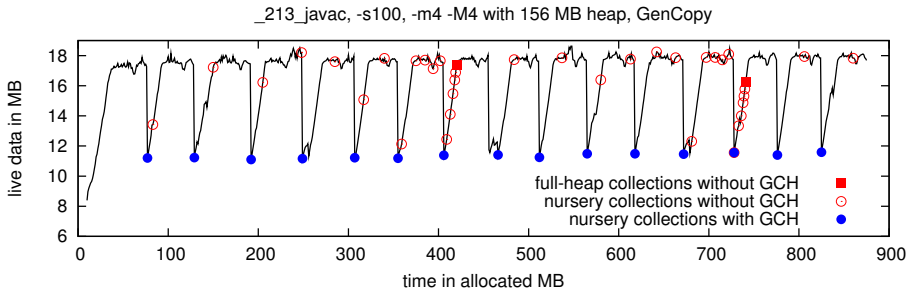

Fig. 1. Garbage collection points with and without GCH.

can reduce the garbage collector time by up to 29X and can improve the overall execution time by more than 10%.

Figure 1 illustrates why GCH actually works for the `_213_javac` benchmark. This graph shows the number of live bytes as a function of the number of allocated bytes. The empty circles denote nursery collections and the squares denote full-heap collections when GCH is not enabled. Without GCH, GC is triggered at points where the number of live bytes is not necessarily low. In fact, the maximum GC time that we observed on our platform for these GC points is 225ms; and 12MB needs to be copied from the nursery to the mature generation. The GC time for a full-heap collection takes 330ms. When GCH is enabled (see the filled circles in Figure 1), garbage gets collected when the amount of live bytes reaches a minimum, *i.e.*, at an FCP. The GC time at an FCP takes at most 4.5ms since only 126KB needs to be copied. From this example, we observe two key features why GCH actually works: (i) GCH preferably collects when the amount of live data on the heap is low, and (ii) GCH eliminates full-heap collections by choosing to perform (cheaper) nursery collections at more valuable points in time.

The main contributions of this paper are as follows.

- We show that GC is usually not triggered when the amount of live data is low, *i.e.*, when the amount of garbage collection work is minimal.
- We show that the collector does not always make the best decision when choosing between a nursery and a full-heap collection.
- We propose GCH which is a feedback-directed technique based on profile information that provides hints to the collector about *when* and *how* to collect. GCH tries to collect at FCPs when the amount of live data is minimal and dynamically chooses between nursery and full-heap collections. The end result is significant reductions in GC time and improved overall performance. GCH is especially beneficial for applications that exhibit a recurring phase behavior in the amount of live data allocated during program execution.
- We show that GCH reduces the pause time during garbage collection.

– And finally, we show that for our experimental setup, GCH improves overall performance compared to forced programmer-inserted GCs. The reason is that GCH takes into account the current live state of the heap whereas forced programmer-inserted GCs do not.

The remainder of this paper is organized as follows. Section 2 presents an overview of our experimental setup. In section 3, we describe the internals of GCH. The results are presented in section 4 after which we discuss related work in section 5. Finally, some conclusions are presented in section 6.

2 Experimental setup

2.1 Java virtual machine

We use the Jikes Research Virtual Machine 2.3.2 (RVM) [4] on an AMD Athlon XP 1500+ at 1.3 GHz with a 256KB L2-cache, 1GB of physical memory, running Linux 2.6. Jikes RVM is a Java virtual machine (VM) written almost entirely in Java. Jikes RVM uses a compilation-only scheme for translating Java bytecodes to native machine instructions. For our experiments we use the *FastAdaptive* profile: all methods are initially compiled using a baseline compiler, and sampling is used to determine which methods to recompile using an optimizing compiler.

Because Jikes RVM is written almost entirely in Java, internal objects such as those created during class loading or those created by the runtime compilers are allocated from the Java heap. Thus, unlike with conventional Java virtual machines the heap contains both application data as well as VM data. We found that there is at least 8MB of VM data that is quasi-immortal. The presence of VM data has to be taken into account when interpreting the results presented in the remainder of this work.

Jikes RVM's memory management toolkit (JMTk) [3] offers several GC schemes. While the techniques presented in this paper are generally applicable to various garbage collectors, we focus on the *GenMS* and *GenCopy* collectors. Both are used in Jikes RVM's production builds that are optimized for performance.

To get around a bug in Jikes RVM 2.3.2 we increased the maximum size of the remembered set to 256MB. In order to be able to model the shrinking/growing behavior of the heap accurately, we made one modification to the original RVM. We placed the remembered set outside the heap.

Performance is measured using the Hardware Performance Monitor (HPM) subsystem of Jikes RVM. HPM uses (i) the `perfctr`[1] Linux kernel patch, which provides a kernel module to access the processor hardware, and (ii) PAPI [5], a library to capture the processor's performance counters. The hardware performance counters keep track of the number of retired instructions, elapsed clock cycles, etc.

[1] `http://user.it.uu.se/~mikpe/linux/perfctr/`

2.2 Benchmarks

To evaluate our mechanism, we use the SPECjvm98[2] benchmark suite. The SPECjvm98 benchmark suite is a client-side Java benchmark suite consisting of seven benchmarks, each with three input sets: `-s1`, `-s10` and `-s100`. With the `-m` and `-M` parameters the benchmark can be configured to run multiple times without stopping the VM. Garbage collection hints work well for long running applications that show recurring phase behavior in the amount of live data. To mimic such workloads with SPECjvm98, we use the `-s100` input set in conjunction with running the benchmarks four times (`-m4 -M4`).

We used all SPECjvm98 benchmarks except one, namely `_222_mpegaudio`, because it merely allocates 15MB each run and triggers few GCs. The other benchmarks allocate a lot more memory.

All SPECjvm98 benchmarks are single-threaded except for `_227_mtrt` which is a multi-threaded raytracer. Note that because both Jikes RVM's sampling mechanism and the optimizing compiler run in separate threads all benchmarks are non-deterministic.

We ran all experiments with a range of different heap sizes. We vary the heap size between the minimum feasible heap size and the heap size at which our mechanism stops triggering GCs or shows constant behavior.

Some benchmarks, such as `_213_javac`, use *forced garbage collections* triggered through calls to `java.lang.System.gc()`. We disabled forced garbage collections unless stated otherwise.

3 Garbage collection hints

Our garbage collection hints approach consists of an offline and an online step, see Figure 2. The offline step breaks down into two parts: (i) offline profiling of the application and (ii) garbage collector analysis. The offline profiling computes the live/time function of the application, *i.e.*, the amount of live bytes as a function of the amount of bytes allocated. Based on this live/time function, favorable collection points (FCPs) can be determined. Determining the FCPs is a one-time cost per application. The garbage collector analysis characterizes the collection cost for a particular garbage collector and application, *i.e.*, the amount of time needed to process a given amount of live bytes. This is dependent on the collector and the platform on which the measurements are done. In the online part of GCH, the methods that have been identified as FCPs are instrumented to invoke a cost-benefit model that helps the garbage collector make decisions about *when* and *how* to collect. This decision making is based on the amount of heap space available, the live/time function of the application and the characteristics of the garbage collector. The following subsections discuss GCH in more detail.

[2] `http://www.spec.org/jvm98/`

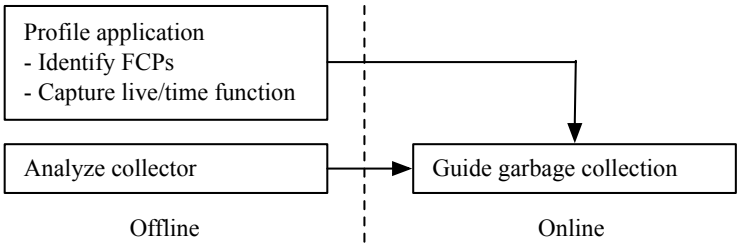

Fig. 2. An overview of the GCH methodology.

3.1 Program analysis

Live/dead ratio behavior. The first step of the offline profiling is to collect the live/time function which quantifies the number of live bytes as a function of the bytes allocated so far. Moreover, we are interested in linking the live/time function to methods calls. We modified Jikes RVM to timestamp and report all method entries and exits. For each method invocation, we want to know how many objects/bytes died and how many objects are live. Therefore, a lifetime analysis is required at every point an object could have died. There are two reasons for an object to die: (i) an object's last reference is overwritten as a result of an assignment operation, or (ii) an object's last reference is on a stack frame and the stack frame gets popped because the frame's method returns or because an exception is thrown. To avoid having to do a lifetime analysis for every assignment operation, method return and exception, we used a modified version of the Merlin trace generator [6] that is part of Jikes RVM. Merlin is a tool that precisely computes every object's last reachable time. It has been modified to use our alternative timestamping method to correlate object death with method invocations.

Figure 3 shows the live/time function for the various benchmarks. As can be seen from these graphs, the live/time function shows recurring phase behavior. This recurring phase behavior will be exploited through GCH. Applications that do not exhibit a phased live/time function are not likely to benefit from GCH. Next, the live/time function is used to select FCPs and to compute the FCP live/time patterns.

Favorable collection points. For a method to represent a favorable collection point (FCP), it needs to satisfy three criteria:

1. An FCP's invocation should correspond to a local minimum in terms of the number of live bytes. In other words, we need to select methods that are executed in the minima of the live/time function. This will allow GCH to collect garbage with minimal effort.

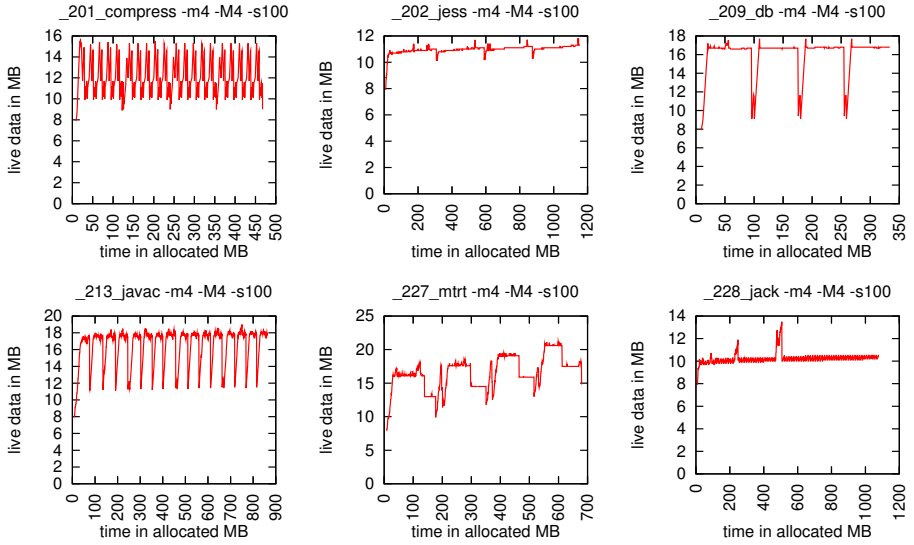

Fig. 3. The live/time function for the various benchmarks: number of live bytes as a function of the number of bytes allocated.

2. An FCP should not be executed frequently. To minimize the overhead of the instrumentation code, FCPs that represent cold methods are preferred. A method that gets executed *only* in local minima is an ideal FCP.
3. The live/time pattern following the execution of the FCP should be fairly predictable, *i.e.*, each time the FCP gets executed, the live/time function should have a more or less similar shape after the FCP.

Given the live/time function, selecting FCPs is fairly straightforward. Table 1 shows the selected FCPs that we selected for the SPECjvm98 benchmarks. Some benchmarks have only one FCP (see for example Figure 1 for `_213_javac`); others such as `_227_mtrt` have three FCPs.

To illustrate the potential benefit of FCPs, Figure 4 plots the maximum time spent in GC when triggered at an FCP and when triggered otherwise. We make a distinction between full-heap and nursery collections, and plot data for a range of heap sizes. For most benchmarks we observe that the maximum GC time spent at an FCP is substantially lower than the GC time at other collection points. This reinforces our assumption that collecting at an FCP is cheaper than collecting elsewhere. However, there are two exceptions, `_201_compress` and `_228_jack`, for which GC time is insensitive to FCPs. For `_201_compress`, this is explained by the fact that the live/time function shown in Figure 3 is due to a few objects that are allocated in the Large Object Space (LOS). Because objects in the LOS never get copied, GCH cannot reduce the copy cost. Furthermore, because there are only a few such objects it will not affect the scan cost either. For `_228_jack`,

Benchmark	Favorable collection points
_201_compress	spec.io.FileInputStream.getContentLength()
_202_jess	spec.benchmarks._202_jess.jess._undefrule.<init>()V
	spec.harness.BenchmarkTime.toString()Ljava/lang/String;
_209_db	spec.harness.Context.setBenchmarkRelPath(Ljava/lang/String;)V
	spec.io.FileInputStream.getCachingtime()J
_213_javac	spec.benchmarks._213_javac.ClassPath.<init>(Ljava/lang/String;)V
_227_mtrt	spec.io.TableOfExistingFiles.<init>()V
	spec.harness.Context.clearIOtime()V
	spec.io.FileInputStream.getCachingtime()J
_228_jack	spec.benchmarks._228_jack.Jack_the_Parser_Generator_Internals.-compare(Ljava/lang/String;Ljava/lang/String;)V

Table 1. The selected FCPs for each of the benchmark applications. The method descriptors use the format specified in [7].

the height of the live/time function's peaks is very low, see Figure 3. Because _201_compress and _228_jack are insensitive to FCPs we exclude them from the other results that will be presented in this paper. (In fact, we applied GCH to these benchmarks and observed neutral impact on overall performance. Due to space constraints, we do not to include these benchmarks in the rest of this paper.)

It is also interesting to note that for _209_db, a nursery collection can be more costly than a full-heap collection. This is due to the fact that the remembered set needs to be scanned on a nursery collection. As such, for _209_db a full-heap collection can be more efficient than a nursery collection. This is exploited through GCH.

FCP's live/time pattern. For each unique FCP we have to capture the *live/time pattern* following the FCP. This is a slice of the live/time function following the FCP that recurs throughout the complete program execution. We sample the FCP's live/time pattern at a frequency of one sample per 0.5MB of allocated memory and use it as input for the cost-benefit model. An FCP's live/time pattern is independent of the heap size (the same information is used for all heap sizes) and is independent of the collection scheme (the same information is used for both GenMS and GenCopy). And it only needs to be computed once for each benchmark.

3.2 Collector analysis

So far, we discussed the offline application profiling that is required for GCH. We now discuss the characterization of the garbage collector. This characterization will be used in the cost model that will drive the decision making in GCH during program execution. The characterization of the garbage collector quantifies the cost of a collection. We identify three cost sources: the cost of a full-heap collection, the cost of a nursery collection and the cost of processing the remembered set. The cost functions take as input the amount of live data and output the estimated collection time. These cost functions are dependent

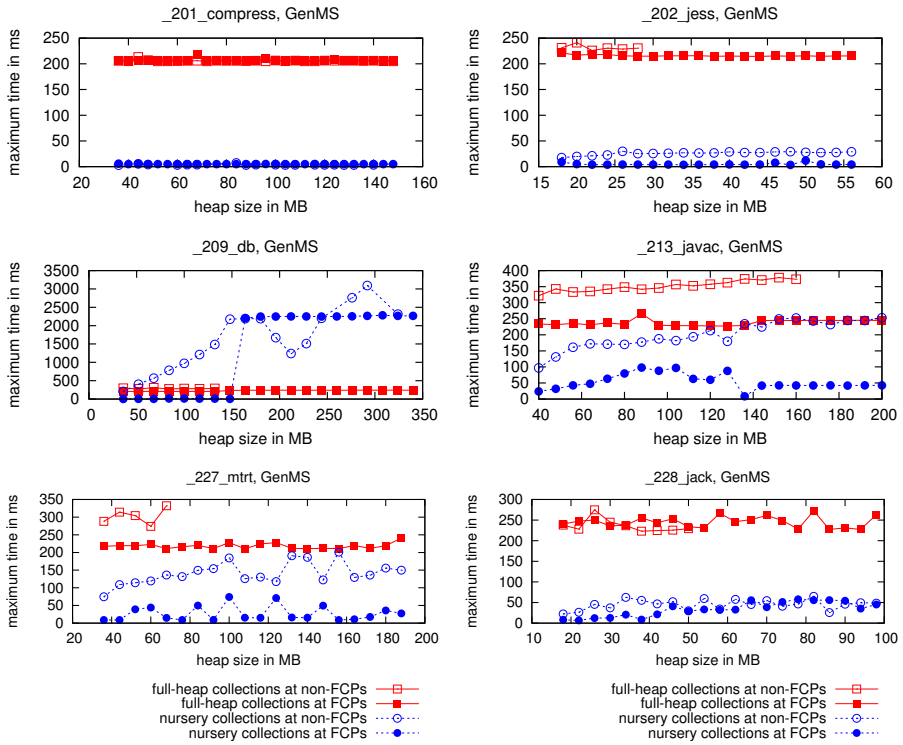

Fig. 4. The maximum times spent in garbage collection across different heap sizes for each of the different scenarios.

on the application, the collector and the given platform (VM, microprocessor, etc.).

Figure 5 shows how the cost functions are to be determined for the GenMS and GenCopy collectors. The graphs are obtained by running the benchmarks multiple times with different heap sizes using instrumented collectors. In these graphs we make a distinction between nursery collections, full-heap collections and processing of the remembered set. Hence, the processing times on the nursery collection graphs do not include the time required to process the remembered sets.

GC time can be modeled as a linear function of the amount of live data for both collectors. In other words, the scanning and copying cost is proportional to the amount of live bytes. Likewise, the processing cost of the remembered set can be modeled as a linear function of its size. In summary, we can compute linear functions that quantify the cost of a nursery collection, full-heap collection and processing of the remembered set.

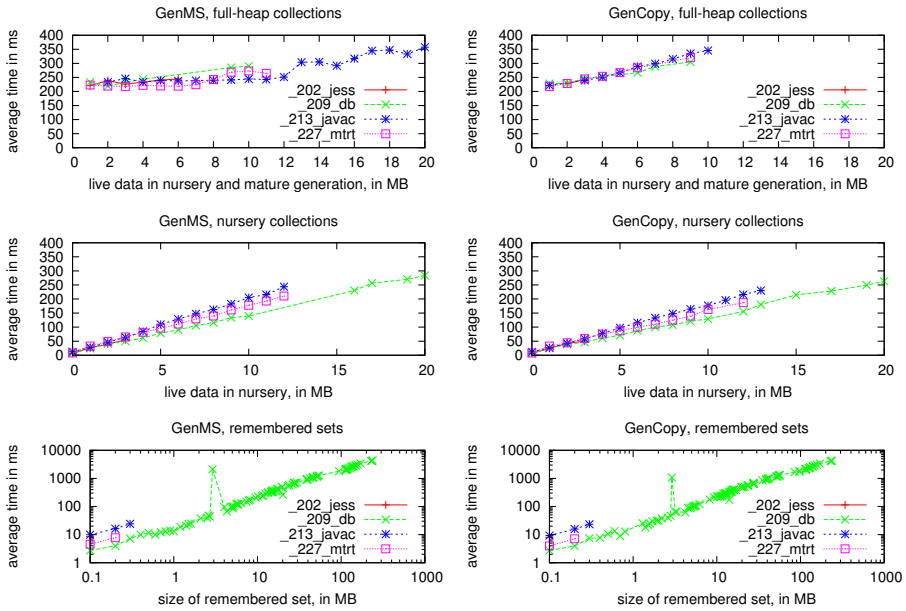

Fig. 5. The cost of a nursery and full-heap collection in terms of the amount of copied/live data.

In this paper we employ both application-specific cost functions as well as cross-application cost functions. In fact, on a specialized system with dedicated long running applications, it is appropriate to consider a cost function that is specifically tuned for the given application. Nevertheless, given the fact that the cost functions appear to be fairly similar across the various applications, see Figure 5, choosing application-independent or cross-application cost functions could be a viable scenario for general-purpose environments. In this paper we evaluate both scenarios.

3.3 GCH at work

The information that is collected through our offline analysis is now communicated to the VM to be used at runtime. Jikes RVM reads all profile information at startup. This contains (i) a list of methods that represent the FCPs, (ii) the live/time pattern per FCP, and (iii) the cost functions for the given garbage collector. Jikes RVM is also modified to dynamically instrument the FCPs. The instrumentation code added to the FCPs examines whether the current FCP should trigger a GC. The decision to collect or not is a difficult one as there exists a trade-off between reducing the amount of work per collection and having to collect more frequently. Clearly, triggering a collection will have an effect on subsequent collections. Because GC is invoked sooner due to GCH than without

GCH, additional collections might get introduced. On the other hand, triggering a collection at an FCP can help reduce the GC overhead. A collection at an FCP will generally introduce modest pause times compared to collections at other points. Moreover, triggering a full-heap collection grows the nursery size and gives objects more time to die, while triggering a nursery collection when few objects are live will result in the mature generation filling up slower, reducing the need for more expensive full-heap collections.

To make this complex trade-off, the instrumentation code in the FCPs implements an *analytical cost-benefit model*. The cost-benefit model estimates the total GC time for getting from the current FCP to the end of its FCP's live/time pattern. The cost-benefit model considers the following three scenarios: (i) do not trigger a GC in the current FCP, (ii) trigger a full-heap GC, or (iii) trigger a nursery GC. For each of these three scenarios, the cost-benefit model computes the total GC time ($C_{total,i}$ with i one of the three scenarios above) by *analytically simulating* how the heap will evolve through the FCP's live/time pattern. The total GC time can be split up in a number of components: $C_{total,i} = C_{FCP,i} + \sum_{j=1}^{n} C_{profile,j} + C_{end}$. We now explain each component in more detail. First, the cost-benefit model computes the GC cost *in the current FCP* under the following three scenarios:

(i) The cost for not triggering a GC is obviously zero. The available heap size remains unchanged. So, $C_{FCP,notrigger} = 0$.

(ii) For computing the cost for triggering a full-heap collection in the current FCP, we first calculate the number of live bytes at the current FCP, $livebytes_{FCP}$. We get this information from the live/time pattern. We subsequently use the full-heap GC cost function to compute the GC time given the amount of live data in the current FCP. The available heap size after the current (hypothetical) full-heap collection then equals the maximum heap size minus the amount of live data in the current FCP. The cost of a full-heap collection at the current FCP, $C_{FCP,fullheap}$, can be computed using the linear function of the form $A \times x + B$ where are A and B are derived from Figure 5. So, $C_{FCP,fullheap} = A_{fullheap} \times livebytes_{FCP} + B_{fullheap}$.

(iii) To compute the cost for triggering a nursery GC in the current FCP, we assume that the amount of live bytes in the nursery at that FCP is close to zero. The GC cost is then computed based on the nursery GC cost function. This GC cost is incremented by an extra cost due to processing the remembered set. This extra cost is proportional to the size of the remembered set, which is known at runtime at an FCP. The heap size that was occupied by the nursery becomes available for allocation. The cost of a nursery collection at the current FCP, $C_{FCP,nursery}$, equals $(A_{nursery} \times livebytes_{FCP} + B_{nursery}) + (A_{remset} \times remsetsize_{FCP} + B_{remset})$ with the A and B coefficients extracted from Figure 5.

In the second step of the cost-benefit model we compute the cost of additional collections over the FCP's live/time pattern for each of the three scenarios. In fact, for each scenario, the cost-benefit model analytically simulates how the heap will evolve over time when going through an FCP's live/time pattern.

Therefore, we compute when the (nursery) heap will be full—when the application has allocated all memory available in the heap. In case the system would normally trigger a full collection (*i.e.*, when the nursery size drops below the given threshold), we need to compute the cost of a full-heap collection. This is done the same way as above, by getting the amount of live data from the FCP's live/time pattern—note that we linearly interpolate the live/time pattern—and use the full-heap GC cost function to compute its cost. In case the nursery size is above the given threshold, we need to compute the cost of a nursery collection. Computing the cost for a nursery collection is done by reading the number of live bytes from the FCP's live/time pattern and subtracting the number of live bytes in the previous GC point; this number gives us an estimate for the amount of live data in the nursery. This estimated amount of live nursery data is used through the nursery GC cost function to compute an estimated nursery GC cost. The number of terms n in $\sum_{j=1}^{n} C_{profile,j}$ equals the number of analytical simulations we perform going through the FCP's live/time pattern. If a sufficient amount of the heap is still free, or being freed by the simulated GC in the FCP, it is possible that $n = 0$.

When the end of the FCP's live/time pattern is reached within the model, an end cost C_{end} is added to the aggregated GC cost calculated so far. The purpose of the end cost is to take into account whether the next (expected) collection will be close by or far away. This end cost is proportional to the length of the FCP's live/time pattern after the last simulated GC divided by the fraction of calculated unused nursery space at that same last simulated GC point. The more data still needs to be allocated, the closer the next GC, the higher the end cost.

After computing all the costs for each of the three scenarios, the scenario that results in the minimal total cost is chosen. As such, it is decided whether a nursery, a full-heap or no collection needs to be triggered in the current FCP.

Note that the cost-benefit model presented above is specifically developed for GenMS and GenCopy, two Appel-style generational garbage collectors with a variable nursery size. However, a similar cost-benefit model could be constructed for other collectors.

3.4 GCH across inputs

GCH is a profile-driven garbage collection method which implies that the input used for the offline profiling run is typically different from the input used during online execution. Getting GCH to work across inputs needs a few enhancements since the size and the height of an FCP's live/time pattern varies across inputs; the general shape of the FCP's live/time pattern however is relatively insensitive to the given input. We define the height of an FCP's live/time pattern as the difference in live data at the top of an FCP's live/time pattern and at the FCP itself. For example, for `_213_javac`, see Figure 1, the height is approximately 6MB. The size of an FCP's live/time pattern is defined as the number of allocated bytes at an FCP's live/time pattern; this is approximately 60MB for `_213_javac`, see also Figure 1. To address this cross-input issue we just scale the size and the height of the FCP live/time pattern. In practice, the amount of live data at the

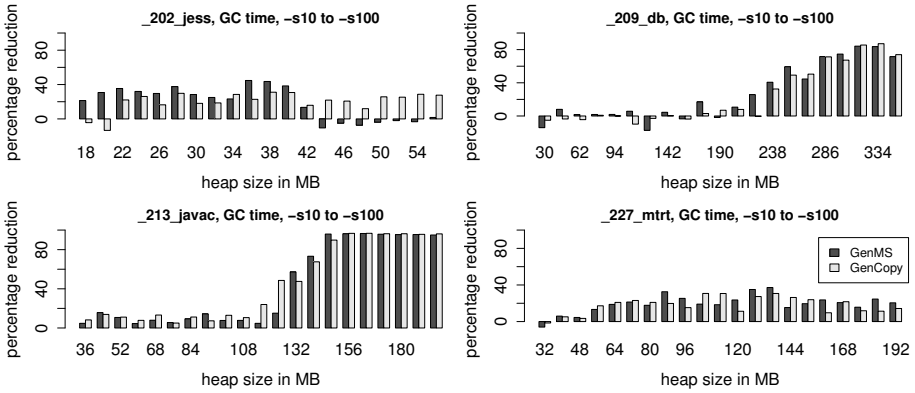

Fig. 6. Reduction in time spent in garbage collection through GCH across inputs. The profile input is `-s10`; the reported results are for `-s100`. Benchmark-specific GC cost functions are used.

top of an FCP's live/time pattern can be computed at runtime when a GC is triggered at the top of an FCP's live/time pattern. The amount of allocated bytes over an FCP's live/time pattern can be computed at run-time as well. These scaling factors are then to be used to rescale the data in the FCP live/time pattern.

4 Evaluation

This section evaluates GCH through a number of measurements. First, we measure the GC time reduction. Second, we evaluate the impact on overall performance. Third, we quantify the impact on pause time. Fourth, we compare forced GC versus GCH. Finally, we quantify the runtime overhead due to GCH.

4.1 Garbage collection time

In order to evaluate the applicability of GCH, we have set up the following experiment. We used the profile information from the `-s10` run to drive the execution of the `-s100` run after cross-input rescaling as discussed in section 3.4.

Figure 6 shows the reduction through GCH in GC time over a range of heap sizes where reduction is defined as $100 \times (1 - \frac{time_{GCH}}{time_{old}})$. Each reduction number is an average number over three runs; numbers are shown for both GenMS and GenCopy. Figure 6 shows that GCH improves GC time for both collectors and for nearly all heap sizes. For both collectors, GCH achieves substantial speedups in terms of garbage collection time, up to 29X for `_213_javac` and 10X for `_209_db`.

The sources for these speedups are twofold. First, GCH generally results in fewer collections than without GCH, see Table 2 which shows the average

	GenMS collector				GenCopy collector			
	Full-heap		Nursery		Full-heap		Nursery	
Benchmark	no GCH	GCH	no GCH	GCH	no GCH	GCH	no GCH	GCH
_202_jess	0	1	245	186	2	3	349	294
_209_db	1	3	16	14	2	4	25	25
_213_javac	2	2	80	62	6	5	93	61
_227_mtrt	0	1	45	36	2	2	81	67

Table 2. The average number of garbage collections across all heap sizes with and without GCH.

number of GCs over all heap sizes; we observe fewer GCs with GCH for all benchmarks except one, namely _209_db for which the number of collections remains unchanged with and without GCH (we will discuss _209_db later on). For _213_javac we observe a 30% reduction in the number of collections. The second reason for these GC time speedups is the reduced work at each point of GC. This was already mentioned in Figure 4.

For _202_jess, GCH only occasionally triggers a collection for heap sizes larger than 44MB for GenMS. GCH then causes the same GC pattern as running without GCH, as the non-GCH directed pattern already is the optimal one.

Note that for _209_db, GCH is capable of substantially reducing the GC time for large heap sizes. The reason is not the reduced number of collections, but the intelligent selection of full-heap collections instead of nursery collections. The underlying reason is that _209_db suffers from a very large remembered set. GCH triggers more full-heap collections that do not suffer from having to process remembered sets, see Table 2. A full-heap collection typically only takes 250ms for this application whereas a nursery collection can take up to 2,000ms, see Figure 4. Note that the remembered set increases with larger heap sizes which explains the increased speedup for larger heap sizes. While the large remembered sets themselves are the consequence of the fact that JMTk uses sequential store buffers without a space cap, it shows that our analytical framework is robust in the face of extreme cases like this.

For _213_javac, GCH reduces the total number of collections. In addition, the cost of a nursery collection at an FCP is much cheaper than a nursery collection at another execution point because less data needs to be scanned and copied. As mentioned with our introductory example in Figure 1, at most 126KB needs to be copied at an FCP which takes about 4.5ms while up to 12MB needs to be copied otherwise, which takes about 225ms. From heap sizes of 132MB on, no other GCs are required than those triggered by GCH. As a direct result, the collector's performance improves by a factor 29.

Remind that because of the way Jikes RVM works, the heap contains both application data and VM data. We believe that our technique would be even more effective in a system where the collector does not have to trace VM data. In such a system, full-heap collections would generally be cheaper opening up extra opportunities to replace nursery collections by full-heap collections.

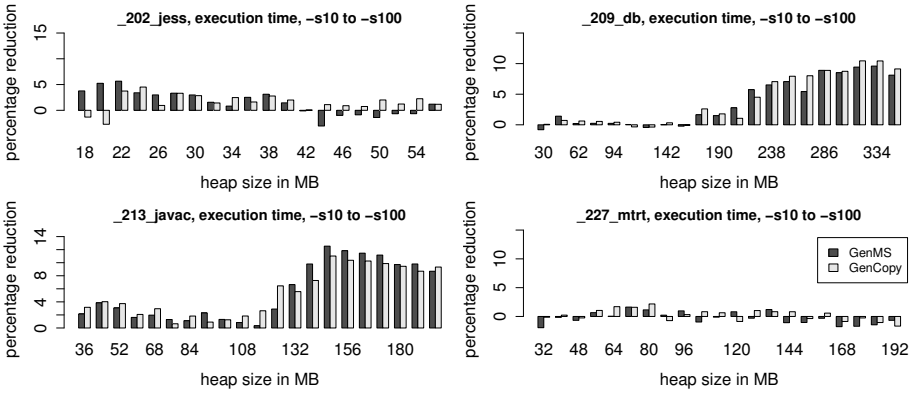

Fig. 7. Performance improvement in total execution time through GCH across inputs. The profile input is `-s10`; the reported results are for `-s100`. Benchmark-specific GC cost functions are used.

4.2 Overall execution time

Figure 7 depicts the impact of GCH on the overall execution time. For `_227_mtrt`, the total execution time is more or less unaffected through GCH because the time spent collecting garbage is only a small fraction of the total execution time. The small slowdowns or speedups observed for `_227_mtrt` are probably due to changing data locality behavior because of the changed GCs. However, for `_202_jess`, `_209_db` and `_213_javac`, performance improves by up to 5.7%, 10.5% and 12.5%, respectively. For these benchmarks, the GC time speedups translate themselves in overall performance speedup.

4.3 Generic cost functions

So far we assumed application-specific GC cost functions, *i.e.*, the cost function for a nursery collection and a full-heap collection as well as the cost associated with scanning the remembered set was assumed to be application-specific. This is a viable assumption for application-specific designs. However, for application-domain specific (involving multiple applications) or general-purpose systems, this may no longer be feasible. Figures 8 and 9 evaluate the performance of GCH in case cross-application GC cost functions are employed instead of application-specific GC cost functions; this is done for the garbage collection time as well as for the overall execution time, respectively. Comparing these figures against Figures 6 and 7, we observe that there is no significant difference between the performance that is obtained from application-specific versus generic cost functions. As such, we conclude that GCH is robust to generic cost functions.

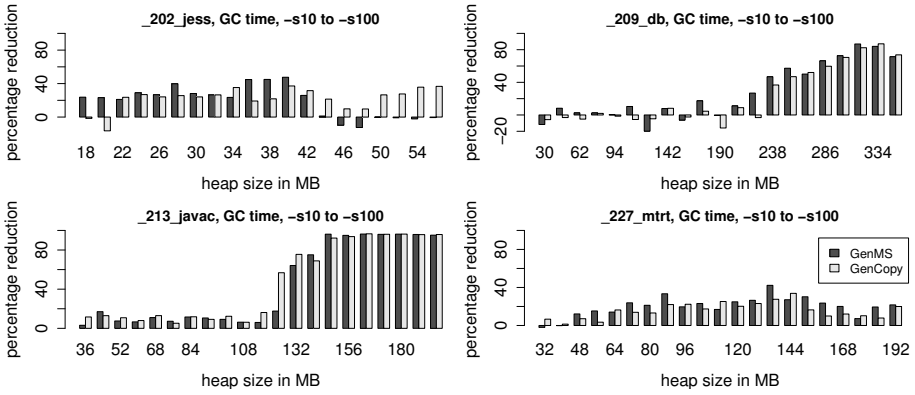

Fig. 8. Reduction in time spent in garbage collection through GCH across inputs using generic cost functions. The profile input is `-s10`; the reported results are for `-s100`.

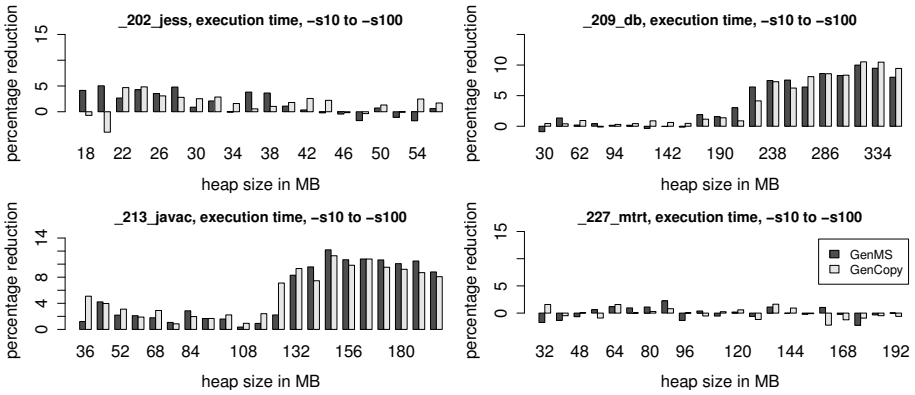

Fig. 9. Performance improvement in total execution time through GCH across inputs using generic cost functions. The profile input is `-s10`; the reported results are for `-s100`.

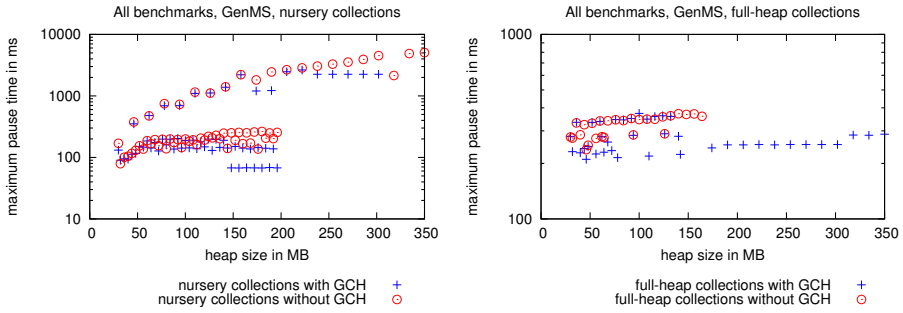

Fig. 10. The maximum pause time with and without GCH. The graph on the left shows the maximum pause time for nursery collections; the graph on the right shows the maximum pause time for full-heap collections.

4.4 Pause time

An important metric when evaluating garbage collection techniques is pause time. Pause time is especially important for interactive and (soft) real-time applications. Figure 10 presents the maximum pause times over all benchmark runs as a function of the heap size. We make a distinction between nursery and full-heap collections, and between with-GCH and without-GCH. The graphs show that the maximum pause time is reduced (or at least remains unchanged) through GCH. Note that the vertical axes are shown on a logarithmic scale. As such, we can conclude that GCH substantially reduces the maximum pause time that is observed during program execution.

4.5 Forced versus automatic garbage collection

A programmer can force the VM to trigger a GC by calling the `java.lang.System.gc()` method. We refer to such collections as *forced garbage collections*. One of the benchmarks that we studied, namely `_213_javac`, triggers forced GCs. When run with `-s100 -m4 -M4`, 21 forced GCs are triggered. Figure 11 shows the GC times normalized to the fastest time for `_213_javac` using the GenCopy collector for a range of heap sizes. The graph depicts the collection times under four scenarios: the VM ignores forced collections, the forced collections are nursery collections, the forced GCs are full-heap collections, and GCH is enabled. According to Figure 11, forced collections can either improve or reduce performance compared to not using forced collections. More specifically, if the forced GCs are full-heap collections, performance is typically reduced; if the forced GCs are nursery collections, performance typically improves—for large heap sizes, performance even improves dramatically. Another important observation from Figure 11 is that GCH performs better than all other strategies for all heap sizes. This can be explained by the fact that while there are 21 forced

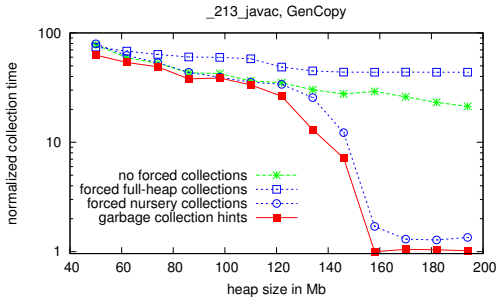

Fig. 11. Garbage collection time under forced garbage collection versus automatic garbage collection versus GCH.

collections, there are only 15 FCPs (see Figure 1). GCH correctly triggers no more than 15 times and takes the current state of the heap into account when making GC decisions. In summary, there are three reasons why GCH is preferable over forced collections. First, in large applications it can be difficult to identify favorable collection points without automated program analysis as presented in this paper. Second, forced GCs, in contrast to GCH, do not take into account the available heap size when deciding whether a GC should be triggered. Third, in our experimental setup, GCH is capable of deciding how garbage should be collected at runtime, *i.e.*, whether a nursery or full-heap collection should be triggered which is impossible through forced GCs.

4.6 Run-time system overhead

To explore the run-time overhead of our system, we compare the performance of a without-GCH Jikes RVM versus a with-GCH Jikes RVM. In the with-GCH version, the profile information is read, the FCPs are instrumented and at each invocation of an FCP the cost-benefit model is computed, however, it will never trigger a collection. For computing the overhead per benchmark, each benchmark is run multiple times and the average overhead is computed over these runs. Table 3 shows the average overhead over all heap sizes with both collectors. The average overhead over all benchmarks is 0.3%; the maximum overhead is 1.3% for `_227_mtrt`. The negative overheads imply that the application ran faster with instrumentation that without instrumentation. We thus conclude that the overhead of GCH is negligible.

5 Related work

We now discuss previously proposed GC strategies that are somehow related to GCH, *i.e.*, all these approaches implement a mechanism to decide when *or* how

Benchmark	GenMS	GenCopy
_202_jess	-0.1%	0.8%
_209_db	0.0%	-0.6%
_213_javac	0.2%	0.3%
_227_mtrt	1.3%	0.6%

Table 3. The run-time overhead of GCH.

to collect. The work presented in this paper differs from previous work in that we combine the decision of both when *and* how to collect in a single framework.

The Boehm-Demers-Weiser (BDW) [8] garbage collector and memory allocator include a mechanism that determines whether to collect garbage or to grow the heap. The decision whether to collect or grow the heap is based on a static variable called the *free space divisor (FSD)*. If the amount of heap space allocated since the last garbage collection exceeds the heap size divided by FSD, garbage is collected. If not, the heap is grown. Brecht *et al.* [9] extended the BDW collector by taking into account the amount of physical memory available and by proposing dynamically varying thresholds for triggering collections and heap growths.

Wilson *et al.* [10] observe that (interactive) programs have phases of operation that are compute-bound. They suggest that tagging garbage collection onto the end of larger computational pauses, will not make those pauses significantly more disruptive. While the main goal of their work is to avoid or mitigate disruptive pauses, they reason that at these points, live data is likely to be relatively small since objects representing intermediate results of the previous computations have become garbage. They refer to this mechanism as *scavenge scheduling* but present no results.

More recently, Ding *et al.* [11] presented preliminary results of a garbage collection scheme called *preventive memory management* that also aims to exploit phase behavior. They unconditionally force a garbage collection at the beginning of certain execution phases. In addition, they avoid garbage collections in the middle of a phase by growing the heap size unless the heap size reaches the hard upper bound of the available memory. They evaluated their idea using a single Lisp program and measured performance improvements up to 44%.

Detlefs *et al.* [13] present the garbage-first garbage collectors which aims at satisfying soft real-time constraints. Their goal is to spend no more than x ms during garbage collection for each y ms interval. This is done by using a collector that uses many small spaces and a *concurrent marker* that keeps track of the amount of live data per space. The regions containing most garbage are then collected first. In addition, collection can be delayed in their system if they risk violating the real-time goal.

Velasco *et al.* [14] propose a mechanism that dynamically tunes the size of the copy reserve of an Appel collector [2]. Tuning the copy reserve's size is done based on the ratio of surviving objects after garbage collection. Their technique achieves performance improvements of up to 7%.

Stefanovic *et al.* [15] evaluate the older-first generational garbage collector which only copies the oldest objects in the nursery to the mature generation. The youngest objects are not copied yet; they are given enough time to die in the nursery. This could be viewed of as a way deciding when to collect.

Recent work [16,17,18] selects the most appropriate garbage collector during program execution out of a set of available garbage collectors. As such, the garbage collector is made adaptive to the program's dynamic execution behavior. The way GCH triggers nursery or full-heap collections could be viewed as a special form of what these papers proposed.

6 Summary and future work

This paper presented garbage collection hints which is a profile-directed approach to guide garbage collection. The goal of GCH is to guide in terms of *when* and *how* to collect. GCH uses offline profiling to identify favorable collection points in the program code where the amount of live data is relatively small (in order to reduce the amount of work per collection) and the amount of dead bytes is relatively large (in order to increase the amount of available heap after collection). Triggering collections in these FCPs can reduce the number of collections as well as the amount of work per collection. Next to guiding when to collect, GCH also uses an analytical cost-benefit model to decide how to collect, *i.e.*, whether to trigger a nursery or a full-heap collection. This decision is made based on the available heap size, and the cost for nursery and full-heap collections. Our experimental results using SPECjvm98 showed substantial reductions in GC time (up to 29X) and significant overall performance improvements (more than 10%); similar speedups are obtained for application-specific as well as cross-application, generic GC cost functions. In addition, we also showed that GCH dramatically reduces maximum pause times. And finally, we showed that, for a specific benchmark, GCH improves overall performance compared to forced programmer-inserted garbage collections.

In future work, we plan to extend and evaluate GCH for other collectors than the ones considered here. We also plan to study dynamically inserted garbage collection hints in which profiling is done online during program execution.

Acknowledgments

Dries Buytaert is supported by a grant from the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). Kris Venstermans is supported by a BOF grant from Ghent University, Lieven Eeckhout is a Postdoctoral Fellow of the Fund for Scientific Research—Flanders (Belgium) (FWO—Vlaanderen). Ghent University is a member of the HiPEAC network. This work is an extended version of previous work [12]. We thank the reviewers for their insightful comments.

References

1. Blackburn, S.M., Cheng, P., McKinely, K.S.: Myths and realities: the performance impact of garbage collection. In: Proceedings of SIGMETRICS'04, ACM (2004)
2. Appel, A.W.: Simple generational garbage collection and fast allocation. Software practices and experience **19** (1989) 171–183
3. Blackburn, S.M., Cheng, P., McKinley., K.S.: Oil and water? High performance garbage collection in Java with JMTk. In: Proceedings of ICSE'04. (2004) 137–146
4. Alpern, B., Attanasio, C.R., Barton, J.J., Burke, M.G., Cheng, P., Choi, J.D., Cocchi, A., Fink, S.J., Grove, D., Hind, M., Hummel, S.F., Lieber, D., Litvinov, V., Mergen, M.F., Ngo, T., Russell, J.R., Sarkar, V., Serrano, M.J., Shepherd, J.C., Smith, S.E., Sreedhar, V.C., Srinivasan, H., Whaley, J.: The Jalapeño Virtual Machine. IBM Systems Journal **39** (2000) 211–238
5. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. The international journal of high performance computing applications **14** (2000) 189–204
6. Hertz, M., Blackburn, S.M., Moss, J.E.B., McKinley, K.S., Stefanovic, D.: Error free garbage collection traces: how to cheat and not get caught. In: Proceedings of SIGMETRICS'02, ACM (2002) 140–151
7. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification (second edition). Addison-Wesley (1999)
8. Boehm, H., Weiser, M.: Garbage collection in an uncooperative environment. Software practices and experience **18** (1988) 807–820
9. Brecht, T., Arjomandi, E., Li, C., Pham, H.: Controlling garbage collection and heap growth to reduce the execution time of Java applications. In: Proceedings of OOPSLA'01, ACM (2001) 353–366
10. Wilson, P.R., Moher, T.G.: Design of the opportunistic garbage collector. In: Proceedings of OOPSLA'89, ACM (1989) 23–35
11. Ding, C., Zhang, C., Shen, X., Ogihara, M.: Gated memory control for memory monitoring, leak detection and garbage collection. In: Proceedings of MSP'05, ACM (2005) 62–67
12. Buytaert, D., Venstermans, K., Eeckhout, L., De Bosschere, K.: Garbage collection hints. In: Proceedings of HiPEAC'05, LNCS 3793 (2005) 233–348
13. Detlefs, D., Flood, C., Heller, S., Printezis, T.: Garbage-first garbage collection. In: Proceedings of ISMM'04, ACM (2004) 37–48
14. J. M. Velasco, K. Olcoz, F.T.: Adaptive tuning of reserved space in an Appel collector. In: Proceedings of ECOOP'04, ACM (2004) 543–559
15. Stefanovic, D., Hertz, M., Blackburn, S.M., McKinley, K.S., Moss, J.E.B.: Older-first garbage collection in practice: evaluation in a Java virtual machine. In: Proceedings of MSP'02, ACM (2002) 25–36
16. Andreasson, E., Hoffmann, F., Lindholm, O.: Memory management through machine learning: to collect or not to collect? In: Proceedings of JVM'02, USENIX (2002)
17. Printezis, T.: Hot-swapping between a mark&sweep and a mark&compact garbage collector in a generational environment. In: Proceedings of JVM'01, USENIX (2001)
18. Soman, S., Krintz, C., Bacon, D.F.: Dynamic selection of application-specific garbage collectors. In: Proceedings of ISMM'04, ACM (2004) 49–60