Java Garbage Collector Performance Measurements

P. Libič and P. Tůma
Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic.

Abstract. Even if garbage collectors are widely used in modern software platforms, performance models tend to ignore their presence. This paper uses benchmark experiments to show that the impact of garbage collection might be significant and to suggest possible approaches leading to incorporation of garbage collectors into performance models.

Introduction

Advances in compiler and virtual machine technologies support spread of programming environments with automated memory management, such as Java. These environments rely on garbage collection to reclaim memory that is no longer used. While in the domain of garbage collection research, every proposed garbage collector improvement is accompanied with detailed analysis of the collector overhead [Detlefs et al., 2004; Zhang and Seltzer, 2001; Blackburn, 2006], in the domain of performance modeling, the collector overhead is usually treated implicitly as a constant background factor [Xu et al., 2006; Becker et al., 2009; Kuperberg et al., 2008].

A potential reason for this treatment is lack of available information on the overhead of the garbage collector implementation. Although some implementations are open source, it is rather difficult to get such information from the code. For example, the garbage collector implementation of OpenJDK counts for more than 50000 LOC. While this particular implementation is relatively well documented [Sun Microsystems, 2004], not all the information can be obtained from the documentation, and worse, the documentation contains misleading claims, such as that the JVM automatically adjusts total collector overhead to 1% of execution time [Sun Microsystems].

In order to provide a more precise collector overhead performance model, we have conducted many experiments to assess selected aspects of the garbage collector performance. Due to size constraints, the experiments are mostly reported online [Babka et al., 2009; Q-ImPrESS Consortium], and selected aspects are described in individual papers [Libič et al., 2009]. In this paper, we do not try to provide generalized claims, because the results of the experiments are bound to a particular virtual machine and collector. We do, however, compare two widely used collector implementations in the paper, and also attach notes on possible approaches towards modeling.

Collector Performance

Clearly, the collector overhead depends on the collector implementation. In this paper, two collectors are considered:

1. Generational collector that uses combination of semispace and copying algorithms in the young generation and mark and sweep in tenured generation. In the young generation, new objects are allocated into Eden space, at collection the still living objects are copied into To space along with surviving object from the From space and the role of From and To is swapped The allocation in young generation requires only pointer addition while free-lists are involved in tenured generation. Objects are promoted from young to tenured generation after survival of several young generation collections. This collector is used in the Sun JVM [Sun Microsystems, 2004].

2. Single generation mark and sweep collector using free lists for allocation is used in the virtual machine from IBM [Borman, 2002].

Workloads

All the executed experiments are based on three basic workloads or their combinations. The workloads exercise several aspects of the collector behavior – the target is to determine the factors that affect the performance of the collector and to develop a performance model according to these factors. Therefore the workloads are from the category of microbenchmarks that may or may not resemble the real-life applications. The other benchmarks widely used currently have different goals. The DaCapo

suite [Blackburn, 2006] and SPEC JVM 2008 [SPEC, 2008] consists of complex workloads to measure the the effect of newly developed features of virtual machines on overal performance. The three workloads are:

Object Lifetime The workload consists of a circular list of classes (called *components*) that contain several *payload objects* each. The workload has two variants. In the first, mostly young objects are allocated and then transformed into garbage. This is achieved by traversing *components* one by one and replacing one of the *payload objects* in each component. The other variant generates more old objects. Again, it traverses *components* one by one and replaces the *payload objects*. It always makes several replacements in the first component before it advances to the next one. In this way, all *payload objects* allocated in the first *component* are turned into garbage when they are still young and all other objects are older.

Heap Depth The workload consists of an array of root references and *payload objects*. The number of *payload objects* is the same as the length of the root array. In the *deep* variant, the *payload objects* are connected into a linked list and all root references from the array are pointing to the first element in the list. In the *shallow* variant each *payload object* is using a reference to point to itself and the root references point to all *payload objects*. The workload chooses one *payload object* and replaces it by newly allocated one. To make this possible in constant time, the objects are also referenced from array of weak references. The weak references are not considered in the garbage collection.

Heap Size The workload consists of *payload objects* that are connected by several references. The objects to connect are chosen at random, so the heap shape is random in general. The workload picks a random victim that is then re-allocated and the references are connected again pointing to random objects. The array of weak references is used to facilitate constant-time re-allocations. This workload is designed to avoid any particular structure on the heap that could affect the performance of the collector – only the size of the heap should be the factor that matters.

A more precise description of the workloads is available in Libič et al. [2009] and Babka et al. [2009].

Experiment Results

The experiments were conducted on a Dell PowerEdge 1955 machine with Dual Quad-Core Intel Xeon CPU E5345 2.33 GHz (Family 6Model 15 Stepping 11), 32 KB L1 caches, 4 MB L2 caches, 8 GB Hynix FBD DDR2-667 RAM, running Gentoo Linux kernel 2.6.27 x86 64, Sun Java SE Runtime Environment build 1.6.0-11-b03, Java HotSpot VM build 11.0-b16 for Sun's JVM and IBM J9 VM (build 2.4, J2RE 1.6.0) for the virtual machine from IBM.

The default maximum heap size limit was set to 64 MB per workload, which helps achieve a reasonable heap occupation without very large workloads.

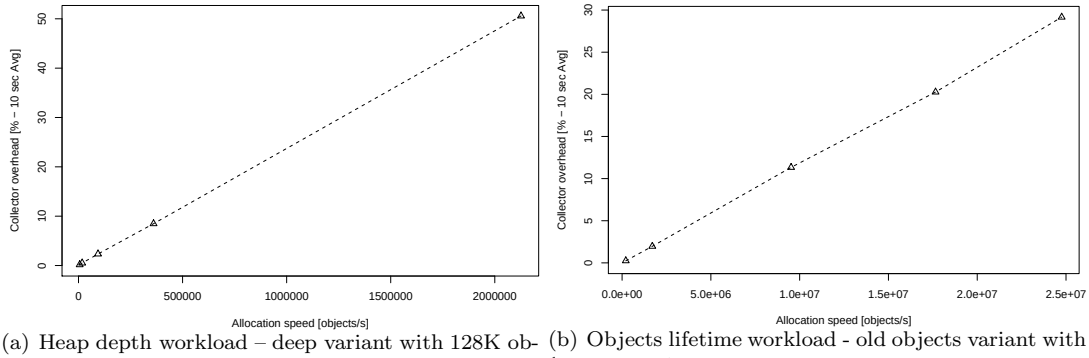
The collector overhead is measured using the diagnostic output of the garbage collector implementation. The combined overhead is reported alongside separate values for the young generation and the tenured generation in case the generational garbage collector is used. The overhead is taken as an average from a stable 5 minutes long execution period.

Dependency on allocation speed. The experiment to assess the dependency of collector overhead on allocation speed is using all the three workloads. The experimental infrastructure does not allow to set allocation speed directly. The experiment is therefore changing the amount of moderating work that avoids allocations, which effectively changes the allocation speed (and also the speed of garbage generation). The measured overhead is plotted against the measured allocation speed.

A selection of four experiment instances is displayed on Figure 1 for the IBM JVM and on Figure 2 for the Sun JVM. The workloads plotted in the individual sub-figures were the same, but measured on different virtual machines. The difference not only in the absolute values but also in the shape of the plot is very visible.

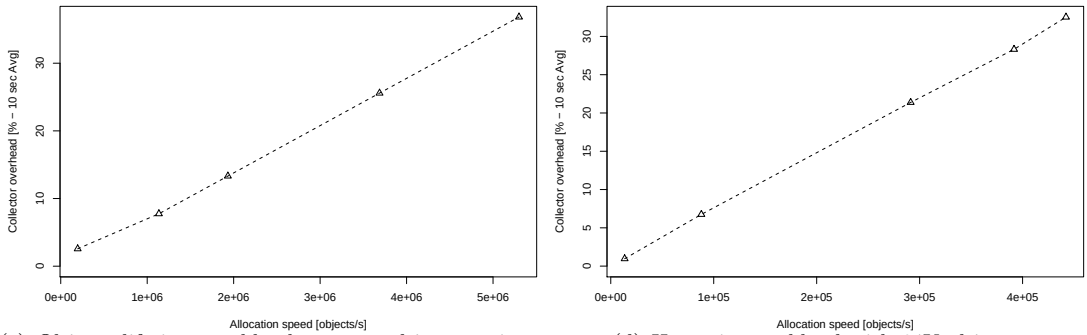
Workload combination – dependency on garbage. The experiment combines the Heap Depth and Heap Size workloads. Both workloads allocate their complete data set during the initialization. In the execution phase the workloads are switched after distinct number of allocations (and thus deallocations). The experiment varies the ratio of allocated objects in one cycle. As a consequence the allocation speed of both workloads changes with the change in the ratio.

The results for deep configuration of the Heap Depth workload are shown on Figure 3 for both virtual machines. On the Sun JVM (sub-figures d-f) the results show there is no difference in overhead



(a) Heap depth workload – deep variant with 128K objects

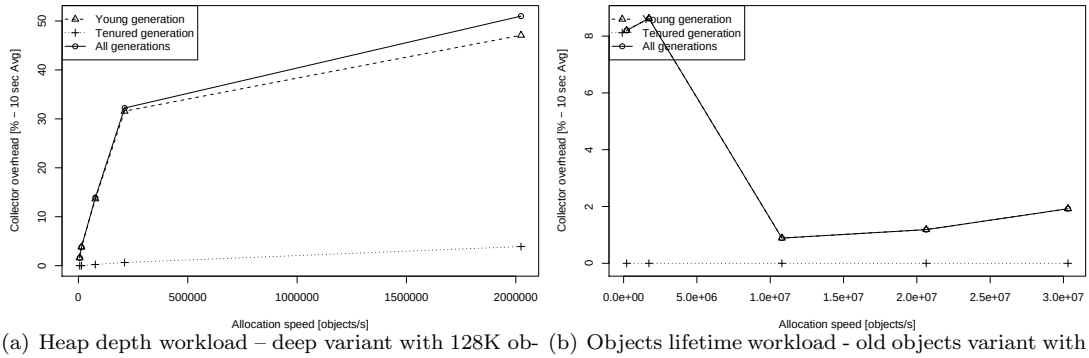
(b) Objects lifetime workload - old objects variant with 1 component



(c) Objects lifetime workload - young objects variant with 16K component

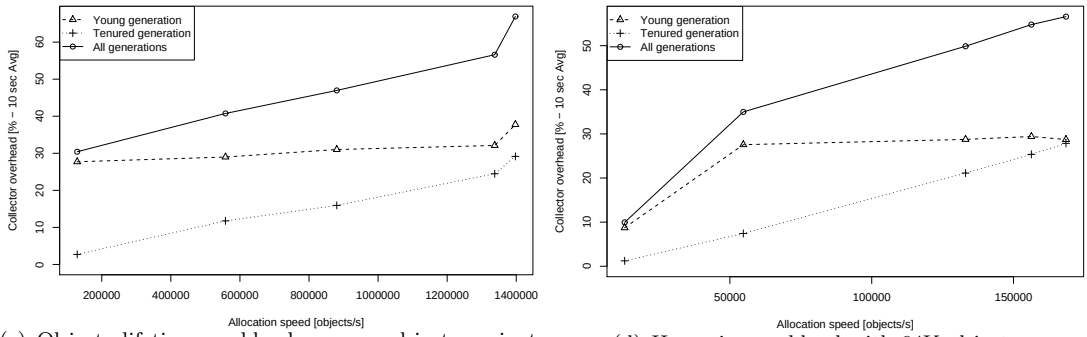
(d) Heap size workload with 64K objects

Figure 1. Dependency of the collector overhead on allocation speed with IBM's JVM.



(a) Heap depth workload – deep variant with 128K objects

(b) Objects lifetime workload - old objects variant with 1 component

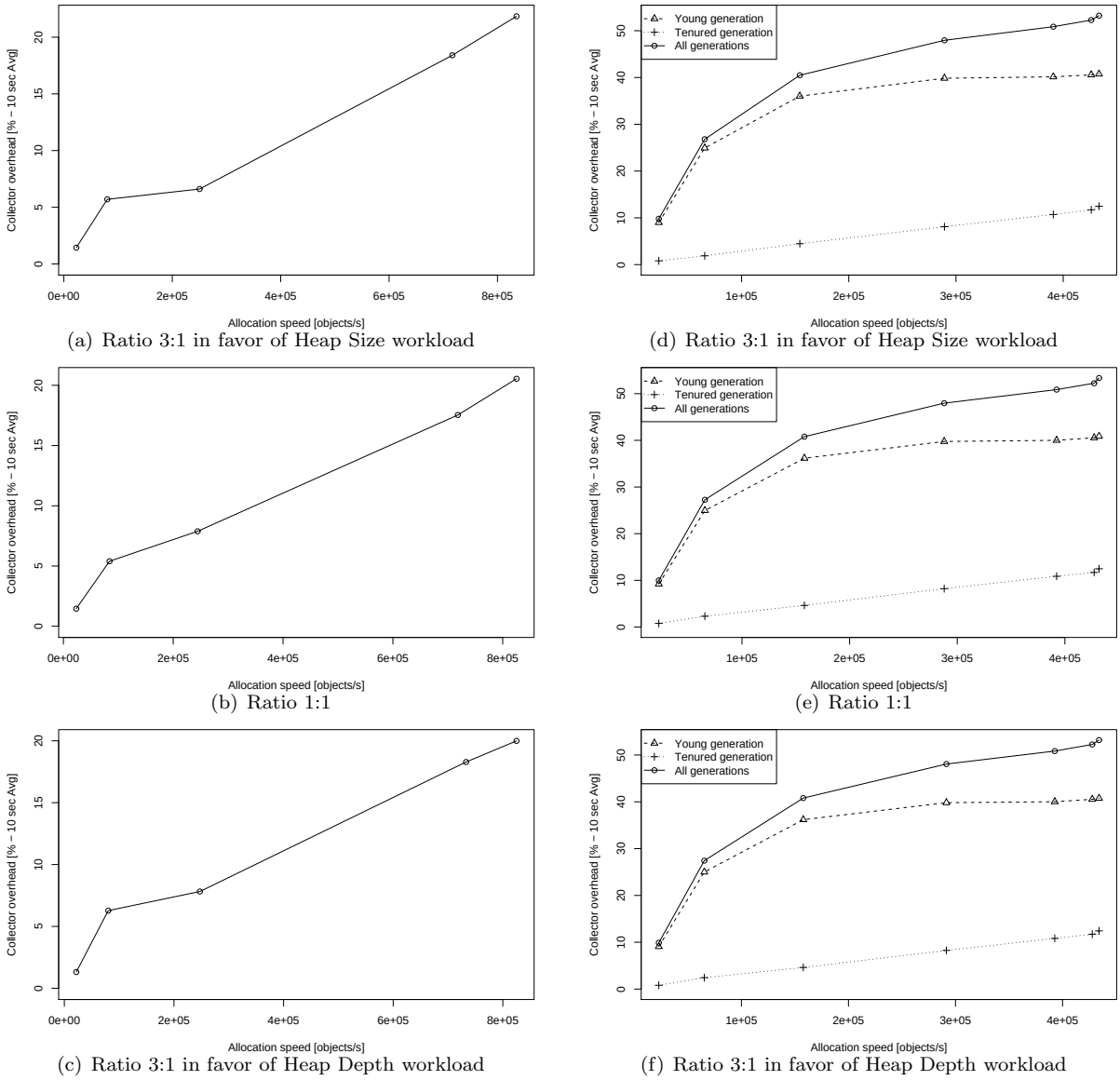


(c) Objects lifetime workload - young objects variant with 16K component

(d) Heap size workload with 64K objects

Figure 2. Dependency of the collector overhead on allocation speed with Sun's JVM.

with different allocation ratios. This proves that the collector overhead does not depend on the garbage on the heap. On the other hand the results from the IBM JVM cannot be interpreted this way because they show small decrease of overhead when the experiment allocates more object in the Heap Depth



(a) Ratio 3:1 in favor of Heap Size workload

(d) Ratio 3:1 in favor of Heap Size workload

(b) Ratio 1:1

(e) Ratio 1:1

(c) Ratio 3:1 in favor of Heap Depth workload

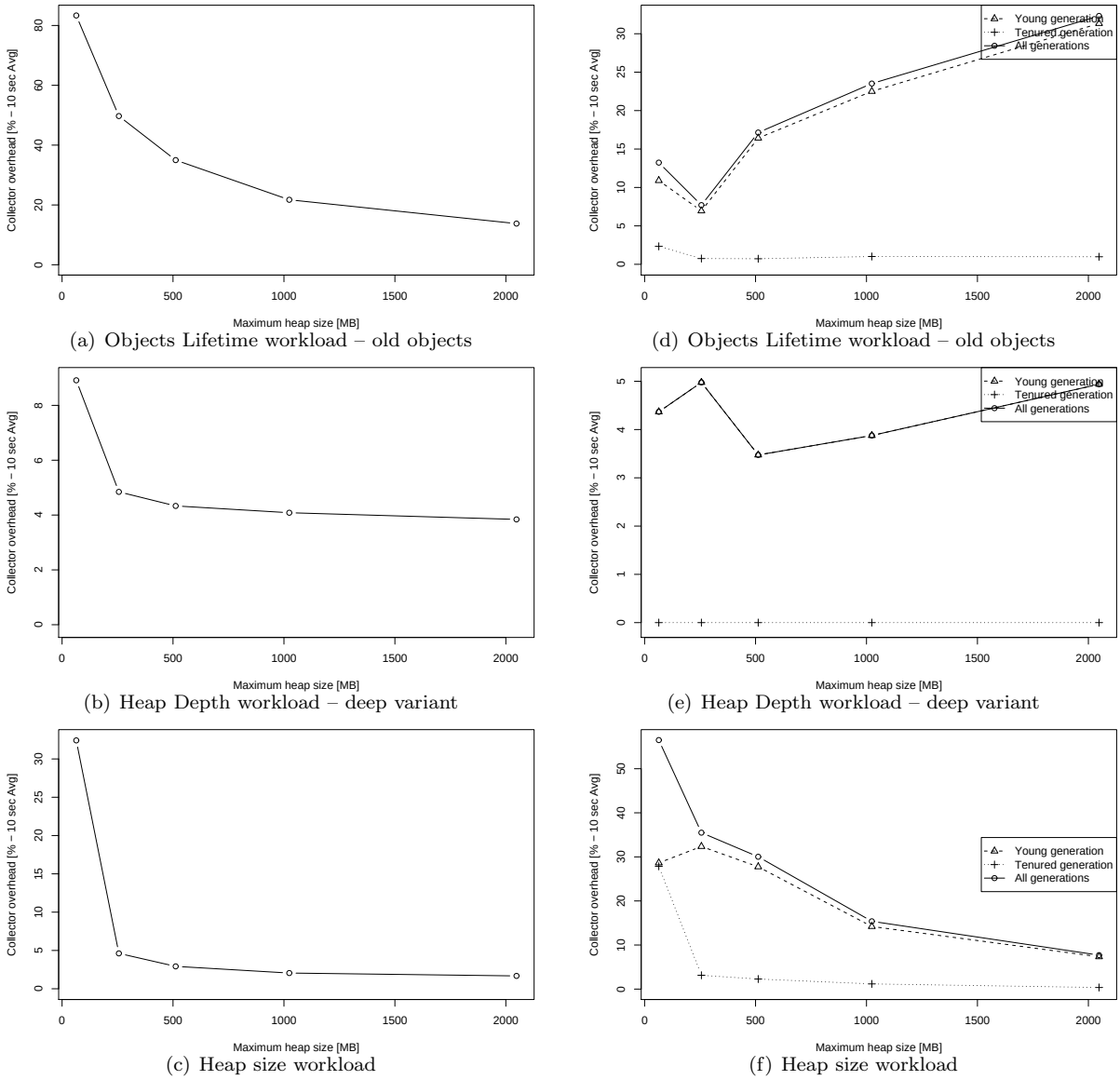
(f) Ratio 3:1 in favor of Heap Depth workload

Figure 3. Dependency of the collector overhead on allocation speed in combined workload and different ratios of allocations in single workloads. The IBM JVM on the left (a-c) and the Sun JVM on the right (d-f).

workload that has lower overhead in isolated scenario (Figures 1(a) and 1(d)). However, the decrease is not big enough to state that there is visible dependency on the garbage for that collector.

Dependency on maximum heap size. Previous work [Blackburn et al., 2004] states that collector overhead should decrease with increased heap size limit. This property seems to be inherent – suppose the same workload has twice as much memory available as before. The time between collections will be also twice as long as before because it will take longer to exhaust all the memory. On the other hand, the collection will take the same time, because the reachable objects are the same. This holds if the collector algorithm does not depend on the garbage in the heap.

The experiments to verify this property are using all three workloads and they are changing the memory available to the virtual machine. The results for selected experiment instances are shown on Figure 4 for both virtual machines. Even if the previous experiment shows the independence on garbage in the heap for the Sun JVM the results do not demonstrate the expected decrease in overhead. This might be caused by incorrectly applied heuristics that are sizing the heap generations. Further investigation is required to support this hypothesis. On the other hand the IBM collector overhead decreases even if the independence on garbage is not supported by the previous experiment.



(a) Objects Lifetime workload – old objects
(d) Objects Lifetime workload – old objects
(b) Heap Depth workload – deep variant
(e) Heap Depth workload – deep variant
(c) Heap size workload
(f) Heap size workload

Figure 4. Dependency of the collector overhead on maximum heap size. The IBM JVM on the left (a-c) and the Sun JVM on the right (d-f).

Stability of collector performance. The experiments on the Sun JVM were executed multiple times to ensure stability of the results. Because of time constrains the experiments on the IBM JVM were not systematically repeated more times, but the stability of results was verified. The results from the Sun JVM shown in this paper were always stable, however in previous experiments we encountered anomalous behavior in between several JVM executions. These anomalies are shown on Figure 5. Note that the total overhead is in the upper area of the plot always when the overhead of the tenured generation collector is non-zero. This might be another example of failure in generation sizing adaptive mechanisms.

Towards Modeling

From the experiment results shown the following lessons can be learned:

- It is probably not possible to create a single performance model for multiple collector implementations. The results from different virtual machines are so different that adjustment of the model to a concrete GC implementation cannot be just matter of parameter tweaking.
- For some implementations an exact model might not be feasible. There are too many factors that should be considered in the model, like adaptive policies that size the generations. The results

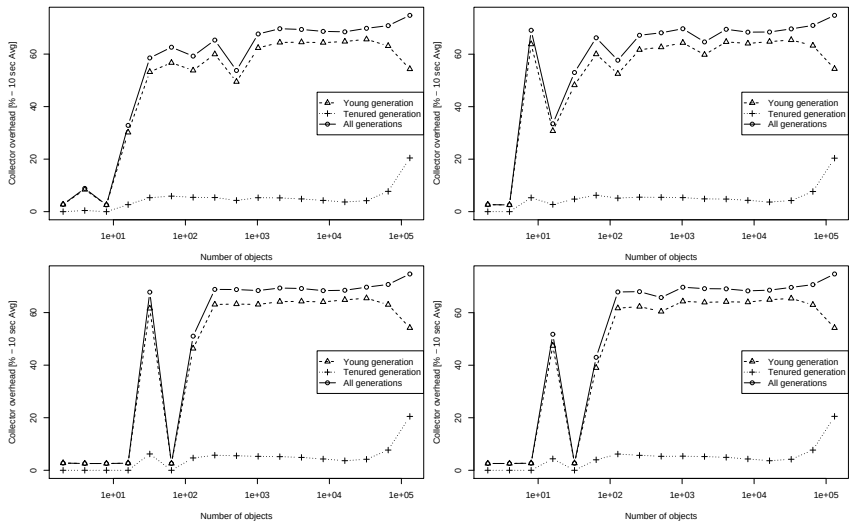


Figure 5. Different executions of Heap Depth with deep heap on the Sun JVM.

suggest this policy has greater influence on collector performance than any other factor, but there is very little information available on the heuristics used. The results obtained using the Sun JVM involves too many anomalies and the behavior is not even always stable between JVM executions. Differences as large as 80% of execution time were observed.

- Other implementation shows much more predictable behavior. With linear dependency on allocation speed the collector has constant overhead per reclaimed item. This constant depends on workload that generates the garbage, however. In combined workloads the constant cannot be determined as a simple average of constants computed from isolated workloads.

Conclusion

We have executed experiments on two different garbage collector implementations with the performance modeling perspective in mind. We have learned several lessons towards modeling and identified problems that prevent precise overhead modeling. However, the IBM JVM showed predictable results and might be a good platform for creating first models that could be later improved and maybe made to cover also the more complex collectors and virtual machines.

Acknowledgments. This paper was partially supported by the Q-ImPrESS research project by the European Union under the ICT priority of the 7th Research Framework Programme, and by the Czech Science Foundation grant 201/09/H057.

References

Babka, V., Bulej, L., Děcky, M., Libič, P., Marek, L., and Tůma, P., Resource usage modeling, http://dsrg.mff.cuni.cz/~libic/bench/report/, 2009.

Becker, S., Koziolek, H., and Reussner, R., The Palladio component model for model-driven performance prediction, *J. Syst. Softw.*, *82*, 3–22, 2009.

Blackburn, S. M., Cheng, P., and McKinley, K. S., Myths and realities: the performance impact of garbage collection, *SIGMETRICS Perform. Eval. Rev.*, *32*, 25–36, 2004.

Blackburn, S. M. e. a., The DaCapo benchmarks: Java benchmarking development and analysis, *SIGPLAN Not.*, *41*, 169–190, 2006.

Borman, S., Sensible sanitation – understanding the ibm java garbage collector, part 2: Garbage collection, http://www.ibm.com/developerworks/ibm/library/i-garbage2/, 2002.

Detlefs, D., Flood, C., Heller, S., and Printezis, T., Garbage-first garbage collection, in *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pp. 37–48, ACM, New York, NY, USA, 2004.

Kuperberg, M., Krogmann, K., and Reussner, R., Performance prediction for black-box components using reengineered parametric behaviour models, in *CBSE '08*, pp. 48–63, Springer-Verlag, Berlin, Heidelberg, 2008.

Libič, P., Tůma, P., and Bulej, L., Issues in performance modeling of applications with garbage collection, in *Proceedings of QUASSOS 2009*, 2009.

Q-ImPrESS Consortium, Q-impress project, http://www.q-impress.eu.

SPEC, SPECjvm2008, http://www.spec.org/jvm2008/, 2008.

Sun Microsystems, I., Garbage collector ergonomics, http://java.sun.com/j2se/1.5.0/docs/guide/vm/gc-ergonomics.html.

Sun Microsystems, I., Memory management in the Java HotSpot virtual machine, http://java.sun.com/-j2se/reference/whitepapers/memorymanagement_whitepaper.pdf, 2004.

Xu, J., Oufimtsev, A., Woodside, M., and Murphy, L., Performance modeling and prediction of enterprise JavaBeans with layered queuing network templates, *SIGSOFT Softw. Eng. Notes*, *31*, 5, 2006.

Zhang, X. and Seltzer, M., HBench: JGC - an application-specific benchmark suite for evaluating JVM garbage collector performance, in *COOTS'01*, pp. 4–4, USENIX Association, Berkeley, CA, USA, 2001.