Garbage Collection for Large Memory Java Applications

Andreas Krall and Philipp Tomsich

Institut für Computersprachen, Technische Universität Wien
Argentinierstraße 8, A–1040 Wien, Austria
{andi,phil}@complang.tuwien.ac.at

Abstract. The possible applications of Java range from small applets to large, data-intensive scientific applications allocating memory in the multi-gigabyte range. As a consequence copying garbage collectors can not fulfill the requirements, as large objects can not be copied efficiently. We analyze the allocation patterns and object lifespans for different Java applications and present garbage collection techniques for these. Various heuristics to reduce fragmentation are compared. We propose just–in–time generated customized marker functions as a promising optimization during the mark–phase.

1 Introduction

The programming language Java is used for a wide range of applications ranging from small applets running in a browser to large scientific programs taking hours of computation time and gigabytes of memory. It is difficult to design a garbage collector which performs well under different workloads. The garbage collector for CACAO – a 64 bit JavaVM for Alpha and MIPS processors [KG97,Kra98] – has been designed for large objects and large memory spaces but also performs well for small objects. In this study we evaluate the behavior of different garbage collection schemes under different workloads to find common patterns which can be used to design efficient garbage collection heuristics.

There exist hundreds of different garbage collection algorithms [JL96,Wil94] which can be largely divided into copying and non–copying collectors. The non–copying mark–and–sweep collectors are very efficient but it is widely believed that they suffer of memory fragmentation problems. Recent studies [JW98] showed that the fragmentation problem is small. For large object spaces copying collectors are unusable because of the copying overhead. It is impractical to copy objects which reach Gbyte sizes. Feasible solutions are treadmill and mark–and–sweep collectors.

The remainder of this paper is structured as follows: Section 2 lists related work. In section 3 we briefly present an overview of the garbage collector and present the result from the lifespan analysis for objects. We also discuss different heuristics to improve garbage collection. Customized marking using just–in–time generated marker methods is introduced in section 4. Section 5 presents the experimental results from our tests. We draw our conclusions in section 6.

2 Related Work

Precise garbage collection for Java virtual machines has been described by Ageson et al. [ADM98]. In contrast to conservative (or partially conservative) collectors precise collectors have exact information about objects on stack and heap. The computation of the stack maps in a JavaVM is complicated by the fact that local variables can contain any type across the call of a local subroutine (`jsr`) which is used to implement exception handler routines. During compilation the Java byte-code has to be rewritten to rename variables which are live across a call of an exception handler with different types. Using the precise collector the heap size can be reduced by 4% on average in comparison with a partial conservative collector.

A recent study by Johnstone and Wilson [JW98] evaluated the fragmentation of conventional dynamic storage allocators. The study analyzed 8 big C and C++ programs using 16 different implementations of malloc/free. Most of the live objects are very small (less than 64 bytes). Usually only very few large objects exist. Large objects have a long life time. The best and also efficient algorithms showed an average fragmentation below 3%. These fragmentation numbers are not directly comparable to garbage collection fragmentation behavior since life times and allocation/deallocation patterns are different.

Hicks et al. [HHMN98] studied the garbage collection times for large object spaces. Using a separate non–copy collected space for large objects results in significant performance improvements for copying garbage collectors. This study evaluated varied size thresholds as well as whether or not large objects may contain pointers. A treadmill collector was compared with a mark–and–sweep collector. As benchmarks different programs written in Java and SML are used. For Java programs optimal threshold values are smaller than for SML. There is no measurable difference in performance between the treadmill and the mark–and–sweep collector.

Colnet et al. [CCZ98] describe compiler support to customize the mark–and–sweep algorithm in the SmallEiffel compiler. The SmallEiffel garbage collector is a classical partially conservative mark–and–sweep collector. Type inference is used to compute the necessary information to segregate objects by type and statically customize most of the GC code. In this study 22 different implementations of Othello (both leaky (which rely on a garbage collector) and non leaky versions) have been evaluated. The results show both a reduction in run time and memory footprint compared with either no GC at all or the Boehm–Weiser collector [BW88].

3 The garbage collector

We implemented a conservative garbage collector using a mark–and–sweep algorithm. During the mark phase the contents of the stack are considered references to root objects. Starting with these root objects, objects on the heap are marked recursively. Every heapblock contained in a marked object is considered a potential reference and has to be examined: if it points to the start of an unmarked object, that object is marked in turn.

In order to store an indication of where objects start and whether objects are marked, we use bitmaps where each bit represents one heapblock. These bitmaps encode whether an object starts at a heapblock, whether it is marked and whether it may contains references to other objects.

The sweep phase is currently implemented using the start and free bitmaps. Continuous free space resulting from neighboring objects is automatically detected and freed as one block of memory. We are currently working on an incremental method of collecting and releasing this garbage, which may reduce the cost of sweeping by up to an order of magnitude as our preliminary tests indicate.

Allocation is performed from a list of free memory blocks. First, we attempt to satisfy the allocation request with an exact block; if that fails, a part of a larger block is split off and used. If no exact or large block can be found, we grow the heap up to a maximum limit. As soon as that limit is reached, a garbage collection is performed.

3.1 Object lifespans

To collect information on object lifespans, we performed a full garbage collection during each allocation request. The results confirmed our hypothesis, that objects remain valid for either a very short period of time or for a very long one. Our experiments show that only about 20% of all objects live longer than 512 bytes (which equals 10 allocations in most programs) (see fig. 1). This implies that the best method to reduce overall fragmentation, which is very significant for the total memory required for the execution of a program, is to collect early and often at the cost of a small increase in garbage collection time.

3.2 Generational garbage collection

Recently, generational garbage collection [HM92] gained in popularity: it exploits the typical allocation patterns found in modern languages to optimize garbage collection. Assuming, that memory allocated on the heap either becomes garbage very quickly or remains valid for most of the program's runtime, garbage collection should differentiate between *young* and *mature* objects. While *young* objects have a high probability to die with the next collection, *mature* objects will likely survive it. The generational approach separates the allocated objects can into different generations according to the number of collections they survived. Whenever a garbage collection is triggered, they youngest generations are observed first, which reduces the number of memory accesses required. Surviving objects are promoted to older generations.

Using a generational garbage collector may appear a good solution to the small lifespans observed, but it leads to severe problems: the write–barrier necessary to record cross–generational references imposes an overhead on all as-

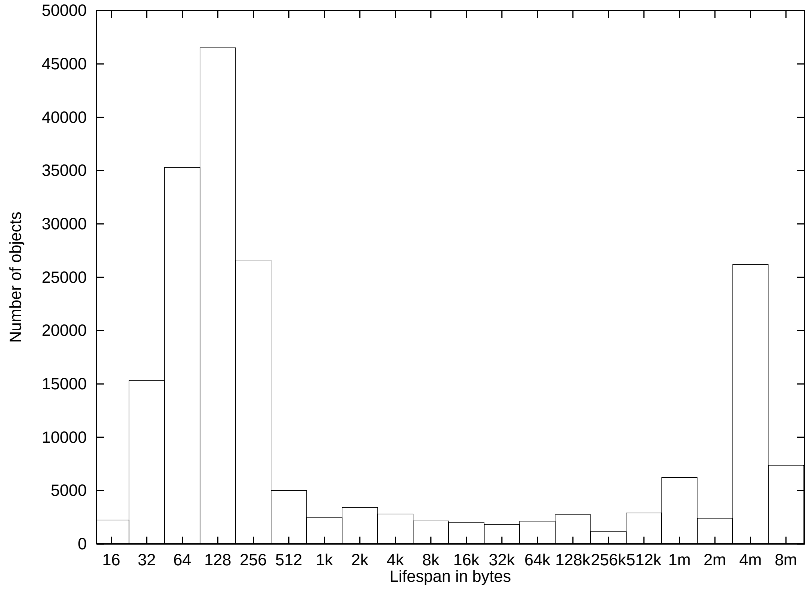

Fig. 1. Lifespan graph for a compilation of JavaLex with javac

signments of objects. In addition, generational garbage collection uses a larger amount of heap space, as garbage in older collections is released only infrequently. This memory, which may remain uncollected for some time even after becoming garbage, can amount to a considerable memory area in applications with large memory requirements.

3.3 Heuristical threshold values

Fragmentation can be reduced by scheduling collections on the extent of heap (i.e., the highest address used within the heap at any given time). Our garbage collector collects as soon as a heuristically chosen threshold for the extent of the heap is exceeded. For the implementation we considered 3 heuristics:

1. **Adding a constant value.**
   This naive technique ignores the smaller size of the heap after the collection and grows the threshold very quickly.
2. **Adding a multiple of the current extent.**
   Adding a multiple of the heap extent resulting from the garbage collection ignores the fact, that non–copying garbage collectors leave a fragmented heap (i.e., the extent of the heap is considerably larger than the heap size necessary to hold all the objects on the heap). This leads to a quickly expanding heap for non–copying garbage collectors, such as the mark–and–sweep collector used in CACAO.
3. **Adding a fraction of the unused space.**
   This is a very effective heuristic, which works well with the fragmented heaps

resulting from non–copying garbage collection. It reduces the overall fragmentation, such that programs can be run with less memory than needed for the other two strategies.

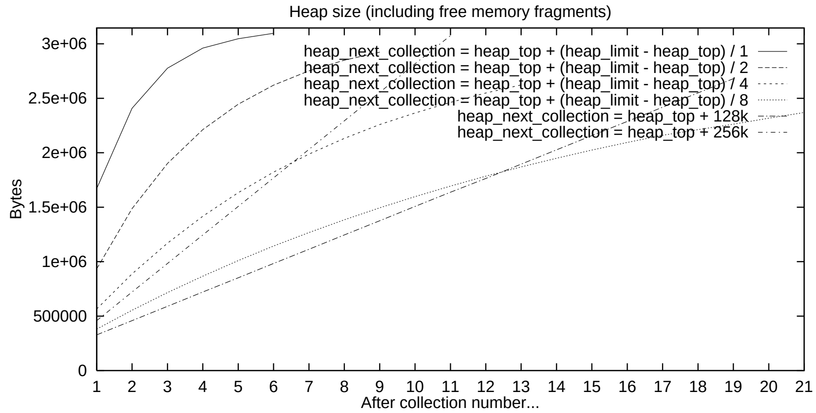

Fig. 2. Heap size for different threshold heuristics (javac)

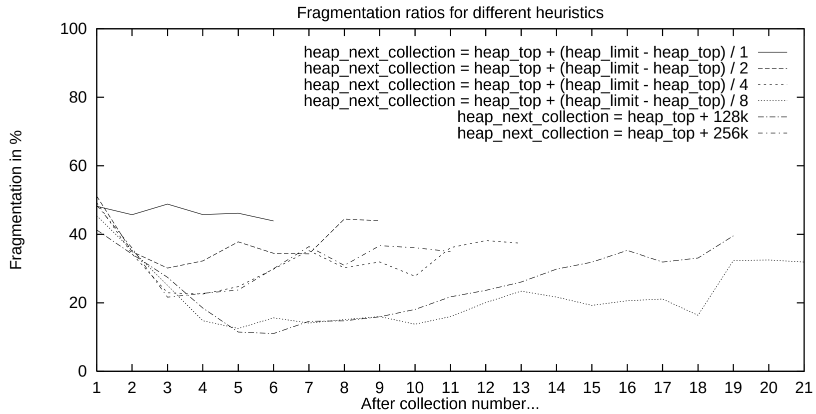

Fig. 3. Fragmentation for different threshold heuristics (javac)

Figures 2 and 3 show that the method of adding a fraction of the unused space is preferable, since it maintains a far more compact heap than the other variants. Additionally, it offers superior better performance during garbage collection by collecting "early and often". This fact is due to the observed pattern of small objects with small lifespans; frequent collections keep the heap compact and reduce the cost of garbage collecting further by improving the caching behavior of the mark–and-sweep algorithm.

4 Customized marking

The data gathered during our experiments indicates that about 60% of all heapblocks examined (for numerical applications this percentage raises to more than 95%) during a marker pass are built–in types other than references (i.e., integers, doubles). Nonetheless, these heapblocks are dereferenced and the resulting value checked against the bounds of the heap and an allocation–bitmap to verify whether an object actually begins at this address. We may conclude from these findings, that a very large potential improvement in the performance of garbage collection for large memory application would result from excluding those heapblocks within an object, which will never contain pointers, from the marker pass. Two methods to store and evaluate this information exist:

1. **Bitmap based methods.** The type–information for the physical components of classes can be stored as bitmaps within the class-info structures. These bitmaps encode boolean values indicating whether the heapblock at a certain offset within the object heapblock may contain a reference or not. During the mark–phase, these bitmaps are interpreted to customize the marking on a per–class basis.
   The disadvantages of this approach are the overheads introduced by the interpretation of the bitmaps and the additional memory accesses necessary to retrieve the bitmaps. In addition, large objects containing only few pointers can neither be represented efficiently nor be marked without examining the entire bitmap.
2. **Just–in–time generated marker methods.** The bitmap–based methods described above may be modified by translating the information encoded within these bitmaps into executable code. For a portable implementation byte code versions of the mark methods are generated during class loading which are translated on demand into native code during garbage collection. This provides just–in–time generated marker methods customized for every class. During garbage collection these methods are called for every live object.
   This solution offers almost optimal execution times for the marker, because only those heapblocks are examined which may contain references (i.e., all heapblocks, for which the compiler can determine that they will never contain a reference, are excluded). While additional code needs to be generated and stored for every class, the storage overhead involved is negligible in size compared to the other information in class-info. Generally the resulting marker method will require less than 50% of the memory accesses needed for the naive approach.

4.1 Conserving stack space

Recursive marking algorithms require large amounts of stack space, particularly for deeply nested structures and long lists. This may cause stack overflows.

Pointer reversal techniques provide an alternative to recursive marking, but impose far more memory accesses.

A method to save both stack space, as well as improve performance by reducing the number of necessary recursive calls is to optimize for tail recursion. This allows the last recursive call to reuse the current stack and return address. This is especially useful in the context of lists, where a `next`–pointer can be detected and placed at the tail of the structure by the compiler. This optimization is particularly beneficial in the context of large memory applications, where large lists are processed.

5 Experimental results

To evaluate fragmentation and different heuristics we used following test applications:

javac	the Java compiler
jess	Java Expert Shell System based on CLIPS
db	memory resident data base system
raytrace	a raytracer rendering a dinosaur
scimark	a mix of numeric applications
linpack	the famous linpack benchmark

Besides the small `linpack` benchmark, all applications allocate between 100 and 500 Mbyte of objects. Our results show that most applications allocate only small objects with average sizes between 30 and 60 bytes (see table 1). Only the scientific/numerical applications `scimark` and `linpack` used big arrays and have an average object size of 810 respectively 1353 bytes. The distribution of the objects shows the peak at the small sizes with most objects smaller than 128 bytes.

Table 2 gives the number of references which have to be checked if they are valid object pointers. For scientific programs nearly 100% of the pointers are either false pointers (i.e., potential pointers that fall outside of the heap) or null pointers. This result shows the potential performance improvement of the JIT-marker. The high percentage of null pointers demonstrates the importance of checking the null pointer at the call site of the mark method.

The data in table 1 and the fragmentation data in the plots and in table 3 show fragmentation between 10 and 30% of the heap. Since the fragments are large enough for allocation of new objects the fragmentation is not a real problem for Java garbage collectors.

6 Conclusion

Java applications with large objects require special techniques for garbage collection. Only non–copying collectors can provide acceptable performance both in

benchmark	javac	jess	db	raytrace	scimark	linpack
heap size	62667832	468663656	124944496	187717936	43611752	334328
objects	1340767	7918647	3202929	6338943	53817	247
object size	46.7	59.1	39	29.6	810	1353
16	0	48	0	0	0	0
32	458205	1425465	3061233	4693884	563	24
64	584415	4474601	98875	1419488	535	10
128	224339	997584	42442	225550	268	4
256	13493	1019885	17	15	16	4
512	783	395	6	4	515	1
1024	2279	214	2	0	51914	1
2048	225	290	1	0	0	203
4096	379	113	1	1	0	0
8192	180	47	1	0	0	0
16384	33	4	1	0	0	0
32768	2	1	1	0	0	0
65536	1	0	1	0	2	0
131072	0	0	346	0	3	0
262144	0	0	1	1	1	0

Table 1. Object counts

benchmark	javac	jess	db	raytrace	scimark	linpack
all references	12637996	6448116	41470054	41133853	478324	47842
false pointers	5630274	2330767	29954232	26416277	445394	912
null pointers	2225825	1089709	3087413	2968714	31372	46620
false percentage	44.5	36.1	72.2	64.2	93.1	1.9
null percentage	17.6	16.8	7.4	7.2	6.5	97.4

Table 2. Reference checking

benchmark	javac	jess	db	raytrace	scimark	linpack
number of fragments	46148	8449	212	599	5	0
average size of fragments	147.3	1692	19394	5145	268041	0

Table 3. Fragmentation

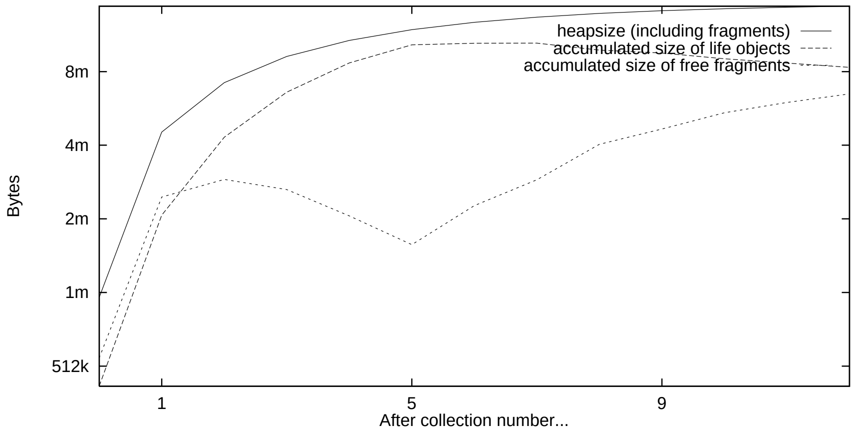

Fig. 4. Fragmentation of javac

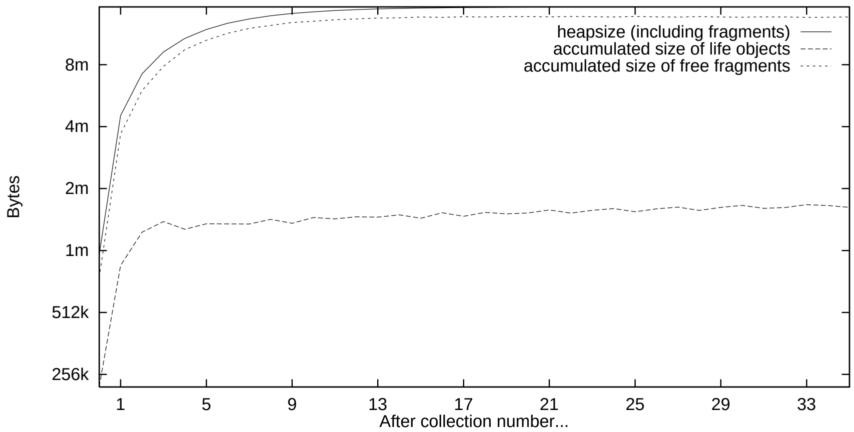

Fig. 5. Fragmentation of jess

applications with small objects and those with large objects. Our analysis of object lifespans demonstrated that almost all objects have very short lifespans. The fragmentation analysis showed that fragmentation is not a significant problem for non–copying garbage collectors for Java. Just–in–time generated customized marker methods reduce the runtime overhead of marking by more than 60%.

References

[ADM98] Ole Agesen, David Detlefs, and J. Eliot B. Moss. Garbage collection and local variable type-precision and liveness in Java virtual machines. In *Conference on Programming Language Design and Implementation*, volume 33(6) of *SIGPLAN*, pages 269–279, Montreal, 1998. ACM.

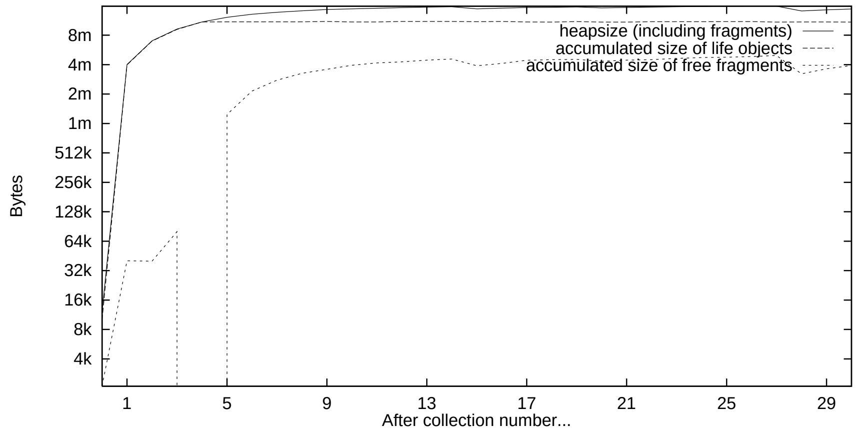

Fig. 6. Fragmentation of db

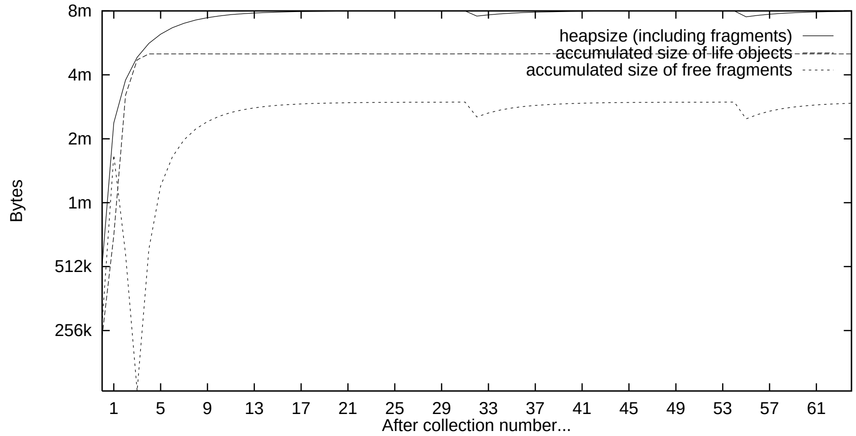

Fig. 7. Fragmentation of raytrace

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[CCZ98] Dominique Colnet, Philippe Coucaud, and Olivier Zendra. Compiler support to customize the mark and sweep algorithm. In *1998 International Symposium on Memory Management*, pages 154–165, Vancouver, 1998. ACM.

[HHMN98] Michael Hicks, Luke Hornof, Jonathan T. Moore, and Scott Nettles. A study of large object spaces. In *1998 International Symposium on Memory Management*, pages 138–145, Vancouver, 1998. ACM.

[HM92] Richard L. Hudson and J. Eliot B. Moss. Incremental collection of mature objects. In *Proceedings of the International Workshop on Memory Management*, pages 388–403, September 1992.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection.* John Wiley, 1996.

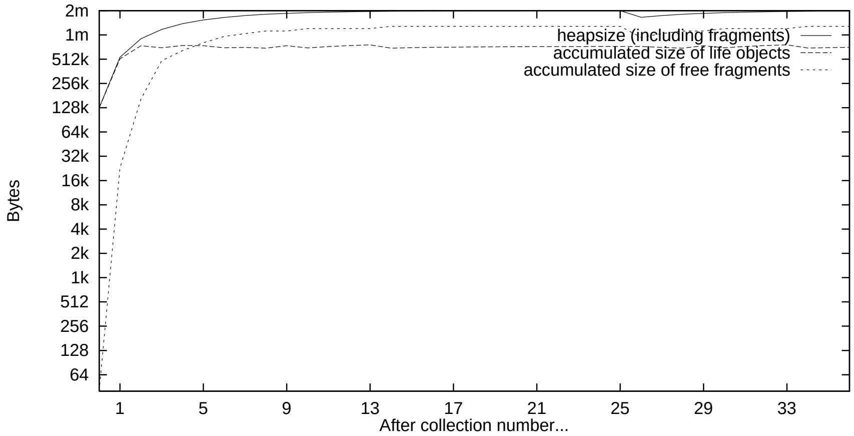

Fig. 8. Fragmentation of scimark

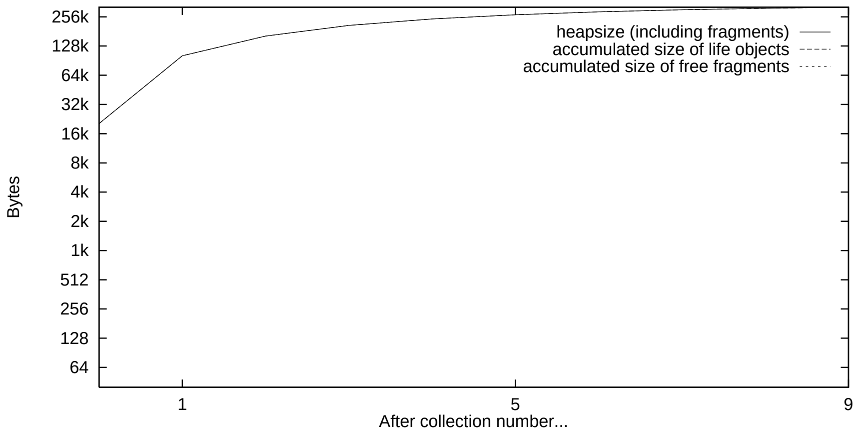

Fig. 9. Fragmentation of linpack

[JW98] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In *1998 International Symposium on Memory Management*, Vancouver, 1998. ACM.

[KG97] Andreas Krall and Reinhard Grafl. CACAO – a 64 bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997.

[Kra98] Andreas Krall. Efficient JavaVM just-in-time compilation. In Jean-Luc Gaudiot, editor, *International Conference on Parallel Architectures and Compilation Techniques*, pages 205–212, Paris, October 1998. IFIP,ACM,IEEE, North-Holland.

[Wil94] Paul R. Wilson. Uniprocessor garbage collection techniques. In *ACM Computing Surveys*, page to apear. ACM, 1994.