RESULTS OF PROFILING AND ANALYSIS OF BEHAVIOUR OF MEMORY OBJECTS IN JAVA

DANKO BASCH and JURICA BOROZAN

University of Zagreb, Faculty of Electrical Engineering and Computing,
APR, Unska 3, Zagreb, Croatia
danko.basch@fer.hr, jurica.borozan@fer.hr

Abstract: The performance of a memory system depends to great extent on the algorithms used for memory allocation and garbage collection (GC). GC implementers need accurate data about memory objects' behaviour in order to produce high quality garbage collectors optimized for their particular problems. Therefore, different GC-related data should be collected and analysed afterwards. We propose a set of parameters that should be traced in benchmark programs and organization of an analysis environment. Then we present several examples of results obtainable by the proposed analysis environment. These results can be useful in improving the existing or in the creation of new GC algorithms. The profiled data are obtained by running benchmarks written in the Java programming language that we use as a representative of the languages that belong to the object oriented programming paradigm.

Keywords: heap, profiling, behaviour of memory objects, garbage collection (GC), Java.

1. INTRODUCTION

Many modern programming languages (especially object oriented (OO) languages) use automatic memory management, i.e. garbage collection (GC) [Jones and Lins, 1996]. GC makes programming easier and less error-prone, which is especially important in today's software production. On the other hand, GC can significantly affect the performance of programs (execution speed and memory usage). Therefore, GC algorithms have been under constant development for decades. The performance of interpreted languages is less affected, but the conclusions drawn from their investigation are also applicable to compiled languages. In our investigation of the behaviour of memory objects, we have used Java as a representative of the OO languages.

GC algorithms are often based on beliefs about common behaviour of memory objects. However, recent measurements show that some of the beliefs are not always true. E.g., generational algorithms are better suited for functional languages and should be adjusted for OO languages [Stefanovic et al, 1998; Stefanovic et al, 1999].

Under the term "memory object", we understand "the unit of memory allocated on the heap". This should be distinguished from the term "object" with the meaning "instance of a class" which is typical in the OO paradigm. However, there is a correspondence between these terms, since every "instance of a class" is "allocated as a unit on the heap". When the context is not ambiguous, we will use the shorter form "object" instead of "memory object".

Memory objects' behaviour patterns are very different because they depend on several factors:

- the programming paradigm (OO, functional, imperative etc.),
- the programming language and its execution style (compiled or interpreted),
- language implementation (implementation of interpreter/compiler/optimizations),
- the application (e.g. interactive, non-interactive, long or short-run, etc.),
- the programmer's style (algorithms used, organization of data in memory etc.).

Obviously, the optimal GC does not exist. Because of that, creation, optimization, or adaptation of GC algorithms are difficult tasks without the exact knowledge about the usual behaviour of objects. Therefore, our aim is to produce data about typical behaviour of memory objects for the Java language. These data could later be used in different ways, e.g. for: investigation of GC algorithms, GC performance prediction and comparison, application performance prediction, trace driven simulation of memory objects' behaviour, static analysis of memory objects' behaviour, synthetic benchmark creation etc.

The heap (memory) profiling is typically used for two different purposes. The first is profiling of particular applications in order to locate bottlenecks, memory leaks, memory usage, functions that allocate most of the memory etc. The main goal is to optimize or debug the particular application, and the profiling is usually built in the compiler, interpreter or IDE (integrated development environment). The second purpose of profiling is used in GC and memory allocation researches. This paper deals only with the second usage of profiling. Because the

goals and collected data are different, the first approach is not suitable for the purposes of the second approach.

We have implemented an analysis environment that can produce and analyse data about behaviour of memory objects. Its main advantages are:

- the collected events/parameters are general enough, so its analysis is not constrained to a predefined set of results, but is rather flexible and adaptable,
- the data is stored in a database, which is not common, but this enables very simple and effective analyses, even when analysing the results in a way that was not intended before,
- the profiled data are analysed directly, without the need for a trace-driven simulation, which makes the analysis very fast.

The paper is organized as follows. Previous research in the area is given in section 2. Section 3 explains some basic facts about different aspects of profiling of memory objects. The main decisions about the analysis environment and its organization are briefly described. Implementation of the analysis environment is given in section 4. The main part of the paper is section 5 where several results obtained by the analysis environment are presented. The section 6 includes the conclusion and gives some directions for future work.

2. RELATED WORK

Different aspects of memory allocation and GC are profiled. Earlier work has focused primarily on memory allocation and the behaviour of programs related to it. Profiling related to GC encompasses events related to memory allocation and therefore can be used for that purpose (although this is not the case in practice).

Older studies mainly described profiling of functional or imperative languages (e.g. ML [Stefanovic and Moss, 1994], C [Zorn and Grunwald, 1992], and LISP [Zorn, 1989]).

The profiling of general performances is used for comparison of different GC algorithms and their evaluation in many studies (e.g. [Zorn, 1989; Stefanovic et al, 1999; van Groningen, 1995] etc.), and also in evaluation of memory allocation algorithms (see [Wilson et al, 1995] for extensive list of references). A GC performance analysis tool Oscar uses snapshots of a heap to replay the GC. Oscar is used for comparison of GC algorithms across different languages and platforms. Its main drawback is its inability to measure the GC related costs during program execution (e.g. reference count updates, write-barriers etc.) [Hicks et al, 1997].

Since collection of trace data can be a time consuming task, there is a proposal for a standard trace format and a collection of traces that can be shared among researchers [Chilimbi et al, 2000]. However, the proposed format is intended for memory allocation events rather than GC related events.

Another kind of profiling is dynamic profiling which is used in run-time tuning of GC, e.g. for the pretenuring of objects in generational collectors [Blackburn et al, 2001; Harris, 2000]. Such profiling has to be simple enough since it is performed during the program execution. The profiled parameters are restricted to those used for the tuning.

Extensive profiling for GC purposes is proposed in [Harrison and Waldron, 1999], but to our knowledge, it was never implemented.

Connectivity of heap objects in Java is profiled in [Hirzel et al, 2002] and the results are later used for the design of connectivity based GC [Hirzel et al, 2003].

The approach undertaken in [Dieckmann and Hoelzle, 1999] is the most similar to our study of memory behaviour of Java benchmarks. It includes several metrics: age distribution, reference density in objects, heap composition (most results are given separately for simple objects and arrays). However, in [Dieckmann and Hoelzle, 1999] the traced data include allocation and assignment events without deallocation events. The second phase uses trace-driven simulation in order to "replicate" the behaviour of benchmark and to collect statistical data (which is very time consuming).

To determine precisely the death-time of objects for profiling, the GC should be triggered at every possible GC point, which is extremely slow. A faster but imprecise solution is to use granulated trace, which means to trigger the GC after the fixed number of GC points. The problem of death-time determination is solved by the Merlin algorithm, which is precise and efficient [Hertz et al 2002].

3. PROFILING OF MEMORY OBJECTS

One of primary tasks in investigation of memory-related performance would be to collect data about memory objects and their behaviour. The collected data should be used as a basic source of knowledge for design, improvement, optimization, and tuning of memory management and GC algorithms, which would improve the overall system performance. The data is collected by profiling memory objects during an execution of benchmark programs (i.e. by tracing various events and parameters related to memory

objects). The collected data is then analysed in order to find possible patterns of behaviour of memory objects.

Research in this field requires development of analysis environment used to acquire and analyse profiled data. Many indicators of behaviour of memory objects can be used in the analysis. Here is a list of possible indicators:

- size and number of objects (represents consumption of memory),
- object's age (time of birth and death, time spent in memory),
- time between allocations,
- number of object's children and memory graph depth,
- object locality (describes alignment of object and its children in memory),
- data types stored in the objects (used in approximation of size, content, access),
- access and order of access to objects,
- number of changes in object (which can be used to improve locality),
- pointer assignments (prediction of changes of the memory graph),
- object's scope (static, local, or heap objects).

Obviously, straightforward data collection can be a very demanding task, but it is not necessary to collect the indicators directly. They should be determinable from the collected data. It can be assumed that the best approach is to divide the profiling process in two subtasks:

- collection of data (i.e. running the benchmark, and storing the collected data),
- analysis of data (i.e. using stored data in search for patterns of object behaviour).

3.1 Benchmark Selection

Several problems are related with the selection of benchmarks. Inappropriately selected benchmarks can provide a too narrow "image" of memory objects' behaviour. Different types of applications (e.g. server, interactive, database, numerical etc.) handle memory in different ways and memory requirements and objects behaviour are not the same. In investigations similar to ours, several sets of benchmarks are usually used: SPECjvm98, Java-Olden, and Colorado benchmarks.

3.2 Trace Data Issues

The collection of trace data shall be integrated with the benchmark execution and memory management procedures. The usual implementation is instrumentation of all operations that could change the memory layout of objects. Instrumentation of operations means that they will be augmented with the profiling procedures. Of course, this can introduce a significant overhead in execution time of benchmark. With careful selection of traced parameters and optimized implementation, the overhead should be reduced as much as possible.

3.3 Trace Format Issues

The profiled data contain traces of all selected events and parameters. They should be stored in the appropriate format for the later use. The trace file format should be designed with the following goals in mind [Chilimbi et al, 2000]:

- expressiveness,
- compactness,
- flexibility.

The trace file can be stored in binary format or text (ASCII) format [Chilimbi et al, 2000].

The advantages of binary format are its compactness (which also speeds up disc-write operations), and low overhead of transformation of collected data into final format. Its main disadvantage is the need for an appropriate software for the processing of the collected data.

ASCII format makes trace data readable, easy to manipulate, and platform independent. Another advantage is the possibility to use one of the mark-up languages (e.g. XML) for outlining the data structure. This would also make easier the processing of results since parsing libraries for XML already exist [Borozan and Basch, 2004a].

3.4 The measure of time

Due to a number of different computer platforms with various operating systems, hardware, and speed, measuring time in real clock or in number of executed instructions would make analysis results platform-dependent and hard to compare. More general and independent notation of time is used - time is expressed in bytes allocated so far, which represents the size of all memory objects created up to the current moment. This notation is usual in the GC community because it has a number of advantages. It is platform- and performance-independent, and easy to implement. It is also related to the frequency of garbage collections.

On the other hand, expressing time in bytes also has its drawbacks. It does not express the real-time frequency of object allocation (which is important in interactive programs), nor the real-time frequency of GC. In addition, the finer tracking of events between two allocations is disabled, because the time is not advanced without an allocation.

3.5 Traced Events and Parameters

After we have presented different profiling issues and their influence on the analysis environment, we have to select events and parameters that will be traced. Three typical event types in similar researches are allocation, deallocation, and reference assignment. For each event, a set of context parameters is traced:

- allocation (time of allocation, requested size, object's address, type of object, method in which allocation occurred),
- deallocation (time of deallocation, released size and address, method in which allocation occurred),
- reference assignment (time of assignment, old and new value of reference, reference type, reference location, method).

Unlike some other researches, we do not rely on trace driven simulation in order to analyse collected data. Instead, we collect parameters that we use directly in further analysis.

4. IMPLEMENTATION

The analysis environment, that we have implemented, collects data obtained by execution of benchmarks written in Java. We have used the Kaffe virtual machine [Lindholm and Yellin, 1999; Venners, 1999; Kaffe.org, 2004]. The implementation is based on modifications of the Kaffe source code, used to interpret byte codes responsible for memory manipulation.

The analysis environment consists of benchmarks (i.e. test programs), instrumented Java virtual machine (Kaffe), transport layer (two file formats or TCP/IP connection), middleware layer, database, and analysis software (figure 1) [Borozan, 2004].

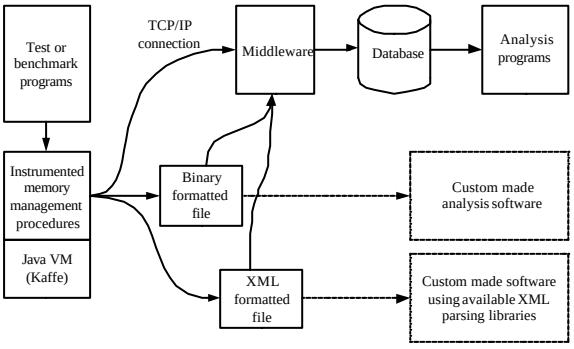


Figure 1: The organization of the analysis environment

As data format we have used both, binary and ASCII (XML) format. Depending on the selected data format, the data can be analysed in three possible ways. Analysis of an XML file is a matter of deployment of XML-parsing libraries in a custom-made program. Analysis of a binary file would require development of custom made software, which would probably be less flexible and extensible.

However, the results are finally stored into a database and we used the third approach, which uses a database as a mean for analysing the profiled data. The analysis is performed by a sequence of SQL queries given through SQL-shell interpreter or custom-made program using SQL libraries. This approach provides an efficient way of searching through large amount of data. Since there is no need to develop custom-made analysis software, the whole process is simplified and is very flexible.

According to his/her needs (or desires), a user can select whether to store data to one of two possible file formats or to make a direct network connection to the middleware. Network connections are much slower than disk access, which can introduce significant overhead on the analysis environment. This overhead can be avoided if the data are first stored in the binary file, and latter moved to the database by using the middleware.

The results consist of a series of events and their parameters. This enables later reconstruction of benchmark execution and its analysis. E.g., parameters are time of event, class of the object, method where event occurs, etc. The precise death-time of an object is determined by using the Merlin algorithm [Hertz et al, 2002]. It means that the overhead of the profiling is relatively small, since GC does not have to be triggered at every GC point. In fact, during the benchmark execution, GC is triggered by the JVM (without our influence). The only time when we have to trigger GC is on the end of the benchmark (in order to let the Merlin to determine the death-time of objects that had survived the last automatically triggered GC).

The performance of the profiling process itself is quite satisfactory. The collection process is 26% slower than the normal execution of benchmarks in the case of the binary trace format, and 62% slower in the case of the XML format. The analysis of all results and for the whole set of Java-Olden benchmarks (10 benchmarks) lasts less than 8 hours. The measurement is done on a fast PC with two Pentium processors (2.4 GHz) and 2 GB of memory (although the usage was never 100% on all processors) [Borozan, 2004; Borozan and Basch, 2004a].

Other studies rarely give the data about the performance of the profiling. Also, the computers used for the profiling have different performances, and the profiling is organized differently. Therefore,

the comparison is difficult. For example, in [Hertz et al, 2002] it is stated that some benchmarks take about an hour to execute (but employing only the Merlin algorithm, without data collection and analysis). These measurements were taken on a double processor PowerPC (533 MHz) with 348 MB of memory. When using the simulation for the analysis [Dieckmann and Hoelzle, 1999], the whole process becomes even slower: the analysis of one benchmark takes four days on Sun UltraSPARC workstation (300 MHz).

5. RESULTS

Here we present some of the results that can be obtained by the analysis environment. We have used two sets of benchmarks: Java Olden [Cahoon, 2004] and Colorado [Henkel, 2004]. Java Olden consists of ten benchmarks and we have used all of them (*bh*, *bisort*, *em3d*, health, *mst*, *perimeter*, *power*, *treeadd*, *tsp*, and *voronoi*), and Colorado consists of four benchmarks, three of which we have used (ipsixql, *nfc*, and *xalan*). We informally divide benchmarks into computational, data-driven, server and interactive. Their characteristics are described only briefly. Interactive programs are not described because the selected benchmarks do not have such programs and because they are not convenient for performance measurement (the measurement cannot be repeated precisely since it depends on human interaction).

- Computational applications are not interactive. They mainly create objects at the beginning, and then they perform computations mainly using the created objects and creating new ones only occasionally. Most of the objects are freed at the end. For such programs, the GC pauses are less important than the total GC overhead.
- Data-driven applications also have low interactivity, but they allocate/deallocate objects during the execution more intensively than computational programs. Their work is usually driven by a large amount of data (read from file or generated during the program). Again, the total GC overhead is the most important.
- Server applications are characterised by continuous and long-run executions that should not be terminated and often should not be paused for larger periods. The new objects are mainly allocated after the request from the client is received, and the same objects are freed after the request is served. The GC pauses can be important as well as precise reclamation of objects since the remaining garbage can fill-up the memory after the long run.

Among different results that can be obtained with the analysis environment, we present only some of them. They are lifetime of objects, objects death-time distribution, reference density in objects and arrays, type and lifetime of objects, age and the number of reference assignments inside objects, and age and the number of reference assignments toward objects. For each results, the measured value is explained. In addition, we briefly explain how the measured value is extracted from the database. We describe several representative graphs selected among all benchmarks. Finally, the results are explained from the viewpoint of their applicability in GC research.

Part of the results from this paper are presented in [Borozan and Basch, 2004b] but in the meantime we have changed the profiling process in few aspects. The most important change concerns the memory objects that are traced. Before, we traced all allocations in the Kaffe VM. Now we trace only objects that are instances of Java types. Both approaches have their advantages. The former approach gives better insight into behaviour of memory objects in particular implementation of VM (Kaffe in this case). More general results are obtained with the latter approach since they are more VM-independent. However, predefined types and types from standard libraries still depend on the particular VM-implementation, but this dependency cannot be excluded easily.

5.1 Lifetime of Objects

Lifetime (or age) of objects is typically measured in the research of GC algorithms. Lifetime is the age of the object in the moment of its death. The moment when the object is deallocated during GC-cycle is not considered as death. Rather, the real death of the object is the moment when the last reference to the object stops pointing to it. For each benchmark, the lifetime of all its objects is presented in this section.

The lifetime is simply calculated by subtracting birth time and death-time (both times are stored in the database) for every object in a particular benchmark. The obtained lifetimes are presented as a histogram.

Although it is a common practise to present the same results as a cumulative function, we feel that the histogram is more appropriate for the purpose of identifying the patterns of object lifetimes. A similar problem exists with the cumulative distribution function of random variables, since for the different variables their cumulative distribution functions have visually similar shape (so called S-shape). Probability density functions have shapes that are more distinctive. Since they are analogous to the histograms, we use that form for the presentation of the results. A cumulative function is more appropriate for identifying minimal age for which majority of objects die, and in that case the mortality and survivor functions are used [Stefanovic et al, 1998].

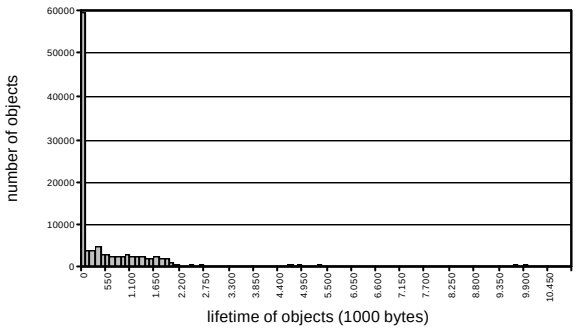


Figure 2: Lifetime of objects in *bh*

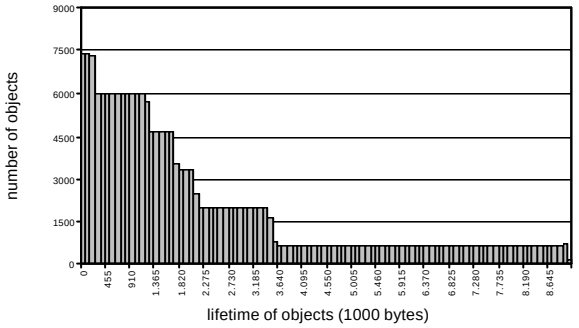


Figure 3: Lifetime of objects in *tsp*

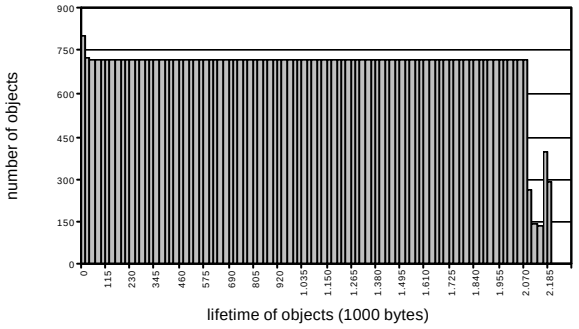


Figure 4: Lifetime of objects in *bisort*

The first category of results conforms to the weak generational hypothesis ("most objects die young"). The histogram has a peak in the youngest bins, and older bins have an insignificant number of objects (figure 2). Seven benchmarks belong to this category. Figure 3 shows that *tsp* has similar shape, but with the number of objects that fall of more smoothly (similar to exponential decay, at least at the first sight).

Bisort, *perimeter*, and *treeadd* form the second category. They have approximately equal number of objects uniformly distributed over all bins (figure 4). These results are in contrast to the generational hypotheses and generational collector will obviously have a poor performance for such benchmarks.

The third category includes e*m3d* (figure 5) and *mst* (figure 6). They have irregular histograms, and *em3d* even has a distribution where most objects die old, which is opposite to the generational hypothesis. From the other results, it will be seen that the object behaviour is similar for the second and the third categories. The benchmarks from the last two categories mostly belong to the numerical type of applications.

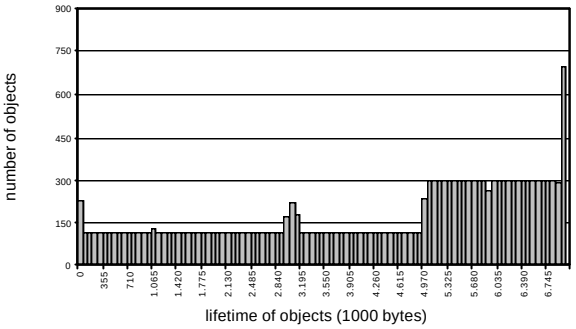


Figure 5: Lifetime of objects in *em3d*

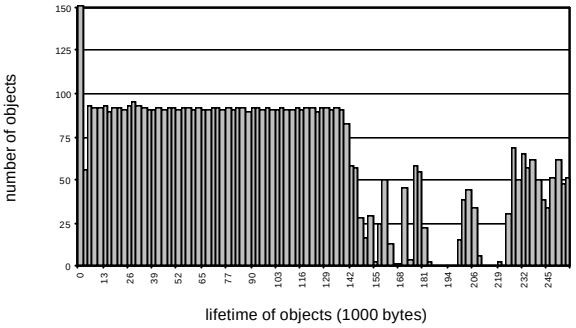


Figure 6: Lifetime of objects in *mst*

Generational algorithms are probably the most widely used in modern collectors. These results show that generational algorithms may significantly degrade the performance of certain types of applications. Theoretically, this could be easily recognized in run time, and even the other algorithm could be activated in such case. However, the write barriers (for inter-generational pointers) cannot be efficiently deactivated and the performance will still be affected. In case of static profiling before the application is released, it is easy to observe that generational collector is inadequate, and some other algorithm can be used instead of it.

To simplify further analyses, the objects are categorized according to their age in three groups [Hudson and Moss, 1992]:

- immortal objects die at the end of the program (regardless of their lifetime); remaining objects are either shortlived or longlived,
- shortlived objects are those, whose lifetime is lower than the 20% of high-watermark,
- longlived objects are those whose lifetime is higher than the 20% of high-watermark.

High-watermark is the theoretically minimal size of the heap needed to execute a particular program (or maximal size of the live objects). Immortal objects

can be further divided into truly and quasi immortal [Hirzel et al, 2002]. We decided not to do so, because it was shown that the number of quasi immortal objects is small in comparison to the number of truly immortal ones [Hirzel et al, 2002].

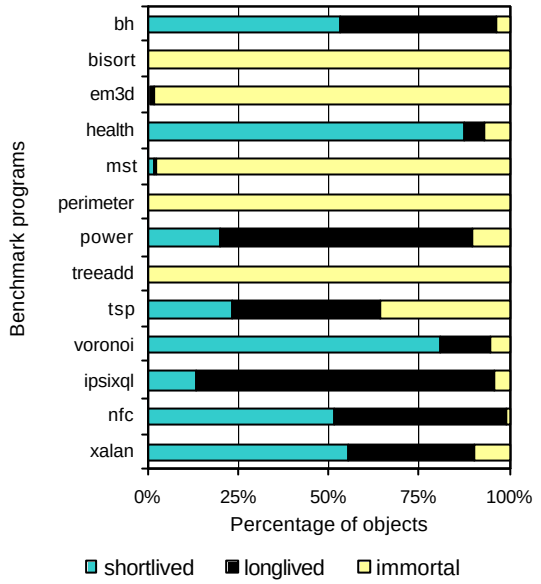


Figure 7: Lifetime groups for all benchmarks

Figure 7 shows the distribution of number of objects across three lifetime groups. These graphs represent the previous results more concisely. It can be seen that in several benchmarks majority of objects are immortal. The rest of the benchmarks have objects with the mixed lifetimes, and they belong to the first category of shapes of lifetime distributions.

5.2 Objects Death-time Distribution

Objects die during an entire application run but not with the same intensity. This analysis explores the distribution of objects' deaths from the application start until its termination. The results are organized as a distribution of number of deaths in particular bins of the histogram.

The results are obtained very simply - for each object, its time of death is extracted from the database and is used as the data for the histogram.

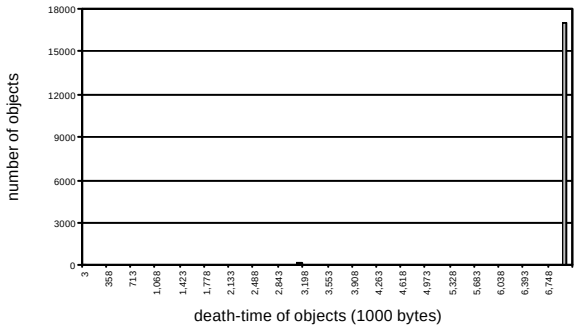


Figure 8: Death-time of objects in *em3d*

The first typical shape of death-time distribution has a high peak at the end of the benchmark execution and remaining bins are nearly empty in comparison to the peak (figure 8). Such behaviour can be observed in *bisort*, *em3d*, *mst*, *perimeter*, and *treeadd*. These are exactly the benchmarks with the majority of immortal objects from the previous section which is obvious, since almost all objects die at the very end of the execution. These benchmarks have two distinct shapes of lifetime distribution - uniform and irregular - but they still have the same shape of death-time distribution. The explanation lies in the way the time is measured. For benchmarks where most objects have similar size, the lifetime distribution is more uniform, since the time advances in regular steps. For benchmarks with different sizes of objects, the time advances "slower" or "faster" and the lifetime has irregular distribution. Death-time distribution reveals that in both cases the majority of objects are immortal.

Shapes from the second group also often have a peak near the end, but the peak does not contain the majority of objects as in the first shape type. Remaining bins are not empty. The intensity of deaths in those bins can be almost constant (figure 9) or more irregular (figure 10). Sometimes, such behaviour is superimposed with several peaks distributed more or less evenly across the whole program run (figures 11 and 12). Similar peaks exist in figure 10, but they are lower. This means that the objects die during the entire application run with similar intensity, but in addition sometimes there are larger groups of objects that die simultaneously.

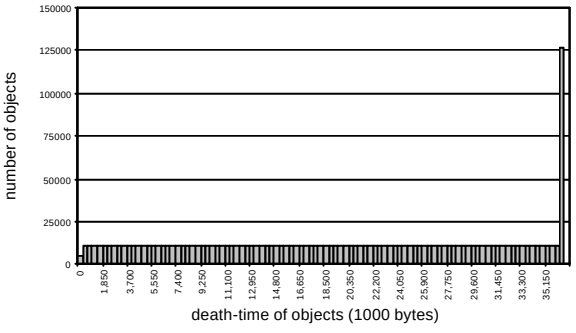


Figure 9: Death-time of objects in *health*

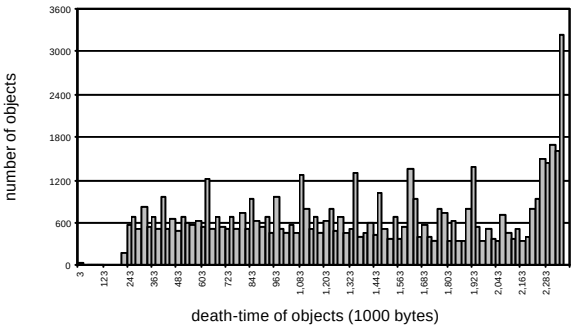


Figure 10: Death-time of objects in *voronoi*

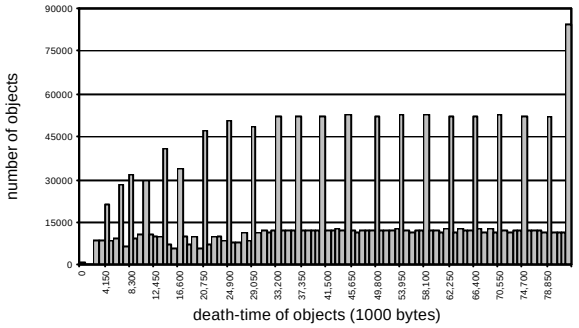


Figure 11: Death-time of objects in *ipsixql*

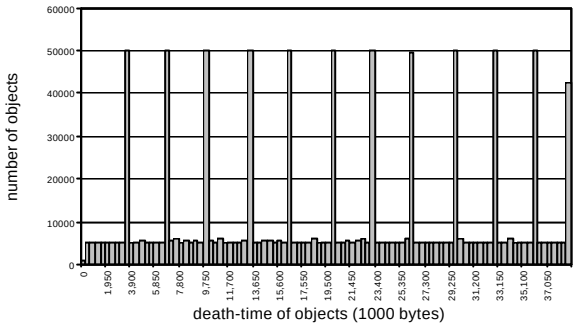


Figure 12: Death-time of objects in *bh*

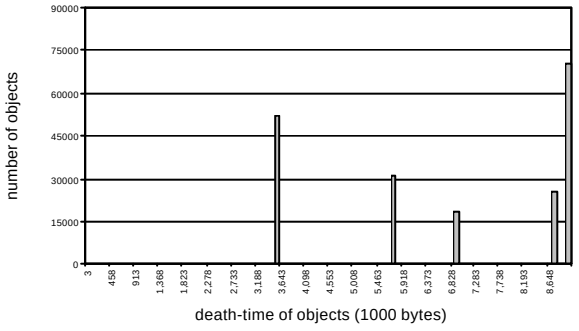


Figure 13: Death-time of objects in *tsp*

Figure 13 reveals that objects in *tsp* do not obey generational hypothesis, as it seems from the life-time distribution. The objects in *tsp* die in only five distinct moments, and many objects are immortal.

Death-time results show that conclusions drawn only from lifetime distribution can sometimes be misleading. They also show that it is true that objects tend to die in groups (i.e. clusters), although sometimes there is only one large group of immortal objects. It means that key-object based collectors [Hayes, 1991] may be even more generally applicable than pure generational ones.

5.3 Reference Density in Objects and Arrays

In OO languages, every object in a memory consists of fields (members, instance variables). In Java, the fields can be classified as either references or data. This classification is of interest for GC since only references connect objects in the heap. We shall say that a field is a data field if its type is primitive (integer, Boolean, character, etc.). This section presents the results regarding to the heap composition: the type of memory objects (objects or arrays) and their content (data or references). In this section, the term "object" is used strictly in the sense "instance of a class", since an array is not an instance of a class. Objects are heterogeneous in the sense that they can have arbitrary number of reference and data fields. Arrays are homogenous, since all their "fields" have the same type. The purpose of applications dictates the structuring of the objects. Therefore, it could be expected that this group of results will have low regularities.

All results in this section are obtained simply by extracting the size for every memory object and eventually by extracting the number of references from the corresponding class. The size of the memory objects includes only fields since their other parts depend on the implementation of the virtual machine. Other parts can be a virtual table pointer or class information, GC fields, and allocator fields.

First, for every benchmark we give some basic information about sizes of objects and arrays (table 1). The minimal (*min*) and the maximal sizes (*max*), as well as the average size (*avg*) and the standard deviation (*s*) are given. As it is expected, the arrays are typically larger than objects, which can be of interest for memory allocation and GC. For all Olden benchmarks minimal and maximal sizes of both, objects and arrays, are the same (12 and 88, 16 and 164000). These benchmarks use only small objects and arrays, and the extreme sizes belong to internal classes of Kaffe and classes from libraries. The overhead of additional fields for GC (or allocator) can be estimated from that data.

Table 1: Sizes of objects and arrays

Bench-mark	Objects				Arrays			
	min	max	avg	s	min	max	avg	s
bh	12	88	16.3	2.5	16	16400	40.2	30.8
bisort	12	88	24.1	1.0	16	16400	147.5	928.7
em3d	12	88	41.8	6.2	16	16400	534.2	294.0
health	12	88	20.9	2.2	16	16400	32.4	51.5
mst	12	88	21.4	6.1	16	16400	144.3	866.9
perimeter	12	88	36.0	0.5	16	16400	156.8	881.4
power	12	88	31.3	6.3	16	16400	32.1	24.8
treeadd	12	88	24.0	0.3	16	16400	141.6	917.2
tsp	12	88	37.3	7.6	16	16400	152.8	947.6
voronoi	12	88	28.2	3.8	16	16400	38.8	215.3
ipsixql	12	130	34.0	11.6	16	65535	39.1	392.3
nfc	12	112	25.8	5.8	16	20153	161.5	257.6
xalan	12	460	36.7	15.2	16	61466	148.9	875.5

Results in figure 14 show the heap composition (in percentages) regarding the total sizes of reference fields and data fields in all allocated memory objects (objects and arrays). Although the size ratios vary

without regularity, it can be seen that the size of the references is often relatively high (around a half or more). It could be noted that, on average, computational applications have more references (two thirds) than the data-driven applications (one third, but with a high variance).

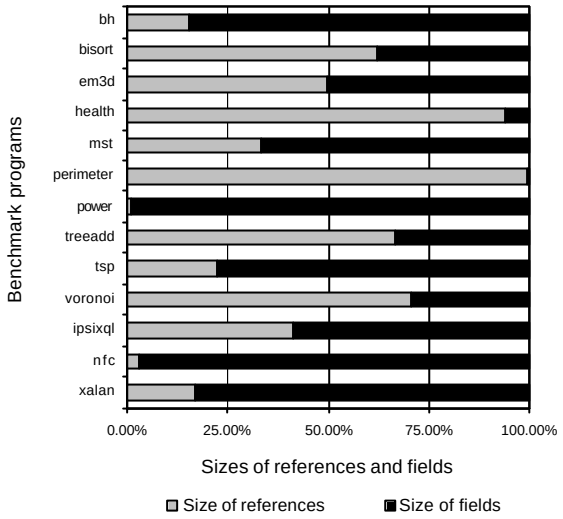


Figure 14: Total sizes of reference fields and data fields in memory objects (objects and arrays)

These results can be used for estimating the amount of work performed by a collector. The number of references affects scanning that is performed in marking and copying collectors. The number of references also affects the pointer update that must be done if the memory objects are moved during the collection (copying- and compacting-based collectors). Depending on the algorithm, references are updated while moving the memory objects, or afterwards in one or more extra passes. For higher amount of references, copying collectors may be more appropriate than compacting ones. The boundaries for applicability of a particular collector should be determined experimentally.

Next, the sizes of objects and arrays are given in figure 15. Again, there is no regularity. Since arrays are larger than objects, arrays can be treated in another way. Repeated copying of large memory objects can significantly degrade the performance of GC. The situation is even worse when large memory objects are long-lived. The collection of arrays should not be improved at the expense of collection of objects, since it cannot be assumed that arrays are larger than objects in the majority of programs.

Finally, the total sizes of arrays of references and all other memory objects in the heap (non-array objects and arrays without references) is given in figure 16. The results show that arrays of references occupy small portion of heap (the only exception is *em3d*). The number and position of references in objects/arrays affects their scanning in the absence of type information (e.g. in conservative collectors). Such scanning must check all fields in objects/arrays and treat as a pointer every data that may be a pointer. For arrays, it would be enough to check only the first element of an array (provided it is not a null-pointer, and that arrays can be distinguished from other objects). For precise (i.e. type-aware) scanning, most arrays can be treated as leaf-objects, which is favourable for any kind of collector.

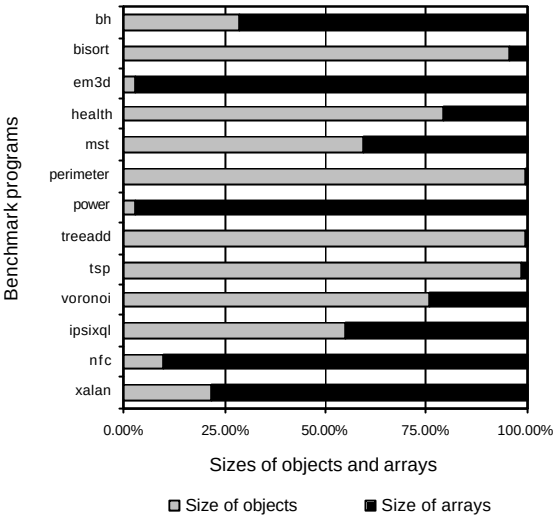


Figure 15: Total sizes of objects and arrays

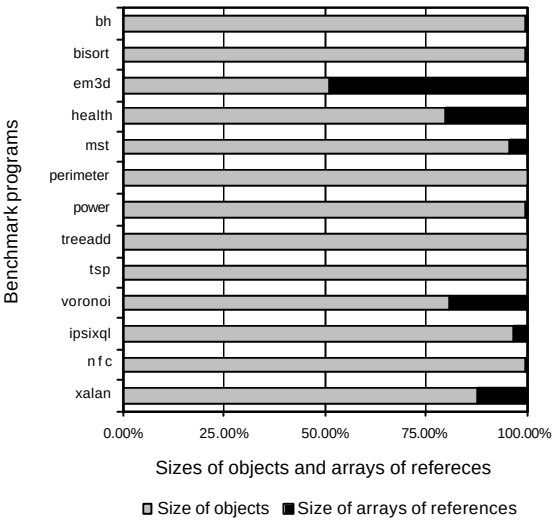


Figure 16: Total sizes of arrays of references and all other memory objects

5.4 Type and Lifetime of Objects

The results presented in this section relate the object lifetime with the object type. Types with less than 100 instances are not included in the results since their influence on memory behaviour and GC is insignificant (and since they would make the graphs too large). The number of instances varies widely from type to type, so the absolute numbers of instances for some types is small and ratios between lifetime categories cannot be seen (left parts of the

figures). Therefore, we additionally present the same results showing percentages of lifetime categories (right part of the figures). However, the absolute number of instances is significant for the influence that particular type (i.e. its instances) has on the behaviour of memo ry and GC.

Benchmarks whose objects belong to mostly one lifetime group (i.e. *bisort*, *em3d*, *mst*, *perimeter*, and *treeadd*) are not shown in this section (cf. figure 7). *Nfc* is not shown because its graph is too large (since it uses many classes).

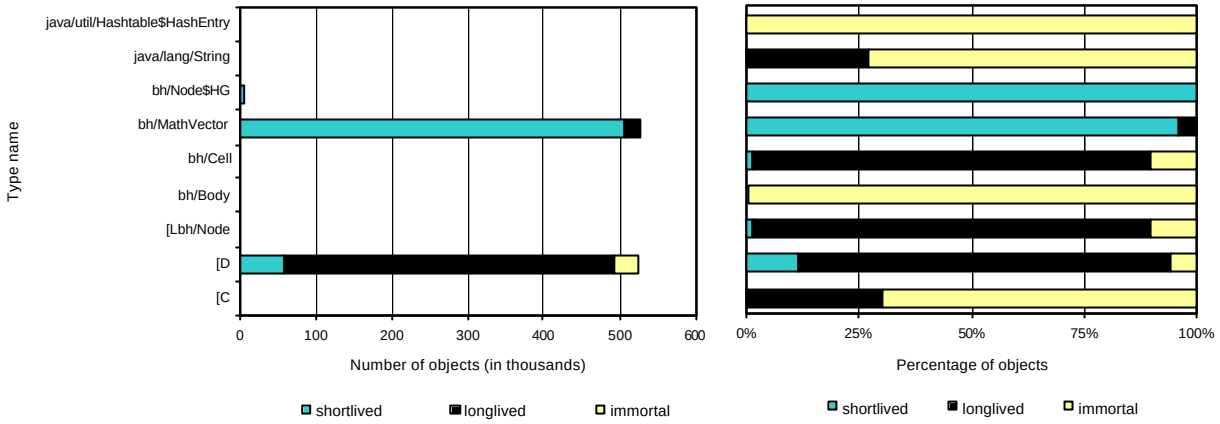


Figure 17: Objects lifetimes per object type for *bh*

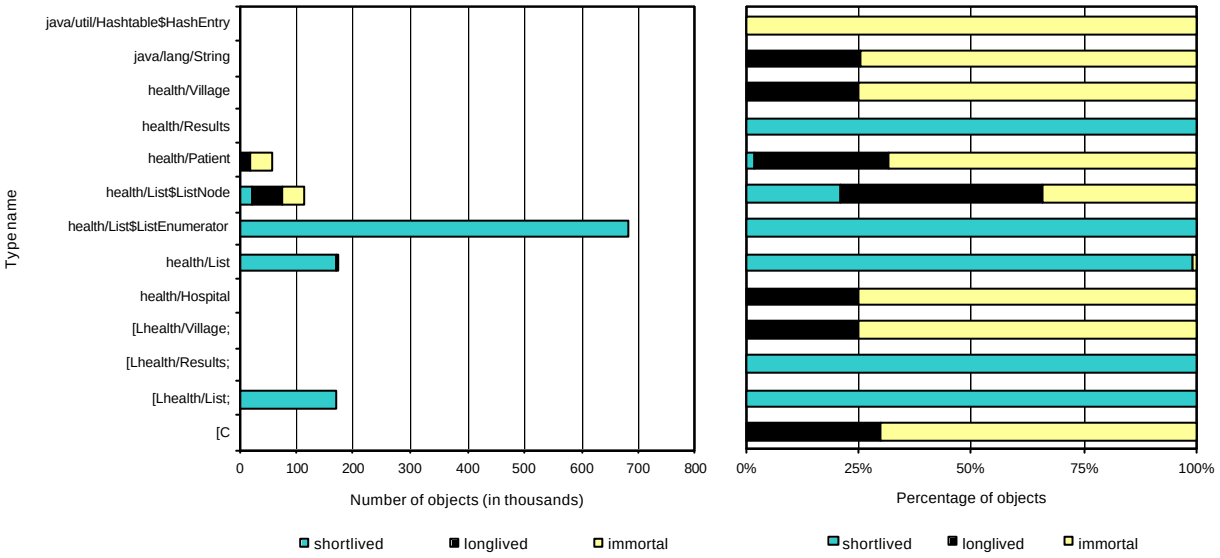


Figure 18: Objects lifetimes per object type for *health*

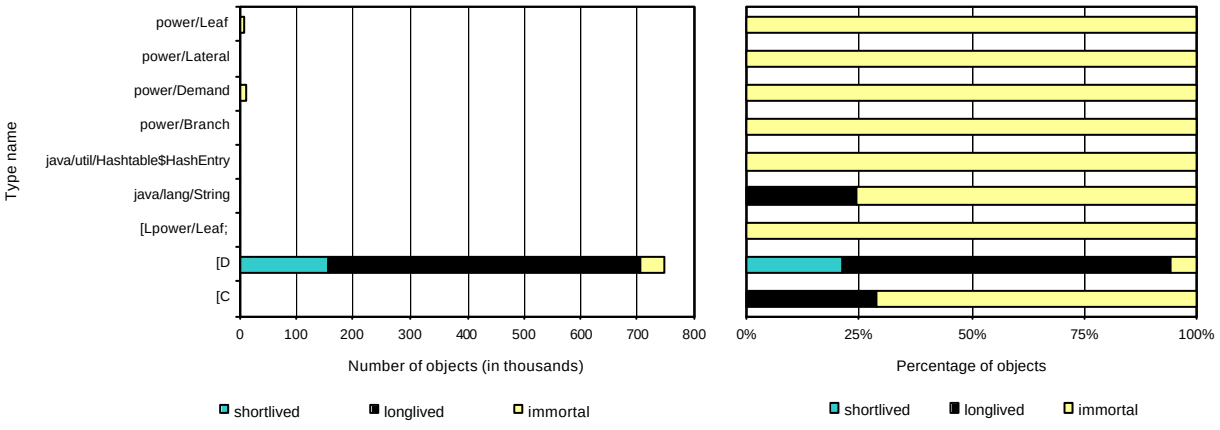


Figure 19: Objects lifetimes per object type for *power*

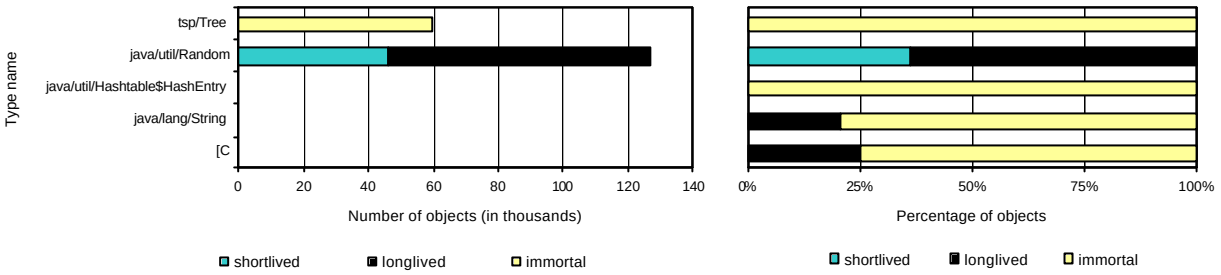


Figure 20: Objects lifetimes per object type for *tsp*

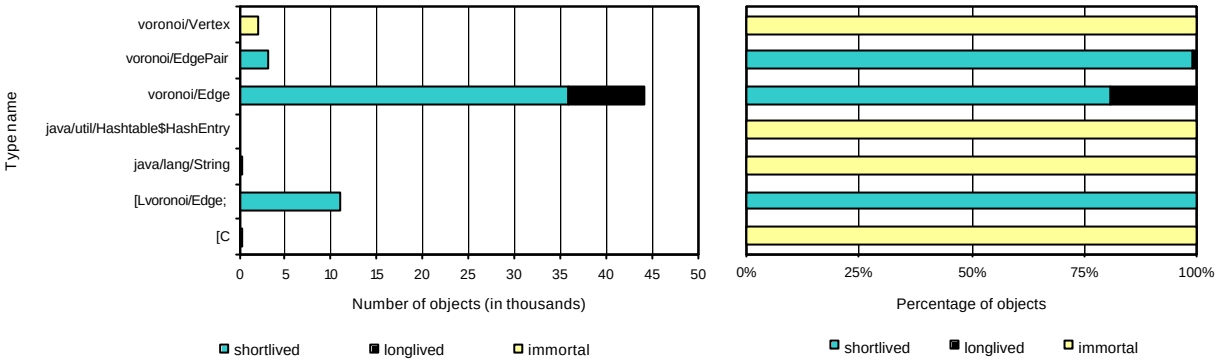


Figure 21: Objects lifetimes per object type for *voronoi*

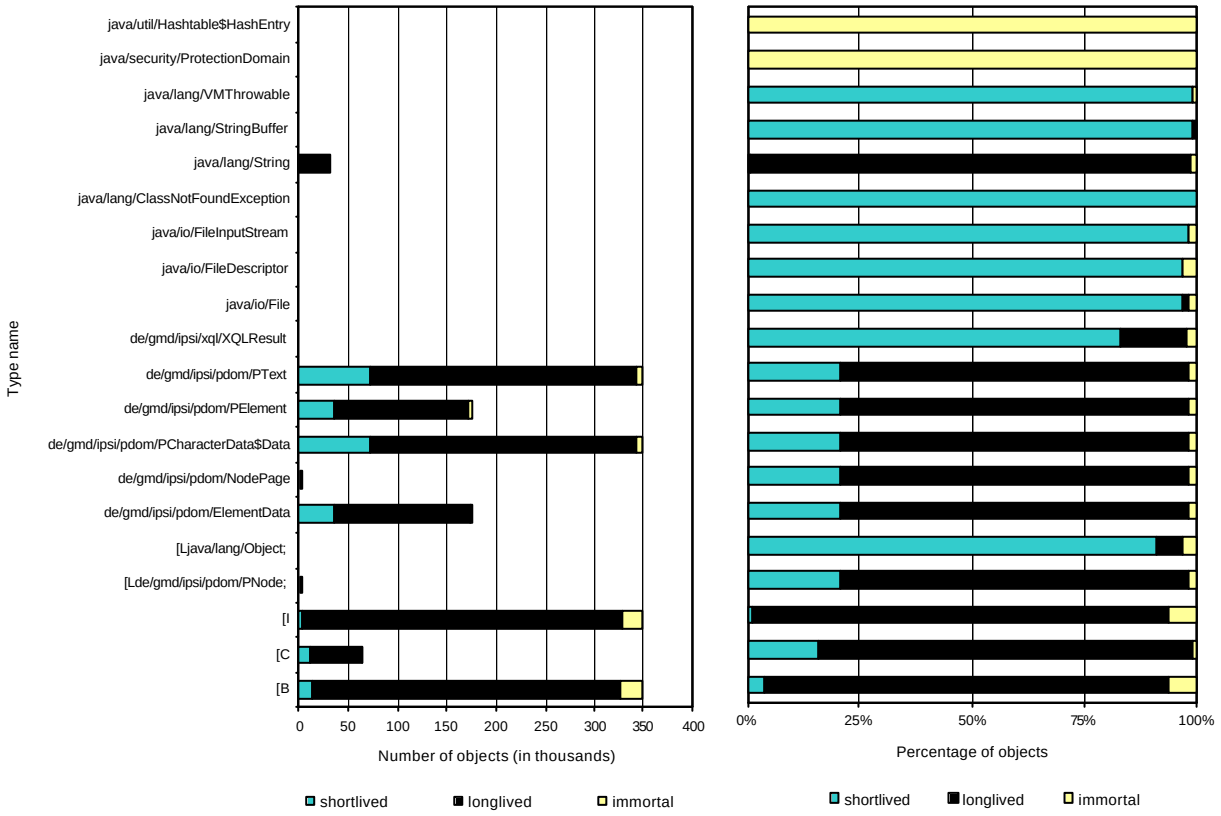


Figure 22: Objects lifetimes per object type for *ipsixql*

The results are obtained by calculating the age of the object in the moment of object's death, and then by categorising each object into shortlived, longlived and immortal (as described in section 5.1). Then, for all types, all of their instances belonging to each lifetime category are counted. The percentage graphs and the graphs with the absolute numbers of objects use the same data.

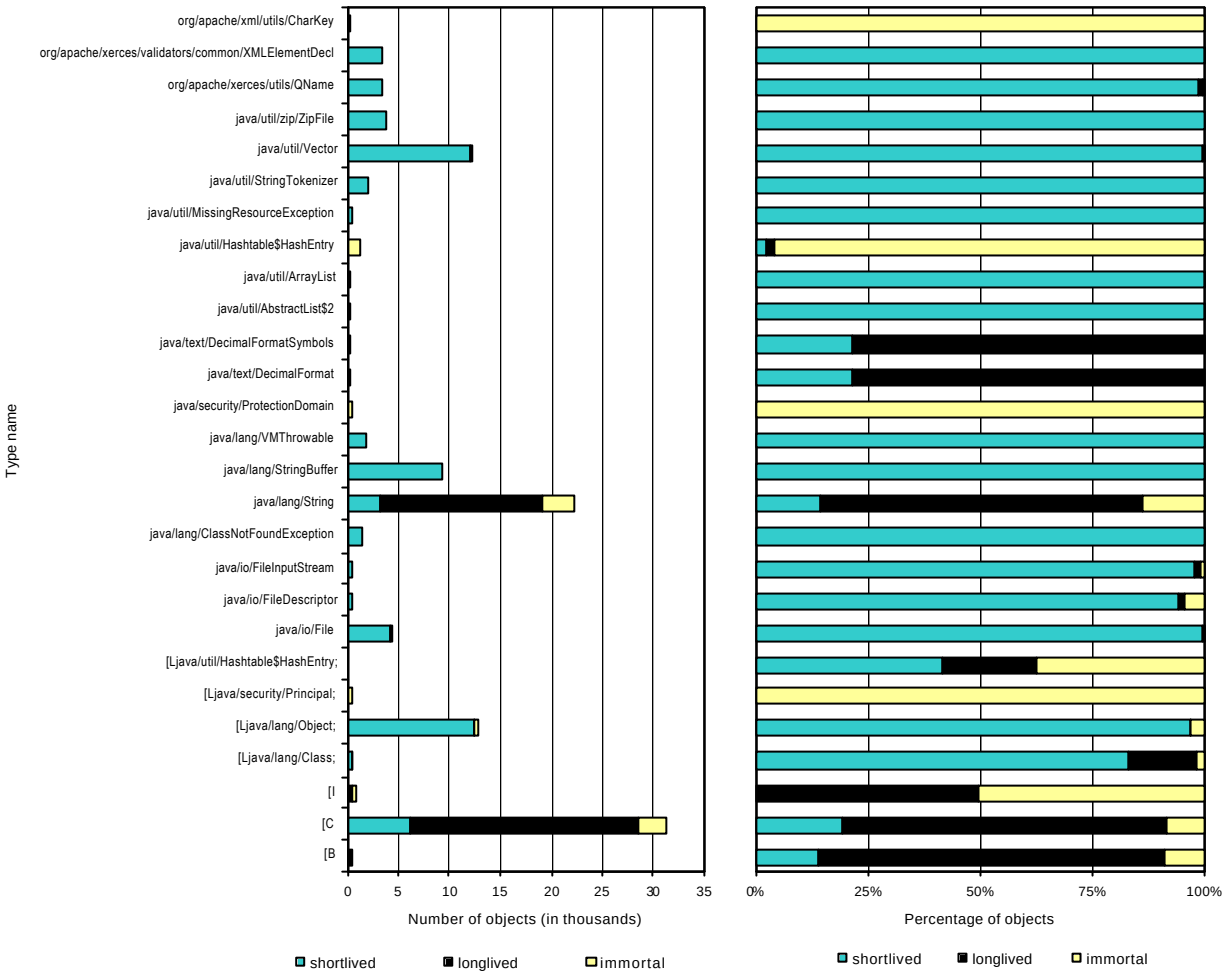


Figure 23: Objects lifetimes per object type for *xalan*

The percentage graphs (figures 17-23, right sides) show obvious correlation between type and lifetime. For most of the types, almost all their instances belong to only one lifetime category. Some types have instances that belong to two categories, but in majority of those cases, one category is also dominant, since the other category typically comprises about 20% of instances (or even less). Two categories that are combined together are either shortlived and longlived, or longlived and immortal.

The more "distant" categories, i.e. shortlived and immortal occur very rarely together in one type. In addition, one type rarely has all three categories with significant percentages. These two cases are relatively rare but their occurrence is restricted to few types. Such types are built in types (Integer, String, File, etc.), arrays (type names that begin with "["), or some similar "generic" types (HashEntry, ListNode, etc.). The opposite relation does not hold: in many cases instances of these types have the usual distribution of lifetimes. Diverse behaviour of these types is easily explainable, since such types are often used for different purposes in a single program, and hence the instances have different lifetimes. User-defined classes are much more strongly correlated with lifetime and behave much more regularly.

These results show that the object's type can be used for prediction of object's lifetime. Although this may seems ideal, the absolute number of instances should also be taken into account (figures 17-23, left sides). Unfortunately, the most often-used types show somewhat less regular behaviour than the less used types, but the correlation between type and lifetime is still strong.

The basic idea in exploiting the type information is to obtain a hint about object lifetime for every newly allocated object. For generational collectors, this may be used to allocate an object directly in the older generation (pretenuring). For some other GC algorithm, this hint may be used in another way, e.g., for copying GC algorithms the immortal object can be allocated in a region that is not copied. It is obvious that each program uses different types in different way. Therefore, for each program the correlation between lifetime and type is different, which means that to correlate them one should use a static or a dynamic profiling.

5.5 Age and the Number of Reference Assignments Inside Objects

Each object can have fields that contain references. The references make connections to other objects. This analysis explores the relation between the current object's age and the number of changes in the connectivity between this object (as a parent) and the other objects (as children of that parent). The results are organized as a distribution of number of reference assignments inside parent objects belonging to certain age bins and displayed as a histogram.

The results are obtained in the following way. For each reference assignment event in the database, the age of the object (containing reference) in the moment of the assignment is used as data for the histogram.

Most of the benchmarks show the same behaviour: almost all reference changes are concentrated in the youngest bin, which means that references are updated almost exclusively in the youngest objects. Such behaviour is observed in: *bh*, *health* (figure 24), *mst* (figure 25), *perimeter*, *power*, *treeadd*, *voronoi*, *ipsixql*, *nfc*, and *xalan*. This comprises ten of thirteen benchmarks. Most of the benchmarks have shapes similar to figure 24, and only two of them have small number of reference assignments distributed in older objects as in figure 25.

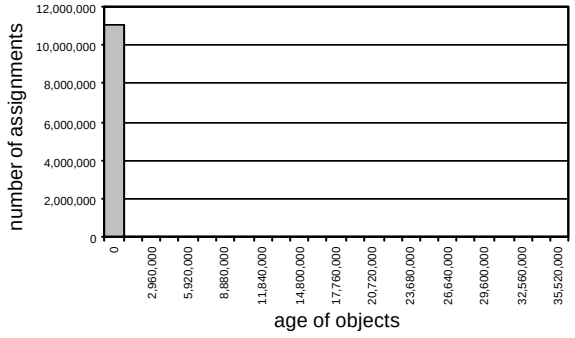


Figure 24: Number of reference assignments inside objects per objects' age for *health*

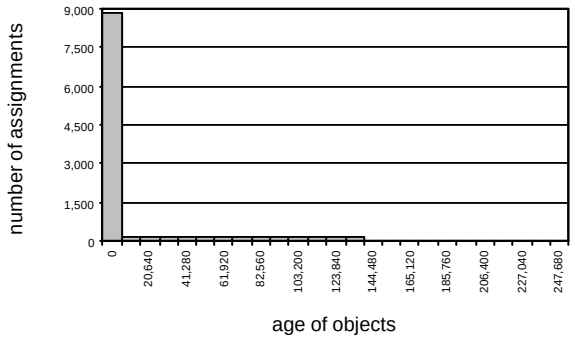


Figure 25: Number of reference assignments inside objects per objects' age for *mst*

Bisort (figure 26) and *tsp* (figure 27) have changes distributed evenly through all ages. Although the mode is also in the youngest bin, the percentage of changes in younger bins is not significantly higher than in older bins.

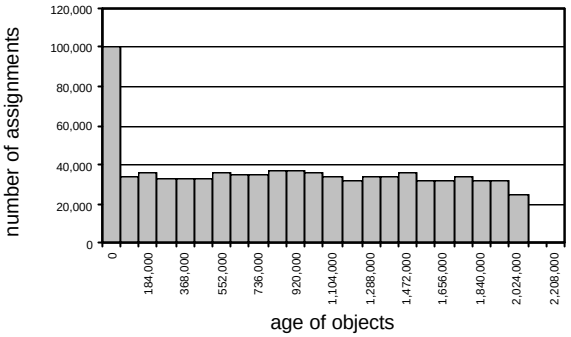


Figure 26: Number of reference assignments inside objects per objects' age for *bisort*

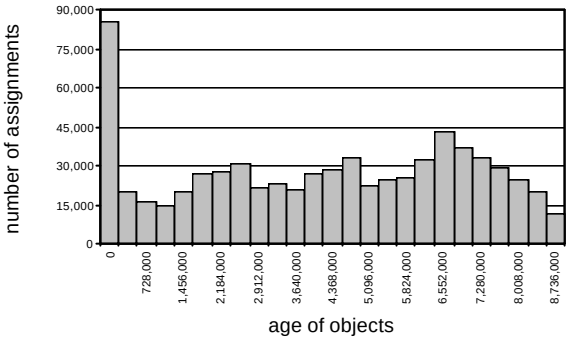


Figure 27: Number of reference assignments inside objects per objects' age for *tsp*

Em3d (figure 28) has combined behaviour where changes are distributed through all bins, but they are more concentrated in the younger bins.

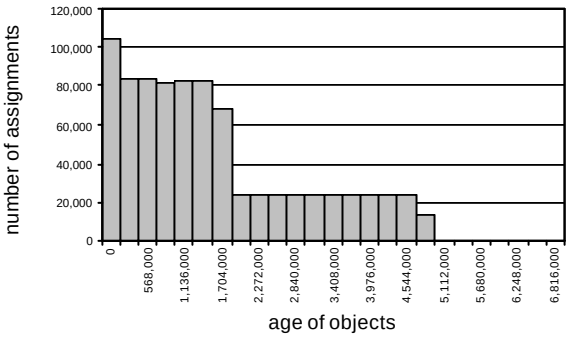


Figure 28: Number of reference assignments inside objects per objects' age for *em3d*

High regularity of these results shows that the object graph in the memory is changed on the graph's "border". By "border", we mean the newly allocated part of the graph. This fact can be exploited in several ways.

First, this is an indication that for generational collectors, only a small number of inter-generational pointers from older to younger generations will be created, reducing the memory requirements for recording such pointers. The number of executions of the write-barrier remains the same since it must intercept all assignments to references on the heap. However, the cost of the write barrier can be reduced if the parent age is checked first in the barrier. The parent address must be known in the moment of assignment, so additional memory-read operations are not necessary and the check is fairly simple and efficient. This check will rule out the further execution of the barrier for the majority of the assignments since the percentage of stores in the old objects is low.

Second, incremental marking algorithms should try to visit the older objects before the younger ones. This would reduce the problems of read/write barriers and colour changes that have to be done to maintain the correctness of the tricolour marking scheme. Again, the barriers still have to be executed, but they can be optimized. The colour changes would be reduced, and this will make the marking either faster or more precise.

Third, marking algorithms can be adapted in a way to accelerate the marking phase. Since the reference changes are rare inside the older objects, they should not be visited in every GC-cycle. Instead, the marking should be concentrated on younger objects (similar to the idea that leads to the development of generational algorithms).

5.6 Age and the Number of Reference Assignments Toward Objects

Previous section explores the number of reference assignments inside the parent objects. Similar results can be obtained for the child objects. This analysis explores the relation between the current object's age and the number of reference assignments that establish connection toward that object. The results are again organized as a distribution of number of reference assignments toward child objects belonging to certain age bins and displayed as a histogram.

To obtain the results from the database, for each reference assignment event, the age of the object (whose address is stored in the reference) in the moment of the assignment is used as data for the histogram.

Most of the benchmarks show similar behaviour as in the previous section (figure 29). Benchmarks that have majority of reference changes inside the youngest objects, again have most reference changes toward the youngest objects. The slight difference in comparison to previous section is somewhat higher number of changes toward older objects (although this number is still low in comparison to the youngest bin). The only exception is health (figure 30). In *health*, the number of referenced objects is approximately inversely proportional to the age (for objects younger than 24 MB), with the exception of the youngest bin, which is referenced more often than the other bins, but again not significantly more.

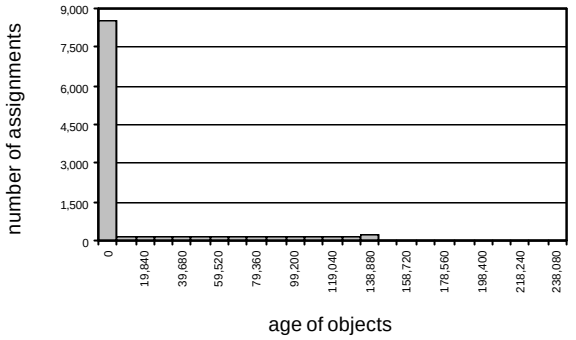


Figure 29: Number of reference assignments toward objects per objects' age for *mst*

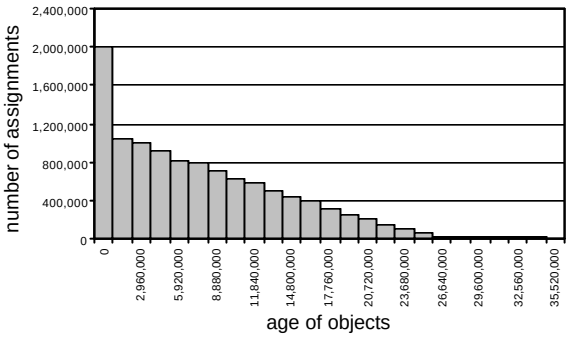


Figure 30: Number of reference assignments toward objects per objects' age for *health*

Bisort (figure 31) and *tsp* (figure 32) have similar graphs as in the previous section (cf. figures 26 and 27). Beside shapes, the graphs are also similar in numerical values. Because of high similarity, it can be assumed that two objects that become connected have very close ages (but this is not for sure).

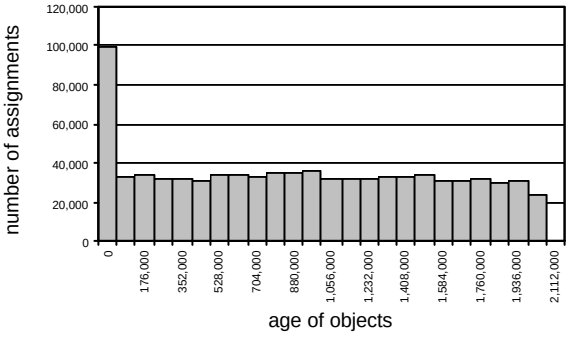


Figure 31: Number of reference assignments toward objects per objects' age for *bisort*

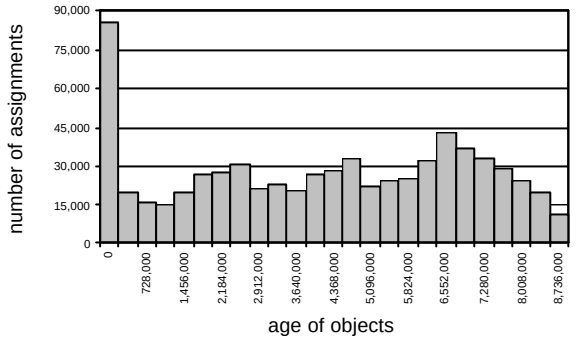


Figure 32: Number of reference assignments toward objects per objects' age for *tsp*

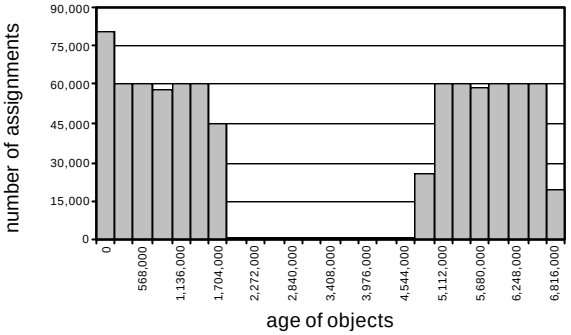


Figure 33: Number of reference assignments toward objects per objects' age for *em3d*

Em3d has different shapes of distributions for parents and children. In *em3d*, the majority of reference assignments toward children objects belong to younger and older bins, and the bins between are almost empty (figure 33).

Regularity of these results reinforces the findings from the previous section. They show that the connections in the object graph are established on the graph's "border". Now, we can revise some of the conclusions from the previous section.

Since the youngest objects are often referenced, the old-to-young pointers will probably be established whenever the parent is old. This further confirms the proposed order of checks that should be done in the write-barrier in generational algorithms: only after the parent age is checked, the age of the child should be examined.

For marking order (or marking avoidance), the main problem is to determine the objects where the marking should stop (or start). However, this needs further investigation, which is not related to profiling but rather to GC algorithms.

6. CONCLUSION

In this paper, we describe the analysis environment for profiling and analysis of behaviour of memory objects in object oriented languages. We use Java as a representative language for OO paradigm. The profiled data are stored in a database, and the analysis is performed by SQL-queries, which also provides a flexible way to obtain new results. This approach has proved as efficient since there is no need for simulation in order to replicate the execution of benchmarks.

Several groups of obtained results are presented and commented. Some results confirm assumptions about behaviour of memory objects known about the other programming paradigms. Other results show no regularity usable in GC research. Nevertheless, some results show certain patterns of behaviour of memory objects that can be employed in development of new GC algorithms or in improvement of the existing ones. We show and explain such examples in the obtained results. For example, the composition of heap is measured; the correlation between objects' class and its lifetime is found; the changes of the connections on the heap are found to be located in the youngest objects. We give several ideas how these findings could be used.

The future work should use more benchmark programs in order to obtain results that are more reliable and that represent wider class of applications. In addition, different kinds of results should be analysed by creating more SQL-queries. The analysis environment can also be enhanced and improved. E.g., the measure of time could be refined by introducing sub steps of time, because now the reference assignment events cannot be precisely related to object deallocation events. That refinement could be used in the investigation of causes of objects' death.

REFERENCES

Blackburn S.M., Singhai S., Hertz M., McKinley K.S. and Moss J.E.B. 2001, "Pretenuring for Java". In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '01)* (Tampa Bay, USA, October) ACM Sigplan Notices 36(11). Pp342-352.

Borozan J. 2004, *"Tracing and Analysis of GC-Related Events in Object Oriented Languages"* (in Croatian). M.Sc. Thesis, University of Zagreb, Zagreb, Croatia.

Borozan J. and Basch D. 2004a, "Usage and Validation of Different Formats in Profiling Process". In *Poster Abstracts of the 26th*

International Conference on Information Technology Interfaces (ITI 2004) (Cavtat, Croatia, June). Pp39-40.

Borozan J. and Basch D. 2004b, "Profiling and Analysing Memory Objects in Java". In *Proceedings of 20th Annual UK Performance Engineering Workshop (UKPEW '04)*, (Bradford, UK, July). Pp28-137.

Cahoon B. 2004, "Java-Olden benchmarks". http://www.cs.utexas.edu/users/cahoon, [2004-9-13].

Chilimbi T., Jones R. and Zorn B. 2000, "Designing a Trace Format for Heap Allocation Events". In *Proceedings of International Symposium on Memory Management (ISMM '00)* (Minneapolis, USA, October) ACM Sigplan Notices 36(1). Pp35-49.

Dieckmann S. and Hoelzle U. 1999, "A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks". In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99)* (Lisbon, Portugal, June) LNCS 1628, Springer-Verlag, Berlin, Germany. Pp92–115.

van Groningen J.H.G. 1995, "Optimizing Mark-Scan Garbage Collection". In *The Journal of Functional and Logic Programming* 1995(2). Pp1-18.

Harris T.L. 2000, "Dynamic Adaptive Pre-Tenuring". In *Proceedings of International Symposium on Memory Management (ISMM '00)* (Minneapolis, USA, October) ACM Sigplan Notices 36(1). Pp127-136.

Harrison O. and Waldron J. 1999, *"Profiling Java Memory Demographics for Garbage Collection Purposes"*. Working Paper Series CA-2299, Dublin City University, Dublin, Ireland.

Hayes B. 1991, "Using Key Object Opportunism to Collect Old Objects". In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '91)* (Phoenix, USA, October) ACM Sigplan Notices 26(11). Pp33-46.

Henkel J. 2004, "Java-Colorado benchmarks". http://www-plan.cs.colorado.edu/henkel/projects/colorado_bench/, [2004-9-13].

Hertz M., Blackburn S.M., Moss J.E.B., McKinley K.S. and Stefanovic D. 2002, "Error-Free garbage Collection Traces: How to Cheat and Not Get Caught". In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (Marina Del Rey, USA, June). Pp140-151.

Hicks M.W., Moore J.T. and Nettles S.M. 1997, "The Measured Cost of Copying Garbage Collection Mechanisms". In *Proceedings of International Conference on Functional Programming (ICFP '97)* (Amsterdam, Netherlands, June). Pp292-305.

Hirzel M., Diwan A. and Hertz M. 2003, "Connectivity-Based Garbage Collection". In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)* (Anaheim, USA, October) ACM Sigplan Notices 38(11). Pp359-373.

Hirzel M., Henkel J., Diwan A. and Hind M. 2002, "Understanding the Connectivity of Heap Objects". In *Proceedings of International Symposium on Memory Management (ISMM '02)* (Berlin, Germany, June) Supplement to ACM Sigplan Notices 38(2). Pp143-156.

Hudson R.L. and Moss J.E.B. 1992, "Incremental Collection of Mature Objects". In *Proceedings of International Workshop on Memory Management (IWMM 92)* (St. Malo, France, September) LNCS 637, Springer-Verlag, Berlin, Germany. Pp388–403.

Jones R. and Lins R. 1996, *"Garbage Collection: Algorithms for Automatic Dynamic Memory Management"*. John Wiley & Sons, Chichester, UK.

Kaffe.org web site 2004, http://www.kaffe.org/, [2004-9-13].

Lindholm T. and Yellin F. 1999, *"The Java Virtual Machine Specification"*. Addison-Wesley, Reading, USA.

Stefanovic D., McKinley K. and Moss J. 1999, "Age-Based Garbage Collection". In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)* (Denver, USA, November) ACM Sigplan Notices 34(10). Pp370-381.

Stefanovic D. and Moss J.E.B. 1994, "Characterisation of Object Behaviour in Standard ML of New Jersey". In *Proceedings of ACM conference on LISP and functional programming (LFP '94)* (Orlando, USA, June). Pp43-54.

Stefanovic D., Moss J.E.B. and McKinley K.S. 1998, *"Oldest-First Garbage Collection"*. Technical Report UM-CS-1998-081, University of Massachusetts, Amherst, USA.

Venners B. 1999, *"Inside the Java Virtual Machine"*. McGraw-Hill, New York, USA.

Wilson P.R., Johnstone M.S., Neely M. and Boles D. 1995, "Dynamic Storage Allocation: A Survey

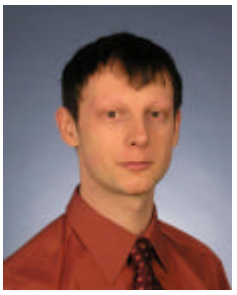
and Critical Review". In *Proceedings of International Workshop on Memory Management (IWMM 95)* (Kinross, UK, September) Lecture Notes in Computer Science 986, Springer-Verlag, Berlin, Heidelberg, Germany. Pp1–116.

Zorn B.G. 1989, *"Comparative Performance Evaluation of Garbage Collection Algorithms"*. Ph.D. thesis, University of California at Berkeley, Berkeley, USA.

Zorn B. and Grunwald D. 1992, *"Empirical Measurements of Six Allocation-Intensive C Programs"*. Technical Report CU-CS-604-92, University of Colorado, Boulder, USA.

Biographies

Danko Basch received B.Sc. in electrical engineering (1991), M.Sc. in computer science (1994), and Ph.D. also in computer science from the Faculty of Electrical Engineering and Computing (FER), University of Zagreb, Croatia. In 1992 he joined the FER (Department of Control and Computer Engineering in Automation) as a researcher. At present, he works at the same department as an assistant professor. His research interests include programming language design and implementation, garbage collection algorithms, and modelling and simulation.



Jurica Borozan received B.Sc. in computer science (1998) from the Faculty of Electrical Engineering and Computing (FER), University of Zagreb, Croatia. In 1998 he joined the FER (Department of Control and Computer Engineering in Automation) as a collaborator. He continued his professional career in the firms Eurolabores (2001) and OptimIT (2003). Currently he is finishing his M.Sc. study at FER. His research interests include profiling and performance evaluation of JVM, and operating systems.