GARBAGE COLLECTION FOR THE DELFT JAVA PROCESSOR

A. BERLEA* S. COTOFANA[#] I. ATHANASIU* J. GLOSSNER[#,$] S.VASSILIADIS[#]

* "Politehnica" University of Bucharest
Computer Science Department
Bucharest, Romania

[#]Delft University of Technology
Faculty of Electrical Engineering
Delft, The Netherlands

[$]IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
USA

ABSTRACT. The Delft Java Processor (DJP) is a hardware approach aiming at accelerating bytecode execution. One of the critical tasks in a JVM that could benefit from hardware support is garbage collection. To efficiently perform garbage collection in the DJP we have to better understand the general garbage collection related issues in real life situations. In this respect we provide in this paper a study of the dynamic allocation behavior of Java programs based on the SPECjvm98 benchmark suite. Age, size and type distribution of Java objects are presented and interpreted from the general point of view of garbage collection, together with other garbage collection related measurements. Thus we can identify the features of an efficient DJP dedicated garbage collector and propose supporting architectural extension(s) in the DJP. The study confirms the weak generational hypothesis according to which objects are most likely to die very young, suggesting that a collector with two generations could improve garbage collection performance with corresponding hardware support. To fully take advantage of the generational hypothesis efficient hardware-implemented write barriers are needed. Hardware support is also recommended for describing the dynamic layout of the stack (*stack pointer map*) for accurate garbage collection.

KEYWORDS: Java, Garbage Collection

1. INTRODUCTION

One of the main drawbacks of Java, and of languages based on virtual machines in general, is that program execution can not be as fast as the equivalent solution in a fully compiled language, as the execution process is interpretation based. In this direct interpretation approach a software program emulates the Java Virtual Machine (JVM). To improve performance and yet maintain the flexibility, a number of Java execution alternative techniques have been proposed. Just-in-time compilers, off-line compilers, native compilers are among them. Sun's *picoJava* implementation [1] for example directly executes the JVM Instruction Set and also provides support for garbage collection, instruction optimization, method invocation and synchronization. New technologies for improving Java programs performances covering the compiling process, garbage collection and synchronization, have recently been proposed in Sun's Hotspot JVM [2].

Another approach to hardware acceleration is *dynamic instruction translation*. To the best of our knowledge, the Delft-Java Processor [3] is the only processor to incorporate this feature.

In hardware assisted dynamic translation, Java Virtual Machine instructions are translated on the fly into the Delft-Java instruction set. The hardware requirements to perform this translation are not excessive when support for Java language constructs are incorporated into the processor's Instruction Set Architecture (ISA). The Java language provides processor architects with opportunities for exploiting Instruction Level Parallelism (ILP). Rather than requiring the processor to extract all ILP from a single executing thread, the Java language intrinsically supports programmer specification of parallelism through threads. In the Delft-Java architecture the goal is to extract maximal parallelism as defined by the Java language without burdening the programmer to specify any additional parallelism that is not inherent in the language constructs. At the highest level a programmer views the DJP as a Java Virtual Machine. In the Delft-Java approach, Java Virtual Machine execution is enabled by translating Java Virtual Machine bytecode into the Delft-Java RISC-based architecture. Special ISA support is provided for more complex Java Virtual Machine instructions. In addition to JVM execution, the Delft-Java architecture provides general support for C compilers and other operations that are required in general purpose processors. Architectural support for Multimedia SIMD and Digital Signal Processing (DSP) is also incorporated into the architecture.

The intended capabilities of the Delft-Java processor also include architectural support for synchronization for multithreaded organizations, garbage collection, array bounds checking, and vector operations [4].

Hardware support for the garbage collection is very important for the performance of the DJP when used as a JVM. In this work we analyze the general features of dynamic memory allocation of Java programs aiming at finding the appropriate garbage collection scheme in a JVM. Establishing a general pattern of dynamic memory allocation for Java programs is important, as a garbage collector can be best tailored when this pattern is known. Based on the observations made, actions the DJP could take to support garbage collection are suggested. Programs from the SPECjvm98 benchmark suite released by the System Performance Evaluation Corporation [5] are used as relevant, real-world Java programs for our study. These benchmarks are considered relevant for Java applications in general in the computer world as they provide a standardized test suite intended to measure and test performance of Java Virtual Machines.

Choosing a garbage collection scheme for a language implementation is a task best accomplished when the exact dynamic allocation behavior of applications written in that language is known. As it is not possible to have a separate garbage collector implementation for each application, if we aim at best average performance we should try to find the general allocation pattern of the applications written in the language for which the garbage collection is targeted. Instrumentation of memory allocation behavior like object sizes and lifetimes will help the implementer choose the appropriate garbage collection algorithm.

A number of studies have addressed the allocation behavior problem for several garbage collected languages like ML, Lisp, Smalltalk as well as for C/C++ as indicated in [6]. However as the allocation behavior is expected to be significantly affected by the programming language the applications are written in, no programming language independent general behavioral conclusions can be based on these studies. Therefore separate study of the allocation behavior is needed for the Java programs.

A first in-depth study of the memory usage in Java programs has been made by Dieckman and Hölzle [6]. Age, size and type distribution and the overhead of object alignment have been measured. Time-related measurements, like object lifetimes, are expressed in terms of Mbytes allocated. This metric is justified as from the point of view of garbage collection the amount of memory allocated directly correlates with the amount of the work that the memory allocator and the garbage collector have to execute. Object lifetimes are determined by repeatedly forcing garbage collection cycles. Thus the "time" accuracy is limited by the arbitrary intervals at which garbage collection is forced. Also, the application should allocate constantly and at about the same rate if we are to receive precise results on a "real-time" scale. Allocation of big objects requires special treatment, or otherwise objects could artificially become old due to an allocation of a very large object.

Also using the SPECjvm98 benchmarks we studied the dynamic allocation pattern of Java objects aiming at providing useful information for garbage collection implementation decisions. Our study gives an insight view of time-related aspects of the allocation behavior of Java programs by using an approach in which direct time measurements are possible, as opposed to the above mentioned approach. Time measurements are made on the real-time scale avoiding accuracy problems imposed by a two-phase simulation environment. Measurements are made in just one phase, as the benchmarks execute.

We present age, size and type distribution of the Java objects. The age distribution can decide whether the generational hypothesis holds for Java programs and if so how many generations are best to consider when garbage collecting. The size measurements can show whether a compacting algorithm is justified for the case objects prove to be very heterogeneous in size or a non-compacting algorithm could perform similar in case most of the objects are similar sized. The analysis of object type distribution could show whether it makes sense to treat some types of objects differently because for example they occur very often.

In an incremental approach the necessary coordination of the garbage collection process with the running program involves the use of either a *read* or *write-barrier*. The present study evaluates the number of read respectively write-barriers the running program would encounter if either of the above synchronization schemes is to be used. The number of read-barriers is obviously larger that the number of write-barriers (because, as it will be indicated latter in the paper, anytime a write barrier is needed a read barrier is also necessary) but information on the ratio between these two numbers in practice is quite important as it provide guidance in choosing the least expensive, from the hardware implementation point of view, barrier based garbage collection solution.

Furthermore we counted the number of dead objects a pure reference counting collector fails to reclaim. This can show whether an adapted reference counting collector could be a solution for certain Java applications.

The structure of the remainder as this paper is the following. Section 2 presents the experimental setup used for the measurements. Section 3 presents the experimental results and their interpretation. In section 4 the needed support for garbage collection in the DJP is suggested and section 5 presents the conclusions of the study.

2. EXPERIMENTAL SETUP

Because of the large number of API's required to run significant benchmarks, a complete JVM is required to fully characterize the effects of garbage collection. The Delft Java JVM does not currently support the full range of APIs required for this detailed analysis. As a result, the Kaffe JVM was chosen for experimental analysis. Kaffe is a C-written open source JVM distributed under the GNU Public License. Access to the source code was necessary in order to implement the experimental environment and to experiment with different garbage collection variants. The Kaffe source code has been widely ported to several platforms (i386, Sparc, MIPS) and operating systems (Linux, Solaris, Windows).

The analysis presented in [6] uses the SPECjvm98 benchmark suite in an experimental setup consisting of two independent phases. First a tracer in form of an instrumented version of Sun's JDK 1.1.5 VM produces a trace file while executing the benchmark application. Second, a Java-written simulator reads the trace and simulates allocation, pointer assignments and garbage collection while computing statistics on Java objects. Currently, the simulator forces a full collection after every 50 Kbytes of allocation. The disadvantage is that we can not know exactly when objects die. We can only know they died within a certain interval of 50 Kilobytes allocated.

The approach used in [6] has the disadvantage that garbage is not reclaimed immediately, at the moment it is produced. This is also the disadvantage of all tracing collectors. Instead of being instantly collected, garbage is reclaimed only subsequently at the moment the garbage collection cycle occurs. The only type of collector succeeding in reclaiming garbage objects at the very moment they become garbage is a reference counting collector.

Our idea was to exploit this quality of reference counting collectors by creating a special runtime environment containing such a collector in order to instrument Java. Hence, we modified the Kaffe JVM to include a reference counting garbage collector and ran the benchmarks on the modified JVM. The advantage of this approach is that garbage objects are reclaimed immediately after they become useless. Time related statistics for such reclaimed objects is therefore as accurate as possible. Consequently the resolution of the object lifetime related measurements are improved as compared to the above-mentioned study.

The reference counting implementation in the Kaffe JVM aimed at instrumenting the Java language using the JVMspec98 suite of benchmarks. We also let the original mark and sweep conservative collector of Kaffe run. It is this that actually does the garbage collection. The reference counting collector in turn realize when most of the objects become inaccessible and collect the information which is then used for the statistics on the Java programs.

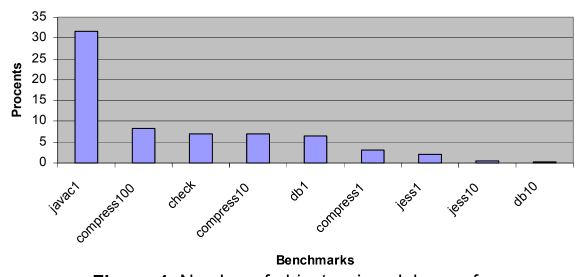

Figure 1. Number of objects missed by a reference counting garbage collector

Possible sources of errors are objects that the reference counting collector fails to reclaim, as for example garbage objects in cyclic structures. Our experiments took into account these objects also, by using besides the reference counting for accurate lifetime analysis of the Java objects the original mark and sweep collector of Kaffe. Thus we were able to see to what extent some of our measurements have been affected by the above-mentioned drawback. As shown in Figure 1, the number of objects missed by the reference counting garbage collector is in general not important, therefore not significantly influencing the accuracy of our measurements.

Reference counting collectors in general have the drawback that they incur a great deal of overhead. However the overhead is uniformly distributed along the effective computation time. In our experiment this was not important because we were interested in relative time measurements, as for example relative lifetimes of objects. This relative time related measurements are not affected by the overhead, as this is uniform distributed along the execution time of the application.

2.1. THE REFERENCE COUNTING COLLECTOR

We briefly present next the reference counting garbage collector implementation used in the experimental environment. As its name already suggests a reference counting collector keeps track of the number of references to every object and when this number drops to zero, it knows the corresponding object is dead and can be safely reclaimed.

Two things are mainly required in order to implement a reference counting collector: a reference counting field within the Java Objects and a mechanism for updating reference counters by incrementing, or decrementing the reference counting field every time a new reference to an object is created respectively destroyed.

We added the reference counting field in each object header and modified the interpreter to keep track of each new reference that appears/disappears and to update the corresponding reference counting field.

All JVM instructions handling references needed modifications as they request reference-counting fields updating. The number of modified instructions in the JVM used for instrumenting Java programs was 54 out of a total of 160 instructions opcodes recognized by the Java interpreter.

A reference counting implementation need to be very accurate in the sense that it can not afford not to update the reference count of an object when a new reference appears or when a reference is destroyed, nor to wrongly update a reference counting when it should not. For correctly updating the reference counters it was necessary to tell reference from non-reference locations among the local variables and in the operand stack in each Java stackframe. Therefore we used a structure parallel to the stack frames (we will call it *stack map*) to record the information related to which locations in the local variables or operand stack are references and which are not. This dynamic structure needs updating as the JVM interprets the bytecode. Local variables need this bookkeeping because they may hold at different points during the execution of their function values which are either references or non-references or at certain moments undefined if they are for example uninitialized. The location types in the operand stack also dynamically change with the program interpreting and thus updating the stack map is necessary for them too.

We present below as an example the modifications in the ALOAD JVM instruction. ALOAD is the instruction used within the JVM to retrieve an object reference from a local variable and put it on the top of the operand stack. In this case a new reference to an object appears on the stack. In order to do reference counting collection the reference count of the object to which a reference is put on the stack needs to be incremented. The top of the stack is also to be marked as containing a reference since this can be further involved in other reference counting updates (for example if this location on the stack will be further stored in another local variable).

The same bookkeeping of a *stack map* is needed if an accurate garbage collector is to be implemented. The collector must be always able to exactly distinguish references from non-references values. Tracing garbage collectors determine reachability of objects, i.e., object liveness, from some set of *roots*. In Java the Java stack forms one component of the rootset. The *stack map* must be available, describing which locations on the Java stack contain references.

The reference counting garbage collection implementation enabled us to collect easily and accurately information needed for our instrumentation of Java programs.

For lifetime measurements a time-stamp was stored in the object headers recording the object creation time. When the reference counting collector finds an object dead, in order to find the object's lifetime, it only needs to subtract from the current time the object creation time, found in the object's header.

A reference counting collector fails to reclaim garbage objects in cyclic structures. For these objects we let the original Kaffe mark and sweep collector run. In order to know by which of the collectors was an object reclaimed a field was also added in the object header used as a *collector signature*. Thus we were able to do statistics related to the number of objects possibly missed by a reference counting collector (already presented in the introductory section).

A problem to be faced when implementing an incremental tracing collector is that while the collector is tracing out the graph of reachable data structures, the graph may change as the running program may *mutate* the graph while the collector "isn't looking" [7]. Incremental marking traversals must take into account changes to the reachability graph, made by the mutator during the collector's traversal. There are two basic approaches to coordinating the collector with the mutator involving read or write barriers respectively.

Read barriers are needed in order to assure that the mutator (or the user program, which is a kind of mutator for the concurrent garbage collector) always sees the actualized version of an object in case a garbage collector is on the run concurrently with the mutator actions.

Write barriers are necessary when the concurrent collector needs to be assured that it catches all the writes to objects already traversed by him, in order to possibly rescan them to find live objects which would otherwise be falsely declared dead.

A read barrier is needed every time a reference to an object is accessed, as the pointed object is certainly live and thus the reference should eventually become updated to the new version of the object. A write barrier is needed when a reference to an object is used in order to modify the referenced object.

Modifications of the Kaffe JVM were also necessary in order to create the necessary instrumenting run-time environment for counting the number of read-, respectively write-barriers a garbage collector would encounter if an incremental garbage collection approach using read-, respectively write-barriers is to be taken.

As an example we bellow present the ASTORE JVM instruction case. The object whose reference is stored in a local variable by ASTORE, needs to be read-protected in the incremental read-barriers approach, as in this approach, when an object is used, i.e., read, it is known to be live and therefore special action is to be taken. Thus the number of read-barriers an incremental garbage collector would encounter in the read-barrier approach needs to be incremented. Also, the number of write-barriers an incremental garbage collector would encounter if it uses the write-barrier approach needs to be incremented. A write-barrier would be needed here because the interpreter writes in a zone, the local variables, which might have been already scanned for pointers, and in the write-barrier incremental approach special action is to be taken in such a situation.

3. EXPERIMENTAL RESULTS

Using the modified Kaffe JVM and the jvmSPEC98 set of benchmarks, representative for Java programs, we studied the following Java objects related features.

3.1 AGE DISTRIBUTION

In order to have a general view over the object liveness feature of all the tested benchmarks we expressed object lifetimes as percents representing their absolute lifetime reported to the absolute lifetime of the oldest object in the benchmark. This means that for example an object whose lifetime was 50% has lived half the life of the oldest object in the same benchmark experiment. The number of objects having a certain lifetime was also expressed in percents out of the total number of objects created by the respective benchmark.

The general observed pattern in all the benchmarks is that the majority of objects die very *small aged*. Figure 2 depicts the number of objects not surviving above a certain age in the *javac* benchmark. For a better observation Figure 2 focuses on very small ages. As shown in the distribution of the objects according to their

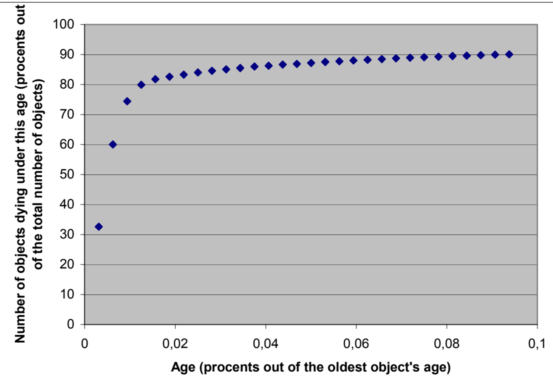

Figure 2. Age histogram (magnifying glass) benchmark = javac

lifetime for the *javac* benchmark, more than 90% of the objects are not surviving above the age of 0.1%. Beside the ephemeral 90% objects (the young generation), the remaining 10% of the objects uniformly cover ages from 0.1% to 100% (the old generation). We have only these two generations as no further clear distinction among the old generation can be made.

All the tested benchmarks presented similar results. In the *jess* benchmark approximately 80% of the objects die at ages of under 20%. The results of the *db* benchmark show that more than 40% of objects die at ages under 0.1%.

3.2 SIZE DISTRIBUTION

We also used relative measure for object sizes in order to compare the results obtained with different benchmarks. The size is measured as percents from the size of the biggest allocated object in the benchmark.

A common tendency to the benchmarks is that the vast majority of objects have very small sizes compared to a very small fraction of big objects.

The size histogram for the *javac* benchmark program is depicted in Figure 3. As indicated in the chart more than 90% of the objects have sizes under 0.4%.

Discontinuities in the graph depicted in Figure 3 are to be interpreted as separations of objects in classes according to their sizes. The plateaus we can observe in the graph represent the classes of objects according to their size. The representative of each class can be considered the object size corresponding to the beginning of each plateau. Objects in an object size class are as size

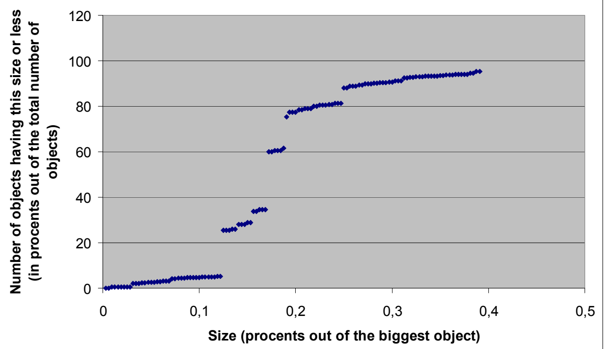

Figure 3. Size histogram (magnifying glass)

homogeneous as flat as the corresponding plateau is.

With the above "key" we can interpret the Figure 3 as indicating mainly seven size classes of objects in the *javac* benchmark occupying sizes between 0 and 0.4%. However, as the stairs are not perfectly horizontal, there are still size differences within a size class, as important as steep the stair is. Anyway there is no predominant object size among the objects.

The size histogram for the *jess* benchmark program suggests that more than 80% of the objects are very small sized. 90% of the objects have sizes under 1.4%. The similarly identified number of size classes is this time nine, but again there is no predominant size among the small sized objects.

The size distribution for the *db* benchmark program showed that almost all the objects are very small sized, having sizes of under 0.1%.

According to the presented results the number of object size classes is not very important. The size pattern of objects significantly depends of applications. This means that, even if a non-compacting collector will avoid fragmentation if it happens to allocate objects belonging to the same object size class together, compactness can not be guaranteed if a non-compacting garbage collector is used. However, possible techniques are conceivable in which objects of same dynamically determined size class are allocated together alleviating thus fragmentation.

3.3 TYPE DISTRIBUTION

A JVM dispose over four instructions for allocating Java objects: NEW for creating “normal” Java objects (*objects*), NEWARRAY for creating an array of numbers or Booleans (*arrays*), ANEWARRAY to allocate an array of objects (*refarrays*), MULTIANEARRAY to allocate a multi-dimensional array (*multiarrays*)

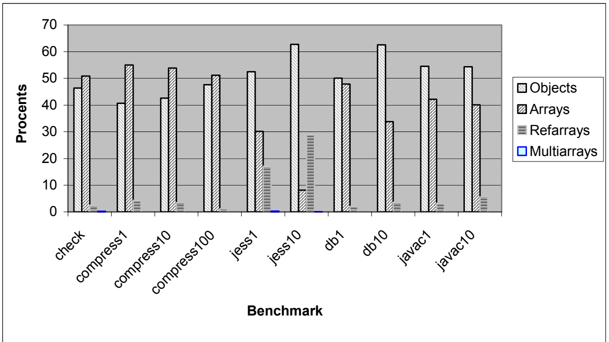

Figure 4. Type of Java objects

Accordingly, we measured the type repartition for the tested benchmarks. The results are presented in Figure 4.

We can observe that most Java objects are normal objects and arrays. Together they represent approximately 90% of all allocated objects.

We also studied the repartition of objects according to their Java classes. Object classes in the *javac* benchmark are depicted in Figure 5. Similar results were also obtained with the other benchmarks. As a general feature relatively many objects have the class java/lang/String. For some benchmarks relatively many objects have the

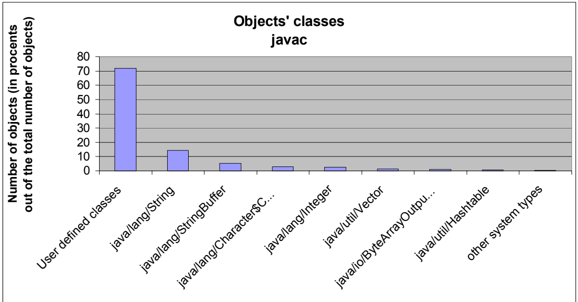

Figure 5. Object classes (javac benchmark)

class java/lang/Integer. Most of the objects have user-defined classes.

In the case of the *javac* benchmark these were 72% of all the objects. The number of objects having System classes (predefined in the Java language, as opposed to the user-defined classes) summed thus approximately 28% of the total number of objects. String object were 14%, StringBuffer 5% and so forth as one can see in Figure 4.

In the *jess* benchmark 70% of the objects have user-defined classes. Among the System class objects the most frequent are Integer objects with 19% and String and String Buffer having together 9%.

The results suggest that there are objects appearing relatively often. The type of the objects appearing frequent is application dependent. However, as these results show, String objects are in general relatively frequent. It would make thus sense that they are therefore considered apart from other objects.

3.4 NUMBER OF WRITE OR READ BARRIERS

In the case of the Java programs the found proportion of read respectively write barriers possibly needed was of 65%, respectively 35%. The absolute numbers for *javac* benchmark was around 45 million barriers that the interpreting of the program would encounter in the read approach as compared with about 25 million write barriers which would be needed for the same program in the write-barrier approach.

Both approaches are expensive in terms of time consumed for the barriers if applied on conventional hardware. Read-barriers are even more expensive since pointer reads are much more often as our results also suggest. The choice of a read or write barrier scheme is likely to be made on the basis of the available hardware [7].

4. INTERPRETING THE RESULTS

The experiments presented above also gave us the opportunity to identify some critical garbage collection tasks. With corresponding hardware support their execution is likely to significantly improve. We present bellow some of our observations and the suggestions for the DJP.

Maintaining the *stack map,* needed for the reference counting collection but also for accurate garbage collection implementations introduces both performance and complexity overhead as each time a reference appears/disappears on/from the stack, the stack map must be updated.

The overhead of maintaining the stack map could be drastically reduced if it would be hardware supported by the automatically updating of the stack map structure.

Our study of the allocation behavior showed that Java could take advantage of a generational garbage collection approach. But in order to fully take advantage of the generational approach write barriers are needed, as we must trap all the pointer writes in order to take special action in case an old-to-young intergenerational pointer is created. Moreover write barriers are also inevitable for an incremental garbage collection approach. The overhead introduced by soft write-barriers could be significantly diminished if write barriers are in part/totally implemented in hardware.

The Java dynamic allocation behavior study suggested that in general a reference counting collector only fails to reclaim a relatively small data percent. One of the advantages of reference counting garbage collection is that memory management overheads are distributed throughout the computation [9]. The DJP could use a reference counting collection scheme, in order to provide smooth response time and have a second tracing collector run from time to time in order to reclaim the garbage that the reference counting collector fails to reclaim. If the amount of garbage to be reclaimed by the tracing collector is not important this scheme provide smooth overall response-time. A reference counting collector basically needs the maintaining of the same *stack map* necessary for accurate tracing collector implementations. Beside marking/unmarking the corresponding bit in the *stack map* when a reference appears/disappears in the stack, the hardware support should then also increment/decrement the corresponding reference counting field, and take the needed reclaiming action in case the reference counting drops to zero.

5. CONCLUSIONS

We considered a number of issues associated with garbage collection in Java in general, and in the Delft Java Processor in particular. Our overall investigations and achievements can be summarized by the following.

We studied the dynamic allocation pattern of Java objects, based on a set of standardized benchmarks. The results confirmed the generational hypothesis, according to which most objects die very young. This suggested that a *generational* collection approach for Java is worth. The age distribution of Java objects suggested that the number of generations required is not greater than two, since there are mainly only two clearly separated generations. The size analysis revealed no specific size pattern for the Java object in general. However only a small fraction of the objects is much more big-sized than the vast majority of the others. We suggest a possible improvement to the simple generational garbage collection by trying to identify this few large objects in order to avoid to copy them repeatedly if a copy collector is used. The type analysis suggested that String objects could be treated specially as they are expected to appear quite often in Java programs.

An accurate collector implementation for Java requires that at the moments a garbage collection cycle can occur a stack map is available in order to tell the collector which locations on the Java stack are pointers. The maintaining of a stack map structure incurs much overhead in the interpreting JVM. Therefore a significant time-performance improvement is to be expected if this could be partially/totally hardware implemented.

A generational garbage collector for Java can take advantage of the Java objects tendency to die relatively very young. However in order to take fully advantage of the generational hypothesis which confirms in Java, write-barriers necessary for generational collections need to be efficiently implemented. Much performance improvement is therefore also to be expected if they could be hardware supported.

REFERENCES

[1] Sun Microelectornics. PicoJava Microprocessor Core Architecture. Technical Report WPR-0015-01, Sun Microsystems, Mountain View, California, November 1996. Available from http://www.sun.com/sparc/whitepapers/wpr-0015-01

[2] http://java.sun.com/products/hotspot/whitepaper.html

[3] John Glossner and Stamatis Vassiliadis, The Delft Java Engine: An Introduction, Third International Euro-Par Conference (Euro-Par '97) Parallel Processing, Lecture Notes In Computer Science, pages 766-770, Passau, Germany, August 1997, Springer-Verlag

[4] John Glossner, DELFT-JAVA, a multi-threaded Java Accelerator. Ph.D. Dissertation draft

[5] http://www.spec.org/osg/jvm98/

[6] Sylvia Dieckman and Urs Hölzle, A Study of the Allocation Behaviour of the SPECjvm98 Java Benchmarks, ECOOP'99, the 13th European Conference on Object-Oriented Programming, Lisbon, Portugal, June 1999

[7] Paul Wilson, Uniprocessor Garbage Collection techniques, 1996

[8] Ole Agesen and David Detlefs, Finding References in Java Stacks, OOPSLA'97 Workshop on Garbage Collection and Memory Management, pages 766-770,

[9] Richard Jones and Rafael Lins, Garbage Collection, Algorithms for Automatic Dynamic Memory Management, JOHN WILEY & SONS, NY 10158-0012, USA, 1996