Constant-Time Root Scanning for Deterministic Garbage Collection

Fridtjof Siebert

Institut für Programmstrukturen and Datenorganisation (IPD)
Universität Karlsruhe
Am Fasanengarten 5
76128 Karlsruhe, Germany
siebert@jamaica-systems.de

Abstract. Root scanning is the task of identifying references to heap objects that are stored outside of the heap itself, in global and local variables and on the execution stack. Root scanning is particularly difficult within an incremental garbage collector that needs to be deterministic or give hard real-time guarantees. Here, a method that allows exact root scanning is presented. The basic idea is to ensure that copies of all root references exist on the heap whenever the garbage collector might become active. This approach reduces the root scanning phase of the garbage collection cycle to an efficient constant-time operation. A Java virtual machine and a static Java bytecode compiler that use this technique have been implemented and analysed using the SPECjvm98 benchmark suite. This Java implementation allows for deterministic memory management as needed in real-time systems that is difficult to achieve with traditional methods to perform root scanning.

1 Introduction

With the rising popularity of modern object-oriented programming languages like Java [1] garbage collection has finally been accepted as a means for memory management, even though it typically brings a high degree of indeterminism to the execution environment. Nevertheless, Java is more and more promoted as a development tool even for real-time critical systems that require deterministic execution [2, 3].

The indeterminism of garbage collection has two aspects: the garbage collector causes pauses that are hard to predict while the automatic detection of free memory makes predictions on the memory demand of an application difficult.

To avoid long unpredictable pauses caused by the collector, incremental or concurrent garbage collection techniques are employed [4, 5, 6, 7]. A garbage collection cycle can be performed in small increments while the main application executes. Each collection cycle starts with the root scanning phase, during which all references outside of the heap are detected and the referenced objects on the heap are marked. This includes references on the local stacks of all threads, in processor registers and in global variables outside of the heap. After the root scan-

ning phase, the collector continues by recursively marking objects referenced by objects that have already been marked.

The root scanning of the stack of one thread typically requires stopping the corresponding thread during the time the references are scanned. This can cause pause times that are too long for time-critical real-time applications that require short response times.

In addition to these pause times, there is often not enough information available on where references are located and which references are live. Conservative techniques can be employed in this case. Conservative root scanning treats all bit patterns that happen to represent legal object addresses on the heap as if they were actual pointers to heap objects. This usually works well since it is unlikely that a random integer or float value on the stack is a legal object address, but it makes the memory management completely unpredictable. Another difficulty in this approach are dangling references that reference objects on the heap, but that represent dead variables that are no more used by the application.

A deterministic garbage collector that can be used in safety-critical systems requires exact information on the roots. Additionally, an incremental garbage collector that is to be employed in hard real-time systems has to guarantee that the pause times for root scanning are bounded and very short.

2 Related Work

Little work has been published in the area of root scanning for real-time garbage collection, but several papers describe mechanisms for conservative and exact root scanning.

The application of conservative root scanning in a *mostly copying* compacting garbage collector has been presented by Barlett [8]. This collector refrains from moving an object and changing its addresses if the objects might be referenced from a conservatively scanned root reference.

Boehm presents a practical implementation of a conservative garbage collector for C using conservative root scanning [9] and describes methods that reduce the likelihood for pointer misidentification during conservative root scanning [10]. Heap addresses that are found to be referenced by misidentified pointers are *blacklisted*, the referenced memory is not used for allocation such that future references to these addresses found during conservative root scanning will not cause the retention of memory.

Goldberg [11] describes a method for tag-free collection based on an idea by Appel [12]. Appel's approach uses the return address stored in a function's activation frame to determine the function an activation frame belongs to. When this function is known, the types of the variables within the activation frame can be determined. Goldberg makes use of the fact that in a single-threaded environment, root scanning might only occur at calls or memory allocations. Additional information can be associated with each call point that describes a function's frame at this point, taking into account the life span of local variables. On some architectures, this can be implemented without a direct runtime cost. Goldberg also proposes an extension of his approach for multi-threaded environments: If

garbage collection is needed, all running threads should be stopped at the next allocation or procedure call.

Diwan et. al. [13] propose the use of tables generated by an optimizing compiler that describe live pointers and values derived from pointers and that allow the garbage collector not only to find references but as well to update the references to compact the heap. For multi-threaded environments, an approach similar to Goldberg's is suggested: when GC is triggered, suspended threads are resumed such that they reach their next *GC-point*. The compiler ensures that the time needed to reach the next *GC-point* is bounded.

Agesen et. al. [14] compare exact root scanning with the conservative approach that is typically used in Java implementations. The average reduction in heap size they achieved when using exact root scanning is 11% for a suite of 19 benchmarks they analysed, while the effect was more dramatic on a few tests.

Stichnoth et. al [15] show that it is even feasible to avoid the need for *GC-points* and instead provide exact information on live root references for every machine instruction generated by a compiler. This avoids the need to resume suspended threads when garbage collection is triggered. *GC maps* are needed to hold the exact information, the space required for theses maps is about 20% of the size of the generated code.

The disadvantage of all these approaches is that they do not provide means to avoid the pause time due to root scanning. Dubé, Feeley and Serrano propose to put a tight limit on the size of the root set such that the root scanning time is limited [7]. Since this will not be practical for all systems, they propose to scan roots incrementally, which would require the use of a *write barrier* for all root references and impose a high runtime cost.

Christopher presented an interesting approach that is based on reference counting [16] and that avoids the need of scanning root references that are stored outside of the heap [17]. The idea is that all objects with a reference count that is higher than the number of references from other heap objects to this object must be reachable from a root reference outside the heap. All the objects reachable from such a root reference are then used as the initially marked set in the marking phase of the garbage collector. The disadvantage is the high runtime overhead that is required to keep the reference counts accurate: any assignment between local reference variables needs to ensure correct adjustment of the reference counts.

3 The Garbage Collector

The technique for root scanning presented here is largely independent of the actual garbage collection algorithm. The explanation is therefore limited to the description of the mechanisms relevant for root scanning. We have implemented an incremental mark and sweep collector, but the technique might as well be applied to different incremental garbage collection techniques that have been presented in earlier publications [4, 5, 6, 7]. For compacting or copying techniques, additional difficulties might arise when updating of root references is required. The use of handles would be a solution here.

4 Synchronization Points

An important prerequisite for the root scanning technique presented in the next section is to limit thread switches and garbage collection activity to certain points during the execution of the application. The idea has been presented earlier [18] and similar mechanisms have been employed earlier [19]. When synchronization points are used, thread switching at arbitrary points during the execution is prohibited. Instead, thread scheduling can occur only at *synchronization points* that are automatically inserted in the code by the compiler or virtual machine. The implementation has to guarantee to insert the code required at a *synchronization point* frequently enough to ensure short thread pre-emption delays. Since thread switches are restricted to *synchronization points*, root scanning might also only occur at these points. A different thread might run and cause garbage collection activity only if all other threads are stopped at *synchronization points*.

A possible implementation of a *synchronization point* is shown in **Listing 1**. A global semaphore is used to ensure that only one thread is running at any time. This semaphore is released and reacquired to allow a different thread to become active. The code is executed conditionally to avoid the overhead whenever a thread switch is not needed. The thread scheduler has to set the global flag *synchronization_required* whenever a thread switch is needed.

```
...
if (synchronization_required == true) {
   ...
   /* allow thread switch */
   V(global_semaphore);
   P(global_semaphore);
   ...
}
...
```

Listing 1. Code for conditional *synchronization point*.

The use of *synchronization points* has several important effects on the implementation:

1. The invariants required by an incremental garbage collector do not have to hold in between two *synchronization points*. It is sufficient to restore the invariant by the time the next *synchronization point* is reached. This gives freedom to optimizing compilers allowing them to modify the code in between *synchronization points*, e.g., the compiler might have *write barrier* code take part in instruction scheduling (*write barrier* code is code that has to be executed when references are modified on the heap such that an incremental garbage collector takes the modification into account).

2. Between two *synchronization points,* no locks are required to modify the memory graph or global data used for memory management. In the context of au-

tomatic memory management this affects *write barrier* code and allocation of objects that can be performed without the need for locks on the accessed global data structures since all other threads are halted at *synchronization points* and are guaranteed not to modify this data at the same time.

3. Exact reference information is required only at *synchronization points* since root scanning can only take place here.

5 Constant Time Root Scanning

The idea presented here is to ensure that all root references that exist have to be present on the heap as well whenever the garbage collector might become active. This means that the compiler has to generate additional code to store references that are used locally on the program stack or in processor registers to a separate *root array* on the heap. Each thread in such a system has its own private *root array* on the heap for this purpose.

All references that have a life span during which garbage collection might become active need to be copied to the *root array*. Additionally, whenever such a reference that has been stored is not used anymore, the copy on the heap has to be removed to ensure that the referenced object can be reclaimed when it becomes garbage. The compiler allocates a slot in the current thread's *root array* for each reference that needs to be copied to the heap. The *root array* might be seen as a separate stack for references.

To ensure that the garbage collector is able to find all root references that have been copied to the *root arrays*, it is sufficient to have a single *global root pointer* that refers to a list of all *root arrays*.

The root scanning phase at the beginning of a garbage collection cycle can be reduced to marking a single object: the object referenced by the *global root pointer*. Since all *root arrays* and all references stored in the *root arrays* are reachable from this *global root pointer*, the garbage collector will eventually traverse all *root arrays* and all the objects reachable from the root variables that have been copied to these arrays. Since all live references have been stored in the *root arrays*, all local references will be found by the garbage collector.

The effect of this approach is that the *root scanning* phase becomes part of the garbage collector's mark phase: While the collector incrementally traverses the objects on the heap to find all reachable memory it incrementally traverses all root references that have been stored in the *root arrays*.

To maintain the incremental garbage collector's invariant, it is important to use the required *write barrier* code when local references are stored into *root arrays*.

Another minor problem are root references in global (static) variables. A simple solution is to store all static variables on the heap in a way that they are also reachable from the *global root pointer*. If this is not possible, a possible solution could be to always keep two copies of all static variables, one copy at the original location and another one in a structure on the heap that is reachable from the *global root reference*.

6 Saving References in Root Arrays

Saving local references in *root arrays* on the heap is performance critical for the implementation: Operations on processor registers and local variables in the stack frame are very frequent and typically cheap. Storing these references in the *root arrays* and executing the *write barrier* typically requires several memory accesses and conditional branches. To achieve good performance the number of references that are saved to the heap must be as small as possible.

The garbage collector might become active at any *synchronization point*. From the perspective of a single method, the collector might as well become active at any call, since the called method might contain a *synchronization point*. It is therefore necessary to save all local references whose life span contains a *synchronization point* or a call point. For simplicity of terms, *synchronization points* and call points will both be referred to as *GC-points* in the following text.

It is not clear when the best time to save a reference would be. There are two obvious possibilities:

1. *Late saving*: All references that remain live after a *GC-point* are saved directly before the *GC-point*. The entry of the *root array* that was used to save the reference will then be cleared right after the *GC-point*.

2. *Early saving*: Any reference with a life span that stretches over one or several *GC-points* is saved at its definition. The saved reference is cleared at the end of the life span, after the last use of the reference. Note that a life span might have several definitions and several ends, so code to save or clear the reference will have to be inserted at all definitions and ends, respectively.

It is not obvious which of these two strategies will cause less overhead. *Early saving* might cause too many references to be saved, since the *GC-points* within the life span might never be reached, e.g., if they are executed within a conditional statement. *Late saving* might avoid this problem, but it might save and release the same reference unnecessarily often if its life span contains several *GC-points* that are actually reached during execution.

A third possibility analysed here is a mixture between *early* and *late saving*, which can be done as follows:

3. *Mixed*: Since *synchronization points* that use a conditional statement like the one shown in **Listing 1** are executed only when a thread switch is actually needed, it makes sense to use *late saving* for life spans that only contain *synchronization points* but no call points. In this case, the saving of the reference in the *root array* before the thread switch and the clearing of the entry in the *root array* when execution of the current thread resumes can be done conditionally within the *synchronization point,* as shown in **Listing 2**. Whenever a life span contains call points, *early saving* is used since it is likely that one of the call points is actually executed and requires saving of the variable.

7 The Jamaica Virtual Machine

Jamaica is a new implementation of a Java virtual machine and a static Java compiler that provides deterministic hard real-time garbage collection. Root scanning

```
...
ref := ...;
...
if (synchronization_required == true) {
   Save_To_Root_Array(ref);

   /* allow thread switch */
   V(global_semaphore);
   P(global_semaphore);

   Clear_From_Root_Array(ref);
}
...
use(ref);
...
```

Listing 2. Conditionally saving a reference *ref* at a synchronization point

is done as just described: The compiler generates additional code to save live references that survive *GC-points* into *root arrays* on the heap. There is only a single *global root pointer*, and the garbage collector's root scanning phase is reduced to marking the single object that is referenced by the *global root pointer*.

Other aspects of the implementation's garbage collector have been described in more detail in earlier publications [20, 21]: An object layout that uses fixed size blocks is used to avoid fragmentation. The garbage collection algorithm itself does not know about Java objects, it works on single fixed size blocks. It is a simple Dijkstra et. al. style incremental mark and sweep collector [4]. A reference-bit-vector is used to indicate for each word on the heap if it is a reference, and a colour-vector with one word per block is used to hold the marking information. These vectors exist in parallel to the array of fixed size blocks.

The garbage collector is activated whenever an allocation is performed. The amount of garbage collection work is determined dynamically as a function of the amount of free memory in a way that sufficient garbage collection progress can be guaranteed while a worst-case execution time of an allocation can be determined for any application with limited memory requirements [22]. This approach requires means to measure allocation and garbage collection work. The use of fixed size blocks gives natural units here: the allocation of one block is a unit of allocation while the marking or sweeping of an allocated block are units of garbage collection work.

The static compiler for the Jamaica virtual machine is integrated in the Jamaica builder utility. This utility is capable of building a stand-alone application out of a set of Java class files and the Jamaica virtual machine. The builder optionally does smart linking and compilation of the Java application into C code. The compiler performs several optimizations similar to those described in [23]. The generated C code is then translated into machine code by a C compiler such as *gcc*.

8 Write Barrier in Jamaica

The purpose of a *write barrier* is to ensure that all reachable objects are found by the incremental garbage collector even though the memory graph is changed by the application while it is traversed by the collector. Three colours are typically used to represent the state of objects during a garbage collection cycle: *white, grey* and *black*. *White* objects have not been reached by the traversal yet. *Grey* objects are known to be reachable from root references, but the objects that are directly referenced by a *grey* objects might not have been marked yet, i.e., they might still be *white*. The incremental garbage collector scans *grey* objects and marks all *white* objects that are reachable from the *grey* object *grey* as well. The scanned object is then marked *black*.

The *write barrier* has to ensure the invariant that no *black* object refers to a *white* object (alternative invariants and *write barriers* have been presented by Pirinen [24]). One way to ensure the invariant is to mark a *white* object *grey* whenever a reference to the *white* object is stored in an object. The marking is typically done using a few mark-bits that represent the colour of an object. The *write barrier* code then sets these bits to the value that represents *grey* whenever a reference to a *white* object is written into a heap object. *Card marking* has been proposed [25] as an alternative for generational garbage collectors. The heap is divided into *cards* of size 2^k words, and every card has an associated bit that is set whenever a reference is stored in a cell of the associated *card*. This technique allows the use of very efficient *write barriers* [26].

Using mark bits is very efficient, but it has the disadvantage that finding a *grey* object might take time linear in the size of the heap (the mark bits of all objects need to be tested), and a complete garbage collection cycle might require time quadratic in the size of the heap. This is clearly not acceptable for deterministically efficient garbage collection. For the collection cycle to be guaranteed to finish in linear time, finding a *grey* object has to be a constant-time operation. A means to achieve this is to use a linear list of all *grey* objects. Marking a *white* object *grey* then involves adding the object to the list of *grey* objects.

Listing 3 illustrates the *write barrier* code that is needed by Jamaica to store a reference in an object. Nothing needs to be done if a *null* reference is to be stored. For non-*null* references, the colour of the referenced object needs to be checked.

```
...
if (ref != null) {
   Object **colour = adr_of_colour(ref);
   if ((*colour)==white) {
      (*colour) = greyList;
      greyList = ref;
   }
}
obj->f = ref;
...
```

Listing 3. Write barrier code required in Jamaica when storing a reference *ref* in the field *f* of object *obj*.

```
...
/* save ref in root array: */
if (ref != null) {
   Object **colour = adr_of_colour(ref);
   if ((*colour)==white) {
      (*colour) = greyList;
      greyList = ref;
   }
}
root_array[ref_index] = ref;
...
/* clear ref in root array: */
root_array[ref_index] = ref;
...
```

Listing 4. Required code to store an object in the root array and to clear the entry.

If it is *white*, the object needs to be marked *grey*, i.e., added to the list of grey objects. One word per object is reserved for the colour value. The colours *white* and *black* are encoded using special values that are invalid object addresses. Any other value represents *grey* objects, these objects form a linked list with the last element marked with a special value *last_grey* that is also no valid object address.

The *write barrier* code from **Listing 3** needs to be executed whenever a reference is saved in the *root array*. **Listing 4** illustrates the code required to save a reference in the *root array* and to release it later.

9 Analysis Using the SPECjvm98 Benchmark Suite

As we have seen above, the best time to save root references is not obvious. To find a good approach, the tests from the SPECjvm98 [27] benchmark suite have been analysed using *late saving, early saving* and *mixed*. Only one test from the benchmark suite, *_200_check*, is not included in the data since it is not intended for performance measurements but to check the correctness of the implementation (and Jamaica passes this test).

For execution, the test programs were compiled and smart linked using the Jamaica builder. They were then executed on a single processor (333 MHz Ultra-SPARC-IIi) SUN Ultra 5/10 machine with 256MB of RAM running SunOS 5.7.

In addition to the performance of Jamaica, the performance using SUN's JDK 1.1.8, 1.2 and 1.2.2 [28] and their just-in-time compilers has been measured as well. However, these values are given for informative reasons only. A direct comparison of the garbage collector implementation is not possible due to a number of fundamental differences in the implementations (deterministic real-time vs. non-deterministic garbage collection, static vs. just-in-time compilation, etc.).

9.1 When To Save References

First, the implementation was instrumented to count the number of references that

are saved during the execution of all seven tests using the three different strategies. The results are presented in **Fig. 1**.

Late saving of references requires the largest number of references to be saved. In one test, *_213_javac*, *late saving* even lead to intermediate C code that contained routines that were too large to be handled by the C-compiler on our system (*gcc 2.95.2).* The large code overhead makes this approach impractical for some applications.

Compared to the other strategies, *late saving* causes a significantly larger number of references to be saved for all but one test. Since multi-threading does not play an important role in most of the tests (the only exception being *_227_mtrt*), this indicates that many life spans that contain a *GC-point* cause several call points to be executed. Execution of several call points might also be the case if the only *GC-point* in a life span is a call point that lies within a loop, while the life span extends over this loop and the single call point is executed several times at runtime.

Early saving also causes a high number of references to be saved. Obviously, many references are saved that do not need to be saved and that are handled better by the *mixed* strategy. The reason for this are *synchronization points* that lie within the life span, but that only infrequently cause thread switches.

In all cases, the lowest number of references were saved using the *mixed saving* heuristic. Compared to *late* and *early saving*, the difference is often a dramatic reduction of a factor between two and four.

9.2 Runtime Performance

The runtime performance of the tests was analysed next. For these measurements, the tests were recompiled without the instrumentation that was used in the previous section to count the number of saved root references. The results of the performance measurements are shown in **Fig. 2**. For the analysis, the heap size was set to 32MB for all tests but *_201_compress,* since it required more memory to execute it was run with a heap of 64MB. The runtime shown is the real time, it was measured for three runs of each test. The values shown are the times from the fastest of these three runs.

In addition to the performance of Jamaica, the performance of SUN's JDK 1.1.8, 1.2 and 1.2.2 [28] and their just-in-time compilers has been measured as well.

The runtime performance decreases with the number of references saved, hence the runtime performance of the *mixed* strategy is best in all cases.

Compared to Sun's implementation, the performance of the *mixed* strategy is similar to that of JDK 1.1.8 or 1.2, while the performance of JDK 1.2.2. was improved significantly. One can expect that better optimization in the compiler implementation and direct generation of machine code will allow improvement of the performance of Jamaica as well.

9.3 Runtime Overhead of Root Saving

With the number of saved variables from **Fig. 1** and the measured runtime performance shown in **Fig. 2** we are able to estimate the runtime overhead of root scan-

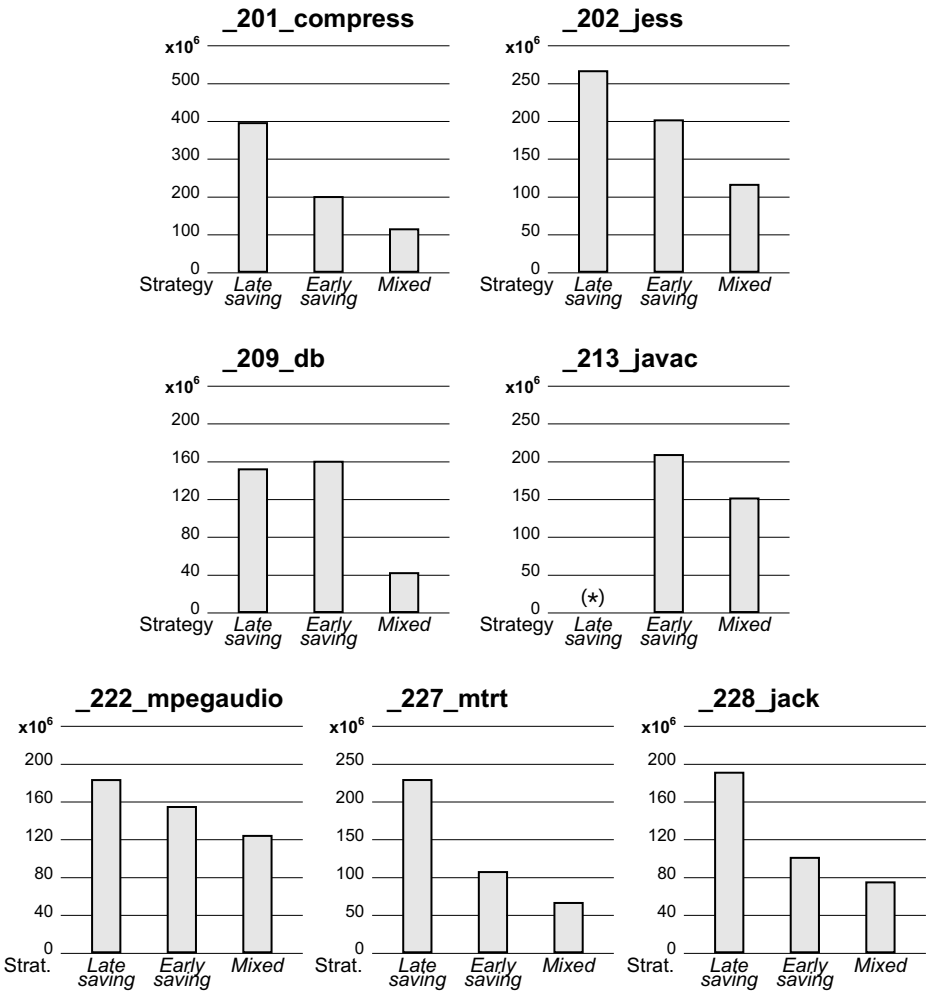

Fig. 1. Number of references saved in *root arrays* using *late saving*, *early saving* or *mixed* strategies.

(*) Late saving in this test caused some methods to become too large to be handled by the C-compiler (*gcc*)

ning. We assume that the cost to save the root references is linear in the number of references saved. This assumption holds if cache and instruction scheduling effects are ignored and the likelihood for a reference value to be *null* or the referenced object to be marked *white* is the same for the the three strategies *late, early* and *mixed saving*.

With these assumptions, linear regression analysis can be used to determine the execution time if no references needed to be saved. **Table 1** presents the results of this analysis. In addition to the estimated execution time with no referen-

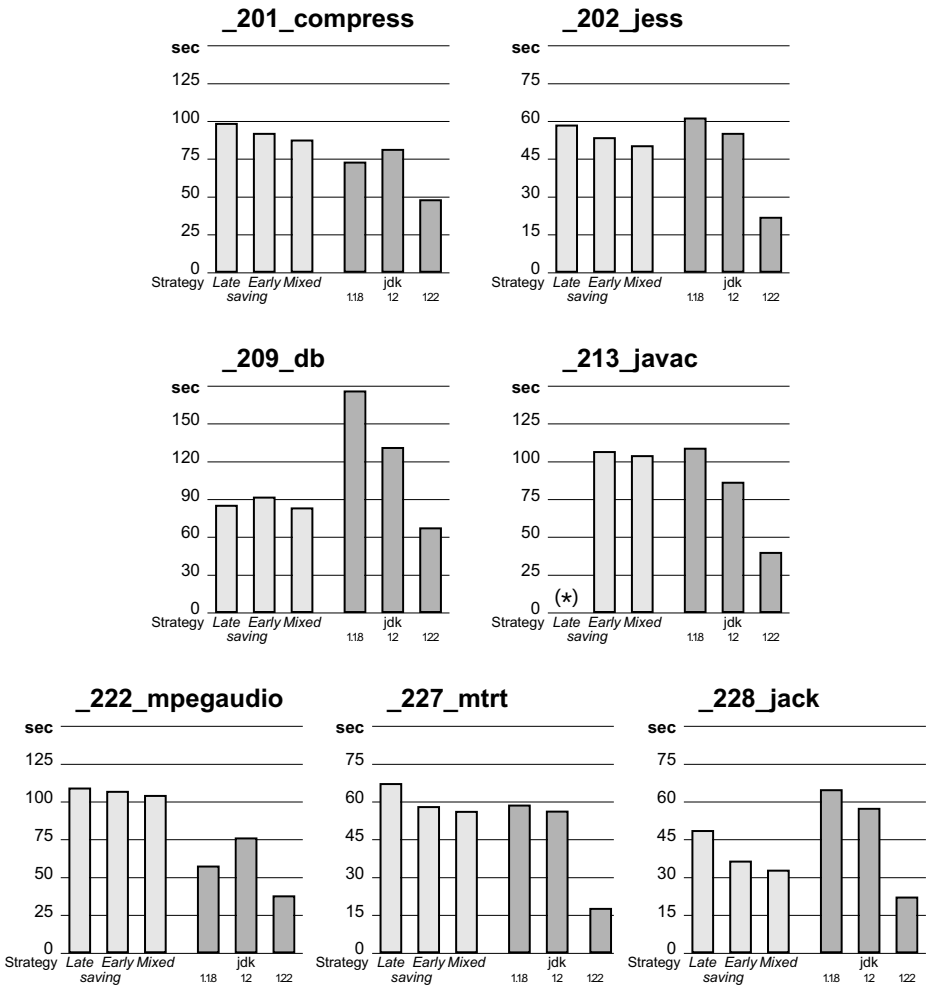

Fig. 2. Runtime performance of the SPECjvm98 benchmarks using Jamaica with *late saving*, *early saving* or *mixed* strategies and JDK 1.1.8, 1.2 and 1.2.2
(*) Late saving in this test caused some methods to become too large to be handled by the C-compiler (*gcc*)

ces saved, the percentage of overhead root saving imposes as a fraction of total runtime for all tests and the three strategies has been calculated.

The estimated overhead for root saving ranges from 3.3% up to 30.8% of the total execution time when the *mixed* strategy is used, with an average of 11.8%. The average overheads for *late* and *early saving* are significantly higher: 23.7% and 16.6%. There is only one test in which *late saving* has a lower overhead than *early saving*, *_209_db*. *Mixed* has the lowest overhead for all the tests.

The average overhead is fairly high, but one can expect that additional optimizations can help to reduce it further. Program-wide analysis can be employed

Benchmark	estimated runtime w/o root saving	root saving overhead late	early	mixed
_201_compress	79.7s	19.6%	13.9%	9.6%
_202_jess	43.9s	25.4%	18.5%	13.3%
_209_db	80.2s	6.6%	13.2%	3.3%
_213_javac	97.2s	-	9.2%	6.9%
_222_mpegaudio	94.4s	13.9%	12.1%	9.9%
_227_mtrt	51.3s	24.0%	12.1%	9.1%
_228_jack	23.0s	53.0%	37.6%	30.8%
Average		**23.7%**	**16.6%**	**11.8%**

Table 1. Extrapolated runtime without root saving overhead and percentage of runtime overhead for *late*, *early* and *mixed saving* strategies.

to avoid multiple saving of the same reference in the caller and the callee method. In addition to that, a more efficient implementation of the *write barrier* code that needs to be executed to save a reference will be possible if the implementation generates machine code directly. A global register could then be used to always hold the head of the grey list and another global register might be used to determine the colour entry associated with each object (the current implementation uses global variables that need to be read explicitly).

9.4 Memory Footprint

Another important impact of the root saving code is on the code size, which is an important factor for many applications in embedded systems with limited resources. The sizes of the executable binary files generated for the tests from the SPECjvm98 benchmark suite are presented in **Fig. 3**. The figures are the sizes of the '*strip*'ped binary files compiled for SPARC/Solaris that were used for the performance measurements in **Fig. 2**.

The code size cost of *late saving* is very significant, the binary files are 35% to 86% larger compared to *early* and *mixed saving*. The smallest binary files are achieved using *early saving*, while the file sizes for *mixed* are between 0.2% and 1.7% larger.

10 Conclusion

A new technique to do exact root scanning in an incremental, deterministic garbage collected environment has been presented. The technique allows to avoid the pause times due to root scanning. The root scanning phase of the garbage collection cycle is reduced to a cheap constant-time operation.

Different strategies to implement the technique have been implemented in a Java environment and analysed. The performance was compared to current implementations that use non-deterministic garbage collection. It has been shown that the overhead of the technique is limited and allows performance comparable to current techniques. It permits a deterministic implementations that will allow new application domains for garbage collected languages like safety-critical controls with tight hard real-time deadlines.

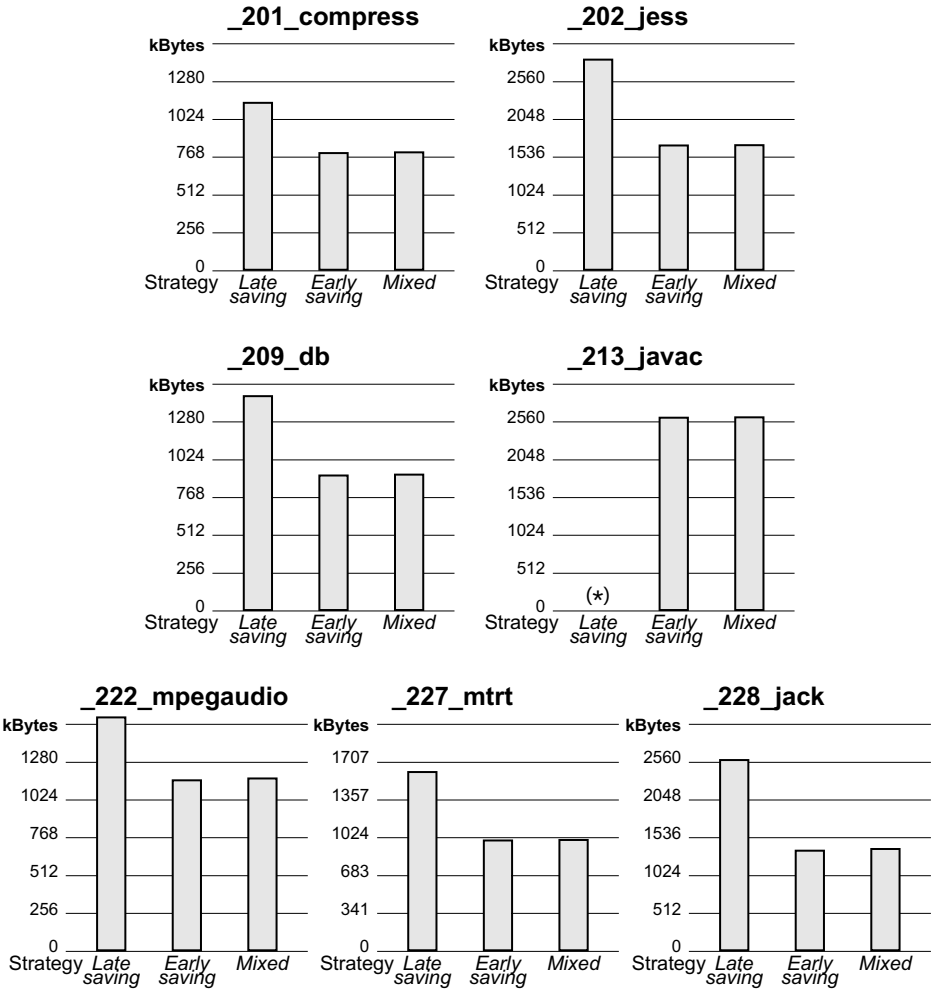

Fig. 3. File size of executable binary file for application using *late saving*, *early saving* or *mixed* strategies when compiled for SPARC/Solaris.
(*) Late saving in this test caused some methods to become too large to be handled by the C-compiler (*gcc*)

References

[1] Ken Arnold and James Gosling: *The Java Programming Language*, 2nd edition, Addison Wesley, 1998

[2] J-Consortium: *Real Time Core Extension for the Java Platform*, Draft International J-Consortium Specification, V1.0.14, September 2, 2000

[3] The Real-Time Java Experts Group: *Real Time Specification for Java*, Addison-Wesley, 2000, *http://www.rtj.org*

[4] Edsgar W. Dijkstra, L. Lamport, A. Martin, C. Scholten and E. Steffens: *On-the-fly Garbage Collection: An Exercise in Cooperation*, Communications of the ACM,

21,11 (November 1978), p. 966-975
[5] Henry G. Baker: *List processing in Real Time on a Serial Computer*. Communications of the ACM 21,4 (April 1978), p. 280-294, *ftp://ftp.netcom.com/pub/hb/hbaker/RealTimeGC.html*
[6] Damien Doligez and Georges Gonthier: *Portable, Unobtrusive Garbage Collection for Multiprocessor Systems*, POPL, 1994
[7] Danny Dubé, Marc Feeley and Manuel Serrano: *Un GC temps réel semi-compactant*, Journées Francophones des Langages Applicatifs, JFLA, Janvier 1996
[8] Joel F. Barlett: *Compacting Garbage Collection with Ambiguous Roots*, Digital Equipment Corporation, 1988
[9] Hans-Juergen Boehm: *Garbage Collection in an Uncooperative Environment*, Software Practice & Experience, Vol 18 (9), p. 807-820, September 1988
[10] Hans-Juergen Boehm: *Space Efficient Conservative Garbage Collection*, PLDI, 1993
[11] Benjamin Goldberg: *Tag-Free Garbage Collection for Strongly Typed Programming Languages*, PLDI, 1991
[12] A. W. Appel: *Runtime Tags Aren't Necessary*, Lisp and Symbolic Computation, 2, p. 153-162, 1989
[13] Amer Diwan, Eliot Moss and Richard Hudson: *Compiler Support for Garbage Collection in a Statically Typed Language*, PLDI, 1992
[14] Ole Agesen, David Detlefs, J. Eliot and B. Moss: *Garbage Collection and Local Variable Type-Precision and Liveness in Java™ Virtual Machines*, PLDI, 1998
[15] James Stichnoth, Guei-Yuan Lueh and Michal Cierniak: *Support for Garbage Collection at Every Instruction in A Java™ Compiler*, PLDI, 1999
[16] Donald E. Knuth: *The Art of Computer Programming*, Volume 1, 1973
[17] Thomas W. Christopher: *Reference Count Garbage Collection*, Software Practice & Experience, Vol 14(6), p. 503-507, June 1984
[18] Fridtjof Siebert: *Real-Time Garbage Collection in Multi-Threaded Systems on a Single Processor*, Real-Time Systems Symposium (RTSS'99), Phoenix, 1999
[19] Ole Agesen: *GC points in a Threaded Environment*, Sun Labs Tech report 70, 1998
[20] Fridtjof Siebert: *Hard Real-Time Garbage Collection in the Jamaica Virtual Machine*, Real-Time Computing Systems and Applications (RTCSA'99), Hong Kong, 1999
[21] Fridtjof Siebert: *Eliminating External Fragmentation in a Non-Moving Garbage Collector for Java*, Compilers, Architectures and Synthesis for Embedded Systems (CASES), San Jose, 2000
[22] Fridtjof Siebert: *Guaranteeing Non-Fisruptiveness and Teal-Time Deadlines in an Incremental Garbage Collector (corrected version)*, International Symposium on Memory Management (ISMM'98), Vancouver, 1998, corrected version available at *http://www.fridi.de*
[23] M. Weiss, F. de Ferrière, B. Delsart, C. Fabre, F. Hirsch, E. A. Johnson, V. Joloboff, F. Roy, F. Siebert, and X. Spengler: *TurboJ, a Bytecode-to-Native Compiler*, Languages, Compilers, and Tools for Embedded Systems (LCTES'98), Montreal, in Lecture Notes in Computer Science 1474, Springer, June 1998
[24] Pekka P. Pirinen: *Barrier techniques for incremental tracing*, International Symposium on Memory Management (ISMM'98), Vancouver, 1998
[25] Paul R. Wilson and Thomas G. Moher: *A card-marking scheme for controlling intergenerational references in generation-based GC on stock hardware*, SIGPLAN Notices 24 (5), p. 87-92, 1989
[26] Urs Hölzle: *A Fast Write Barrier for Generational Garbage Collectors*, OOPSLA, 1993
[27] *SPECjvm98 benchmarks suite, V1.03*, Standard Performance Evaluation Corporation, July 30, 1998
[28] *Java Development Kit 1.1.8, 1.2 and 1.2.2*, SUN Microsystems Inc., 1998-2000