Hard Real-Time Garbage Collection in the Jamaica Virtual Machine

Fridtjof Siebert
Jamaica Systems
siebert@real-time-systems.de

Abstract

Java's automatic memory management is the main reason that prevents Java from being used in hard real-time environments. We present the garbage collection mechanism that is used by the Jamaica Virtual Machine, an implementation of the Java Virtual Machine Specification. This mechanism differs significantly from existing implementations in the way threads are implemented, root references are found and in the object layout that is used. The implementation provides hard real-time guarantees while it allows unrestricted use of the Java language. Even dynamic allocation of normal garbage collected Java objects is possible with hard real-time guarantees.

1. Introduction

Current implementations of the Java Virtual Machine Specification [1] fail either to provide hard real-time guarantees [2, 3], or pose severe restrictions on the Java language when programming real-time code [4].

Even though a significant number of algorithms for hard real-time garbage collection have been developed, e.g., [5, 6, 7, 8, 9], Java's threading system and memory model have so far made an implementation of an exact, incremental, hard real-time garbage collector impossible. The advantages of such an implementation would nevertheless be tremendous: Standard Java code could be used for real-time programming, including the use of Java libraries, dynamic memory management that is essential in object-oriented applications and the security that can only be provided by automatic memory management.

When we are talking about real-time garbage collection, we mean a system that allows worst case execution times to be derived for any task that is performed. In the context of garbage collection this means that the worst case performance of operations like memory accesses, reference assignments, modifications of the root set and allocations are known. For such a system to be useful, this worst case performance must be small, else the worst case performance of a bigger task consisting of several of such operations quickly becomes unacceptable.

The Jamaica Virtual Machine project aims at providing such hard real-time behaviour. In this paper, we present the garbage collection mechanism that is used by Jamaica to achieve this goal.

2. Required Real-Time Guarantees

There are several guarantees that have to be given by a Java implementation so that it can be used for hard real-time programming:

2.1 Hard upper bounds for execution time

There have to be hard upper bounds for the worst-case execution times of all operation that can be performed in Java code. For the implementation to be usable, these worst-case times also have to be small so that a reasonable worst-case execution time for a task that consists of many of these operations can be determined.

In the context of garbage collection, upper bounds have to be guaranteed for the execution times of memory related operations like allocation of objects, reads and writes to the heap, and modifications of references in the 'root set', i.e., on the stack or in processor registers.

2.2 Execution has to succeed in a predictable way

Additionally to the worst-case execution time for an operation, it is also required that the operation succeeds in the desired way. A system that guarantees an upper bound on the execution time of an allocation is of little use if an allocation just results in throwing of an *OutOfMemoryException* in the case that the guaranteed execution time was too short for the garbage collector to free sufficient memory.

Instead, the system has to guarantee to free sufficient memory to satisfy all memory requests. Of course, this can only work as long as the total amount of reachable memory used by the application is limited since the physical memory is limited as well.

Requiring predictability also disqualifies any conservative garbage collection technique, like the one presented in [10]. If conservative scanning is used, random values can cause arbitrary amounts of memory to be retained, making the execution of an allocation completely unpredictable.

A similar problem that has to be dealt with is fragmentation. It has been shown that fragmentation overhead in typical applications is very low [11]. Unfortunately, Johnstone's results are of little help in systems that require hard guarantees instead of average measurements. If predictably correct execution is required, the memory management system has to fight fragmentation, else the system may fail in unpredictable ways.

2.3 Short thread pre-emption times

It has to be possible for high-priority threads to preempt lower priority threads within a short worst-case delay. Pre-emption must also be possible while garbage collection work is going on, e.g., to satisfy a large allocation request from a low-priority process. In this case it must even be possible for the high-priority thread to perform another (smaller) allocation when it is running, without being affected by the lower priority thread's activity.

Another related difficulty in many garbage collector implementations is scanning of root pointers. For this scanning, a thread is typically suspended for the time required to scan the references stored in its stack and the processor registers. This causes a delay that is clearly unacceptable when short pre-emption times are required, so a different solution has to be found.

3. Threads in the Jamaica Virtual Machine

The presence of threads in the Java language that are all able to access the same shared heap is definitely the single most important reason for real-time garbage collection to be difficult in Java implementations. Usage of native (kernel-level) threads for the implementation of Java threads complicates root scanning, since little is known about the threads' states, as to where to find valid reference values, etc.

New Java implementations like [12] do attack this problem by not relying on system threads at all. Instead, thread-switching is implemented explicitly, allowing thread-switches only after the execution of a certain number of bytecode instructions. Since these thread switches can occur only at very few places that are known to the VM, exact information on the state of all threads is available. The disadvantage of this approach is that blocking operations, like I/O that is performed within native code called via the Java Native Interface [13], causes the complete virtual machine and all Java threads to stop.

Another approach is to use native threads, but to provide exact information only at so-called *GC Points* as proposed by [14] or [7]. These mechanisms require that a thread is notified when the GC needs to access its state. At the beginning of a GC cycle all threads have to be notified for root scanning. This complex operation is likely to cause significant worst-case pre-emption delays making it unusable for our approach.

It has also been proposed to provide exact information on a thread's state on every machine instruction [15]. The main reasons that make this proposal inadequate for Jamaica are the difficulty it incurs on portability and its complexity in general.

For Jamaica, we have decided to use a new mechanism that is a mixture between the first two approaches: Native threads are used to implement Java threads. But a global semaphore is used to allow at most one Java thread to execute at any time. Frequently executed synchronization points guarantee that preemption time is limited and small. Additionally, the global semaphore is released for the execution of native code, execution of native code does not stop any Java threads and even blocking operations like I/O can be performed in native code while other threads continue running normally.

The code that needs to be executed at a synchronization point is very simple: A static flag that indicates the need for synchronization has to be tested. Only in the case when this flag is set the current thread's state has to be saved and the global semaphore can be released so that another Java thread can continue execution.

Another advantage of this scheme using synchronization points is that code executed between two synchronization points is automatically atomic. Operations like memory writes that require execution of write-barrier [16] code are simplified, it is sufficient to ensure the required GC invariants by the time the next synchronization point is reached.

Other parts of the virtual machine implementation do also profit from the automic atomicity of code sequences that do not contain synchronization points. One example is the code required to enter or exit Java monitors, that can be implemented very efficiently using mechanisms presented in [17] and [18].

4. Root Scanning

Root scanning is the task of finding all objects on the heap that are referenced by variables that are not stored on the heap themselves. This typically includes static variables, the stacks of all threads and the processor registers.

For static variables, root scanning can easily be implemented: The Jamaica Virtual Machine stores all static variables on the heap so they are not part of the root set.

Scanning the stack and registers poses a bigger problem since scanning can take a significant amount of time while the scanned thread has to be suspended, causing an intolerable worst-case delay for thread preemption.

The approach to root scanning taken by Jamaica is quite radical: There is only one single root pointer, and all other references that are present locally in registers or on stacks also have to be present on the garbage collected heap. Thanks to the fact that at most one Java thread is executing at any time while all other Java threads are stopped at synchronization points, copying reference values to the heap is required only at synchronization points and on calls. To store these references into the heap, the write-barrier code described below is used to ensure consistency of the garbage collector invariants.

5. Object Layout

Jamaica uses a new object layout to represent Java objects. The layout has been developed to avoid the overhead caused by the use of handles and the complexity due to moving objects, while still avoiding fragmentation.

The heap is partitioned into blocks of just one single fixed size. This size is configurable, for most applications a size of 32 Bytes seems to perform best.

Java Objects and any memory that is internally allocated by Jamaica is constructed out of one or several of these blocks that may be non-contiguous in memory. For allocations that exceed the block size, the Java object is partitioned into a graph of these blocks and several such blocks are allocated. For normal Java objects, a linked list of blocks is used, while arrays and strings are represented in tree-like structures.

This object layout completely avoids fragmentation, while there is no need to move objects nor to use handles or otherwise update object references.

6. The Garbage Collection Algorithm

The basic garbage collection algorithm that is used in Jamaica is a simple incremental mark and sweep collector as described in [19]. This algorithm uses three sets of objects that are distinguished by the colours *white, grey* and *black*. A cycle starts with all objects being *white*, and the objects reachable from root pointers are marked *grey*. It then proceeds as long as there are *grey* objects by taking a *grey* object, marking it *black* and marking all *white* objects that are referenced by this object *grey*. A write-barrier ensures that *white* objects are never referenced by *black* objects as a result of an assignment performed by the application. When there are no *grey* objects left, all *white* objects are garbage and are recycled in the sweep phase. In this phase *black* objects are converted back to *white*, so that the next cycle can start.

The Jamaica garbage collector does not directly deal with Java objects or Java arrays and does not know about their structure. Instead, it is working on single fixed size blocks. Working on blocks simplifies the garbage collector loop significantly, while it automatically provides small units of garbage collection work that can be done in incremental steps: The basic operations performed by the garbage collector are scanning or sweeping a single block. The garbage collector's work is allowed to be pre-empted after each of these basic operations, so that other threads can run even while garbage collection work is going on.

The garbage collector has to be able to distinguish reference values from non-references that are stored on the heap. To do this, a bit array large enough to hold one bit for every word that is present on the heap is used. All words that contain references have their corresponding bit set. This allows fast identification of reference values, while it has a minimal memory overhead.

Additionally, memory space is required to hold the colour of every block. Two bits per block are sufficient for this, since there are only three different colours. But allocating just two bits for the colour has a very important impact on the garbage collector's efficiency, since it is not possible to find a *grey* block in constant time. In the worst case, all blocks have to be scanned just to find one more *grey* block, causing an overall quadratic performance of the collector. Caching systems for the *grey* blocks as proposed in [20] can reduce this overhead in the average case, but for guaranteed real-time behaviour this is not sufficient.

An alternative used in [21] would be to use doubly-linked lists, one list for each colour that is used. This approach allows to find *grey* objects and to change an object's colour in constant time, but it has a significant memory overhead of two words per block. Also, the write-barrier code becomes fairly complex since it has to perform several assignments to correctly unlink an object from one list and relink it to a new one.

The approach taken by Jamaica is using one word per block for the colour. For colours *white* and *black*, this word contains a special value (*0* and *-1*, respectively) indicating the colour. Any other value indicates that the block is *grey*. All *grey* objects are stored in a linked list, using the colour word to store the reference to the next element in this list. Another special value (*1*) is used to mark the last element in this list. Adding a block to and removing the first block from the *grey*-list are efficient operations that can be performed in constant time, so that a complete garbage collector cycle is guaranteed to finish in a time that is linear in the number of allocated blocks.

As mentioned above, a write-barrier has to be used to ensure that no *black* block ever contains a reference to a *white* block. For any write of a reference to a *white* block into the heap, the reference is added to the list of *grey* blocks before the write is performed. The code required for a write of a reference to a field $a.f = r$ can be illustrated as follows:

```
if ((r != NULL) && (colour(r) == white)) {
     colour(r) = grey_list;
     grey_list = r;
}
a.f = r;
```

Atomicity of this operation is guaranteed since no synchronization point is executed between *greying* and performing the write.

7. Garbage Collector Activation

To guarantee to satisfy all allocation requests, it has to be ensured that the garbage collector performs sufficient work. On the other hand, it is desirable to avoid garbage collector overhead whenever it is not required either because no allocation is going on or because there is sufficient free memory. Furthermore, garbage collection work should be distributed fairly on the application threads: Threads that perform much allocation should pay for the garbage collection work with their execution time while other threads that perform no or little allocation should not be affected by this work.

The garbage collection work in Jamaica is therefore coupled with allocation, in the way proposed in [22]. For every block that is allocated, the number of garbage collection increments that are performed is $P = M/F$, where M is the size of the heap (in number of blocks) and F is the amount of memory that is currently free. As has been shown in the paper, this guarantees sufficient garbage collection progress as long as the amount of reachable memory stays below the system's total memory, while a worst-case upper bound for P can be determined if the amount of reachable memory R stays below a fixed upper bound of memory K (so that $R \leq K$ always holds).

Since one garbage collector cycle consists of traversing all allocated memory twice, once during the mark and once during the sweep phase, every work increment consists of marking or sweeping two blocks. This guarantees that a cycle finishes at the latest after the number of increments performed exceeds the amount of allocated memory, as is required by this approach (see [22] for the details). **Table 1** illustrates the worst-case number of increments that are required for the allocation of one block, as a function of the upper bound of reachable memory $k = K / M$.

On the PowerPC processor, a marking or sweeping of one block takes about $w_{inc} = 80$ (compiled) machine instructions. For an application that uses at most two thirds of the total memory as reachable objects P_{max} is 13.38, so that the allocation of one block of memory is limited by $2 \cdot w_{inc} \cdot P_{max} \leq 2 \cdot 80 \cdot 14 = 2240$ machine instructions of garbage collector work. Our experience

k:	0%	50%	66.7%	70%	75%	80%	85%	90%	95%	97.5%	100%
P_{max}:	1.0	6.61	13.38	15.72	20.45	27.65	39.77	64.21	137.9	285.8	∞

Table 1: Upper bound for the required number of GC increments per block allocated as a function of the upper bound of the reachable memory.

with the implementation so far shows that the worst-case occurs very infrequently, while the average case requires significantly less garbage collector work.

A high priority thread can pre-empt a lower-priority thread while it is performing this garbage collection work, since synchronization points are present after marking or sweeping of each single block.

The worst-case behaviour can be improved by fixing P_{max} instead of using a function of the amount of free memory. As shown in [9] and [22], a value of $P_{max} = 2/(1\text{-}k)$ is sufficient for an application that uses a fraction of the heap that never exceeds k. In the example above, this will result in $2 \cdot w_{inc} \cdot P_{max} = 2 \cdot 80 \cdot 6 = 960$ instructions of garbage collector work per allocation of one block. But this improvement in worst-case performance comes at a high cost in average-case performance, ignoring the actual memory usage of the application. Furthermore, if the application exceeds the limit on the amount of reachable memory only slightly, the system might fail to recycle sufficient memory. Nevertheless, Jamaica provides optional support for a fixed P_{max} since applications that only depend on the worst-case behaviour can profit from it.

8. Experimental Data

To be able to estimate the effect on the run-time performance of our garbage collection scheme we have collected information during execution of three large Java applications. To record this information, the applications were compiled with an extended version of the TurboJ Java-bytecode compiler [23].

The applications that were examined are

1. HotJava [24], a Web browser written in Java. During the test we used it to browse the Web pages of The Open Group Research Institute [25].
2. SwingSet [26], the demo application that comes with the Swing GUI class library and that makes extensive use of the classes in this library.
3. TurboJ [23], a highly optimizing Java bytecode compiler. We recorded its behaviour while compiling all 1610 Java standard classes.

To facilitate reproduction of our results, we disabled inlining, since the aggressiveness of inlining has an important impact on other optimizations.

8.1 Saving of Root References at call points

We have measured the overhead of storing copies of local references into the heap as it is required at every call to allow constant-time root scanning as presented above. To do this, we counted the number of local variables of reference types (these include local variables found in the Java bytecode, local variables to emulate the Java stack and temporary variables introduced by the compiler for different purposes) that exist at each call point. We have recorded the total number of such variables in a method, and the number of variables that are life at the call point. The results of this simulation are present in **Figure 1**, given as average numbers of local reference variables at the execution of a call. The average number of references

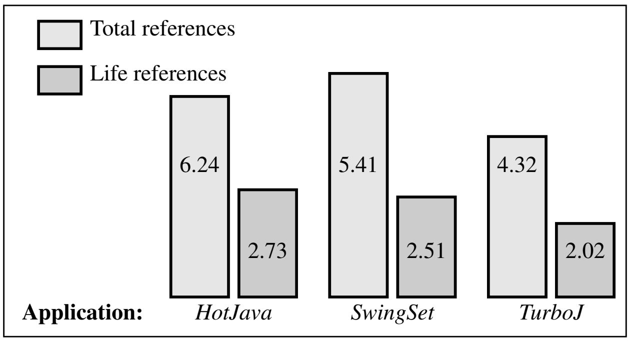

Figure 1: Average number of total and life reference variables on a call

present at a call varies from 4.32 to 6.24, while the number of life references is only between 2.02 and 2.73. The additional overhead of copying these references to the heap on every call corresponds to just a few heap stores per call, while saving only the life variables has about half the overhead of saving all references. Since life values have to be stored in a caller-safe place anyway, the additional overhead appears easily acceptable, especially when regarding the benefits of removing GC latencies due to root scanning completely and avoiding the need to provide any further information on root variables.

8.2 Memory accesses

Next, we compared the usage of fixed size blocks with traditional object models to estimate the cost or gain in execution time overhead. We compare the following five scenarios:

1. A non-moving garbage collector that uses direct references.
2. A defragmenting moving garbage collector that uses handles to update reference values.
3. The fixed size scheme used by Jamaica with blocks of 32 bytes and representing objects as linked lists and arrays as trees. Small objects or arrays can be stored in a single block and can be

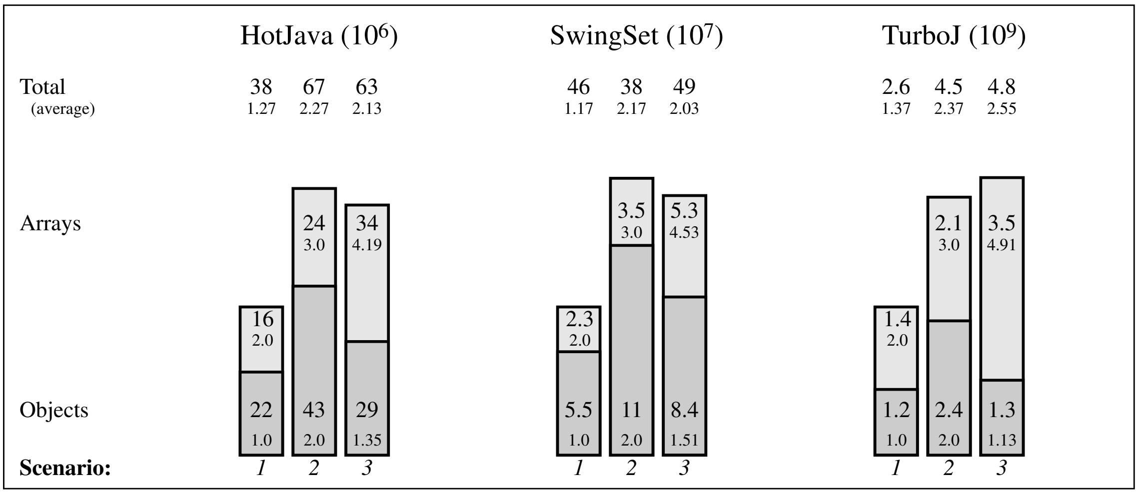

Figure 2: Memory accesses needed for different memory management scenarios

accessed directly, while larger structures require accessing several blocks to read or write a field or array element.

Figure 2 compares the number of memory accesses needed for our three test cases. The absolute numbers of memory accesses and the relative numbers of memory accesses per object or array access are shown. Object accesses for the non-moving scenario (*1*) require only a single memory access, while for arrays a second access is required to read the array's length and do the index checking.

For the moving scheme using handles (*2*), an additional memory access is required to read the actual address of the object or array. We see that this scheme doubles the number of memory accesses for objects, while it increases the number by 50% for arrays.

For the fixed size block approach (*3*), things are a bit more complicated. Fields with small offsets can be read or written in a single memory access, while fields with larger offsets might require 2, 3, or more accesses. But small offsets are very frequent so that a single memory access is sufficient for most object accesses. Compared to the scenario using handles the test cases need between 1.13 and 1.51 memory accesses instead of 2.0.

Accessing arrays is significantly more complex in the fixed size approach, Very small arrays have their data within the array header, and three memory accesses are sufficient to read or write an array element. Larger arrays need 4, 5, 6, or more accesses depending on the size of the array.

Regarding the total number of memory accesses needed for the moving and fixed size strategies, we see that the fixed size approach performs slightly better for the HotJava and SwingSet test cases, while it is performing slightly worse for TurboJ, where the additional memory accesses due to the high frequency of accesses to large arrays cannot be compensated for by the gain due to the significantly cheaper accesses to objects.

9. Conclusion

The Jamaica Virtual Machine approaches the problem of garbage collection in Java implementations in a way that differs significantly from current implementations. The garbage collector has an impact on a significant part of the implementation, including threads, local variables and the object layout.

The result is a virtual machine that provides hard real-time guarantees for normal Java programs without the need for specific attention by the programmer. Short pre-emption times are guaranteed and even memory allocation can be performed with short hard real-time guarantees. No current Java implementation we know of gives comparable guarantees.

10. References

[1] Tim Lindholm, Frank Yellin: The Java Virtual Machine Specification, Addison-Wesley, 1996

[2] *Java Development Kit 1.1*, SUN Microsystems Inc., 1999 http://www.javasoft.com/products/jdk/1.1

[3] *HP ChaiVM Release 3.0*, Hewlett-Packard, http://www.chai.hp.com/

[4] Kelvin Nilsen, Steve Lee: *PERC™ Real-Time API (Draft 1.3)*, NewMonics Inc., July 1998

[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] Guy E. Blelloch and Perry Cheng: *On Bounding Time and Space for Multiprocessor Garbage Collection*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1999

[7] Damien Doligez, Georges Gonthier: *Portable, Unobtrusive Garbage Collection for Multiprocessor Systems*, ACM Symposium on Principles of Programming Languages (POPL), 1994

[8] Kelvin Nilsen: *Reliable Real-Time Garbage Collection of C++*, Computing Systems, Volume 7, no. 4, 1994

[9] Paul R. Wilson: *Uniprocessor Garbage Collection Techniques*. Submitted to the ACM Computing Surveys. 1998

[10] Joel F. Barlett: *Compacting Garbage Collection with Ambiguous Roots*, Digital Equipment Corporation, 1988

[11] Mark S. Johnstone and Paul R. Wilson: *The Memory Fragmentation Problem: Solved?*, International Symposium on Memory Management, 1998

[12] *The KJava Virtual Machine*, White Paper, SUN Microsystems, 1999

[13] *Java Native Interface Specification*, SUN Microsystems Inc., 1997

[14] Ole Agesen: *GC Points in a Threaded Environment*, SUN Labs Tech report 70, 1998

[15] James M. Stichnoth, Guei-Yuan Lueh and Michal Cierniak: *Support for Garbage Collection at Every Instruction in a Java Compiler*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1999

[16] Pekka P. Pirinen: *Barrier techniques for Incremental Tracing*, International Symposium on Memory Management, 1998

[17] David F. Bacon, Ravi Konuru, Chet Murthy, Mauricio Serrano: *Thin Locks: Featherweight Synchronization for Java.* ACM SIGPLAN Conference on Programming Language Design and Implementation, 1998

[18] Byung-Sun Yang, Junpyo Lee, Jinpyo Park, Soo-Mook Moon, Kemal Ebcioğlu, Erik Altmann: *Lightweight Monitor for Java VM*, ACM Computer Architecture News, Vol 27-1, March 1999

[19] Edsgar W. Dijkstra, L. Lamport, A. Marin, 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

[20] Malcolm Wallace and Colin Runciman: *An incremental garbage collector for embedded real-time systems,* Proceedings of the Chalmers Winter Meeting, pages 273-288, Tanum Strand, Sweden, 1993. Published as Programming Methodology Group, Chalmers University of Technology, Technical Report 73.

[21] Henry G. Baker: *The Treadmill: Real-Time Garbage Collection without Motion Sickness*, ACM SIGPLAN Notices 27,3, 1992, pp. 66-70

[22] Fridtjof Siebert: *Guaranteeing non-disruptiveness and real-time Deadlines in an Incremental Garbage Collector*, International Symposium on Memory Management, 1998

[23] *TurboJ V1.1.2*, ahead-of-time Java compiler, The Open Group Research Institute, Grenoble, France, 1997-1999 http://www.gr.opengroup.org/openitsol/

[24] *HotJava (tm) Browser*, V1.1.4, 1998, Sun Microsystems Inc., http://www.javasoft.com/products/hotjava/1.1.4/

[25] The Open Group Research Institute, Grenoble, France. http://www.gr.opengroup.org/

[26] The *examples/SwingSet* program of the Swing 1.0.1 class library, SUN Microsystems Inc., 1998, http://www.javasoft.com/products/jfc/