Realtime Garbage Collection in the JamaicaVM 3.0

Dr. Fridtjof Siebert
aicas GmbH
Haid-und-Neu-Str. 18
76131 Karlsruhe, Germany
siebert@aicas.com

ABSTRACT

This paper provides an overview of the realtime garbage collector used by the RTSJ Java Virtual Machine JamaicaVM. A particular emphasis will be made on the improvements made in with release 3.0 JamaicaVM.

The JamaicaVM garbage collector is incremental to an extreme extent: single incremental steps of the garbage collector correspond to scanning only 32 bytes of memory and have a worst-case execution time in the order of one μsec. The JamaicaVM garbage collector uses automatic pacing, making the system easier to configure than a garbage collector using explicit pacing that requires information on the application's allocation rate.

The recent improvements of the garbage collector that will be present in this paper include support for automatic heap expansion; reduction of the memory overhead for garbage collector internal structures; and significant performance optimisation such as a faster write-barrier and a sweep phase that does not need to touch the objects and therefore reduces the number of cache misses caused during sweep.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS—*Real-time and embedded systems*; D.3.4 [**Software**]: PROGRAMMING LANGUAGES—*Processors: Memory Management (garbage collection)*; D.4.7 [**Software**]: OPERATING SYSTEMS—*Organization and Design: Real-time systems and embedded systems*

General Terms

ALGORITHMS, PERFORMANCE, RELIABILITY, LANGUAGES

Keywords

realtime, garbage collection, Java, RTSJ

1. INTRODUCTION

1.1 Realtime Garbage Collection and the RTSJ

The Real-Time Specification for Java (RTSJ) provides essential infrastructure for realtime programming in Java [4]. However, it sacrifices important advantages of Java by requiring realtime code not to rely on automatic garbage collection, but to use a more manual memory management approach using scoped memory areas.

On systems that do not provide realtime garbage collection, realtime code that does not run in a *NoHeapRealtimeThread*, which is an environment that forbids accesses to memory that is under the control of the garbage collector, will be preempted by garbage collection activity and therefore will not be able to give reasonable realtime guarantees.

The combination of the RTSJ and realtime garbage collection avoids this restriction. Even time critical code can access the heap or allocate objects on the heap.

1.2 JamaicaVM's Garbage Collection Algorithm

The basic garbage collection algorithm that is used in JamaicaVM is a simple incremental mark and sweep collector as described by Dijkstra et al in 1976 [7]. 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 known to be garbage and will then be recycled in the sweep phase. In this phase, *black* objects are converted back to *white*, so that the next cycle can start with all allocated objects being *white*.

The JamaicaVM garbage collector does not deal with Java objects or Java arrays directly and does not know about their structure. Instead, it works 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 may be pre-empted after each of these basic increments, 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 within objects on the heap. To do this, one additional flag bit is assigned to each word on the heap. All words that contain references have their corresponding bit set. This enables 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 by Wallace and Runciman [19] can reduce this overhead in the average case, but for guaranteed realtime behaviour this is not sufficient.

An alternative used by Baker [3] 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 original approach taken by JamaicaVM 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. 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. A reduction of the memory demand for the colours and an improvement of the runtime performance of the write-barrier was an important goal in JamaicaVM 3.0. How this goal was achieved will be presented in the following sections 5 and 6.

The following sections will first explain JamaicaVM's garbage collector's solution to three important issues left open in Dijkstra's original algorithm: Section 2 explains how the root scanning phase is avoided; section 3 describes in more detail how memory fragmentation is avoided by the use of fixed size blocks; and section 4 explains when the garbage collector work is performed. Major improvements in JamaicaVM 3.0 include the support of several non-contiguous chunks of memory that permit dynamic heap expansion, which will be explained in section 5, a new write-barrier explained in section 6 and a sweep phase that does not need to touch the blocks in section 7. Section 8 will present performance results before Section 9 lists important related work.

2. CONSTANT TIME ROOT SCANNING

Root scanning means finding all references that are stored outside of the heap that point to objects on the heap. This mainly includes the stack of each thread. Scanning of the stack to find references typically requires stopping the corresponding thread. 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.

The solution employed by JamaicaVM that was originally published in [17] 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 and virtual machine have 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 the garbage collector 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. To achieve a good performance for saving root references it is therefore essential to have an efficient write-barrier.

Another minor problem are root references in global (static) variables. The solution taken by JamaicaVM is to simply store all static variables on the heap in a way that they are also reachable from the global root pointer. Another possible solution would have been 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.

2.1 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 thread-switch 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 thread-switch point. It is therefore necessary to save all local references whose life span contains a thread-switch point or a call point. For simplicity of terms, thread-switch points and call points will both be referred to as GC-points [8] in the following text. It is not clear when the best time to save a reference would be. There are two obvious possibilities:

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.

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.

The solution taken by JamaicaVM and which has resulted in the best runtime performance is a mixture between late and early saving:

Mixed: Since thread-switch points are executed only when a thread switch is actually needed, it makes sense to use late saving for life spans that only contain thread-switch 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 thread-switch point. Whenever a life span of a reference variable 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.

3. FIGHTING FRAGMENTATION

JamaicaVM uses a unique 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 original idea of this object layout has been presented earlier [16], this section presents the actual memory layout in JamaicaVM in more detail including object headers, etc.

The heap is partitioned into blocks of just one single fixed size. This size can be configurable, however, for most applications a size of 32 bytes (eight 32-bit words) performs best, so this size has been selected as the default (Figure 1).

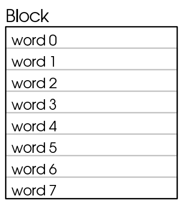

Figure 1: A fixed-size block of eight machine words

Java Objects and any memory that is internally allocated by JamaicaVM 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. This object layout completely avoids external fragmentation, while there is no need to move objects nor to use handles or otherwise update object references.

For normal Java objects, a linked list of blocks is used. Figure 2 illustrates the structure of a Java object with 15 words of field data that is spread over three blocks. The header block additionally contains space for the type of the object, its monitor and a field that is used by the write-barrier code (see section 6). The first two blocks end with a *next* field that contains a reference to the next block in the list, while the last block is only partially used by the fields, the last four words remain unused.

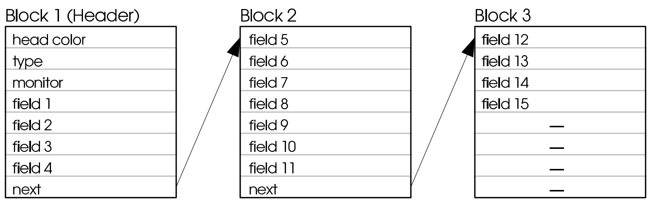

Figure 2: Java Object implemented as list of fixed-size blocks.

Arrays and strings are represented in a tree structure illustrated in Figure 3. The first three words in the header block are equal to those of a Java object: *head colour*, *type*, and *monitor*. After that, additional words contain the length of the array and the depth of the tree. This depth value permits a very efficient traversal of the tree such that an efficient access to the array elements is possible even though the tree structure appears to be complex. The large branching factor of eight in each inner node of the tree ensures that even large arrays will be represented by trees with a small depth (e.g., an array that contains 1MB of element data has a depth of only six).

Even though the tree representation is very efficient, it is often desirable to use the more efficient contiguous represen-

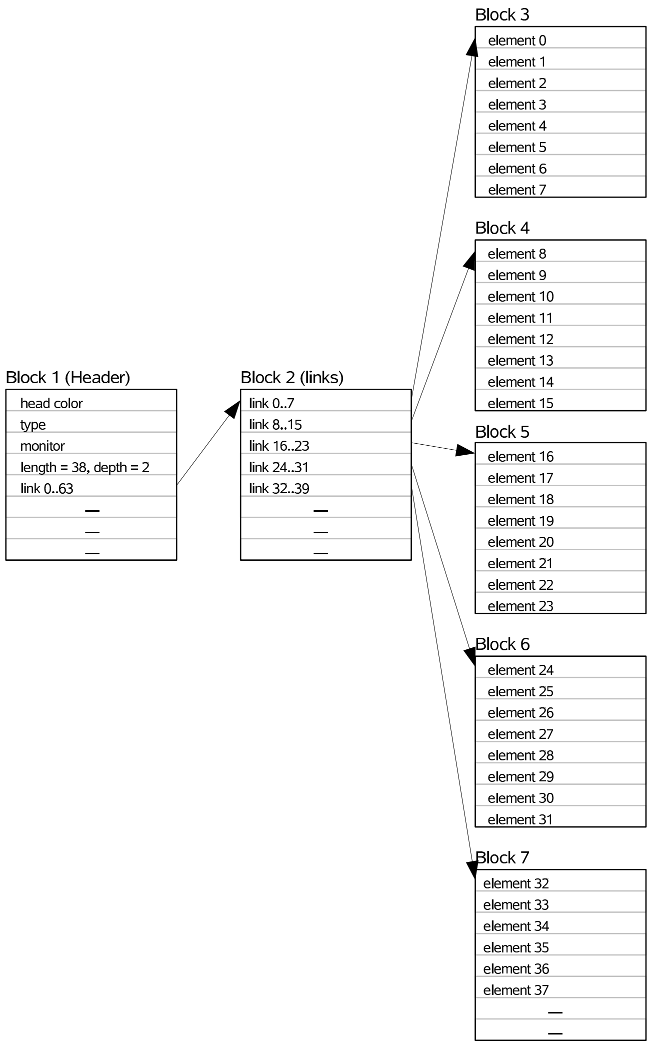

Figure 3: Java array implemented as tree of fixed-size blocks.

tation of an array whenever the memory is not fragmented at allocation time. A contiguous representation results in even faster array accesses and in reduced memory demand. For these reasons, a contiguous representation will be used whenever an array is allocated and a contiguous range of memory is readily available. The structure of such a contiguous array is the same as that of a tree array with the only difference that the *depth* is set to zero. The access code for such an array does not need to be modified, the access to a contiguous array is the same as the access of a tree array that has a depth of zero.

Using this object model that consists of fixed-size blocks, JamaicaVM's memory management does not need to defragment memory. The garbage collector becomes significantly simpler in that it does not need to update reference values or 'pin' objects that are referenced from native code. Also, the access to objects becomes simpler since no handles[1] are required. Since references do not need to be updated to point to a new address, the garbage collector does not need to be informed about all direct references that exist to an object, it is sufficient to ensure that for every referenced object, one reference is known to the garbage collector.

[1]indirect references that can be updated easily after an object has been moved to a new address by the garbage collector

Since Java objects are typically small, one or two blocks are sufficient for most Java objects. This means that typical accesses to fields require just a single memory access (for fields in the header block) or two memory accesses for fields in the second block. The resulting average number of memory accesses for fields is consequently less compared to the use of handles that require two memory accesses for each field access.

For accesses to arrays, the performance is slightly worse compared to a contiguous representation. However, the overall performance does not suffer significantly.

4. AUTOMATIC PACING

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 JamaicaVM is therefore coupled with allocation, in the way proposed in [15]. For every block that is allocated, the number of garbage collection increments that are performed is $P(F) = M/F$, where M is the size of the heap (in number of blocks) and F is the number of blocks that are currently free. The garbage collector guarantees to finish one cycle after the number of GC increments (the sum of all $P(F)$) exceeds the number of blocks allocated at the end of the cycle. Since a mark-and-sweep collector requires two scans of the allocated memory, one during the mark phase and one during the sweep phase, each GC increment corresponds to two mark or sweep steps, depending of the phase the garbage collector is in.

Performing $P(F)$ increments of GC work for each block of memory allocated guarantees sufficient garbage collection progress as long as the amount of reachable memory stays below the system's total memory. Furthermore, a worst-case upper bound P_{max} for P can be determined if the amount of reachable memory R stays below a fixed upper bound K (such that $R \leq K$ always holds). See appendix A for a sketch of a proof for this upper bound. This upper bound also gives an upper bound for the time required to allocate one block of memory.

An upper bound of the maximum amount of reachable memory K should be known by the real-time developer[2], so an upper bound for the GC work on an allocation can be determined. Due to space limitations, the details and the derivation of the dependency between the upper bound on reachable memory K and the upper bound for the GC work P_{max} cannot be presented in full detail here. The interested reader is referred to the authors PhD theses [18].

The worst case execution time of a unit of garbage collection work depends on the target architecture, processor

[2]The determination of an upper bound for the amount of reachable memory is, however, a difficult task for non-trivial applications. Attempts were made to use escape analysis to provide the required object lifespan information [9]

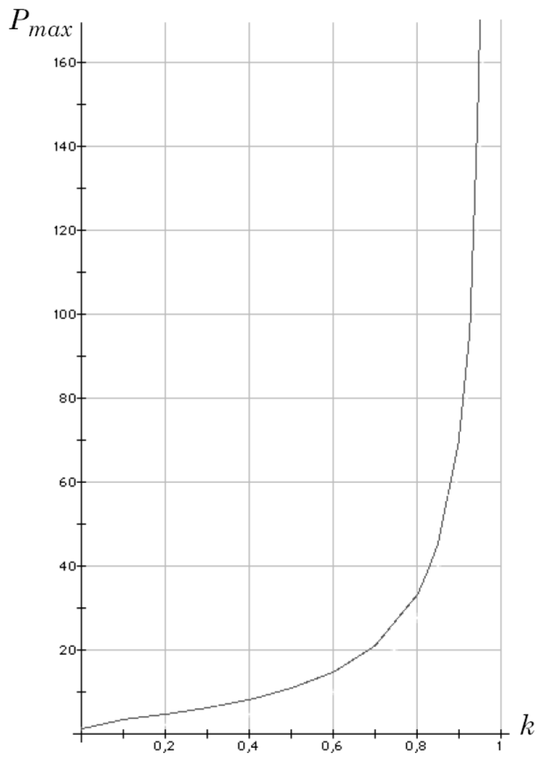

Figure 4: Worst-case number of GC increments (mark and sweep work units) required for the allocation of one block as function of the maximum percentage of reachable memory $k = K/M$.

implementation, cache architecture and clock rate. It is typically in the order of one μsec. For an application that uses up to 66% of the heap memory as reachable memory, $P_{max} = 18$, such that we get a worst-case execution time of this allocation of $18\mu sec$ for a system that has a worst-case execution time of $1\mu sec$ for an increment of garbage collection work.

4.1 Preemption during GC work

As has been explained in the first part of this section, a thread that allocates memory pays for this allocation by a number of GC increments defined by the P function. Depending on the current phase of the GC, a single increment corresponds to two steps of marking or sweeping of a single 32-byte block. In both cases, this increment is a very quick operation that can be performed within a few dozen machine instructions[3]. Each single increment has to be performed atomically, such that the latency of GC preemption by a higher priority thread is influenced by the length of a single step, but not by the size of the object that is allocated.

4.2 GC jitter

Using the $P(F)$ function to guide the garbage collection work makes the amount of garbage collection activity on an allocation a function of the amount of memory that is currently free, causing the GC load to vary over time. However, since an upper bound for P_{max} can be given, we can give timing guarantees. In practice, the jitter of a running system is relatively low: Low values of F cause an increase in GC work which increases F while a larger F reduces the GC work causing F to decrease automatically.

[3]For a PowerPC processors, the worst-case length was calculated to be 289 cycles [18]

However, for a system for which this jitter is not desirable, JamaicaVM provides the alternative to set P to a constant value. For any given upper bound of reachable memory K a $P_{const}(K)$ can be found that ensures that the GC work is sufficient to reclaim memory sufficiently enough. The advantages of using a constant P are that the jitter due to GC work is reduced significantly and that the worst case for P is equal to the average case and lower than the P_{max} for the dynamic $P(F)$ function. The disadvantages are that the systems requires a reliable analysis of the maximum reachable memory and it will no longer degrade gracefully as with the dynamic GC work function: If the reachable memory exceeds K at one point, an $OutOfMemoryError$ may be the result, while in the dynamic approach, $P(F)$ might temporarily exceed P_{max} slightly. Another disadvantage of using P_{const} is an overall higher GC overhead and poorer average throughput for the application.

5. USING CHUNKS

Earlier versions of the JamaicaVM had a fixed heap size. This had important advantages for the implementation: The information required for the garbage collector could be stored in an array separated from the array of fixed size blocks that are used to hold the Java objects while it is very efficient to determine the GC information for a given block using simple arithmetic on the block's address.

Figure 5 illustrates the original heap layout: Parallel to the array of blocks, there was an array with the colour entries for each block (one word per block) and an array of bits that indicates which words on the heap represent references.

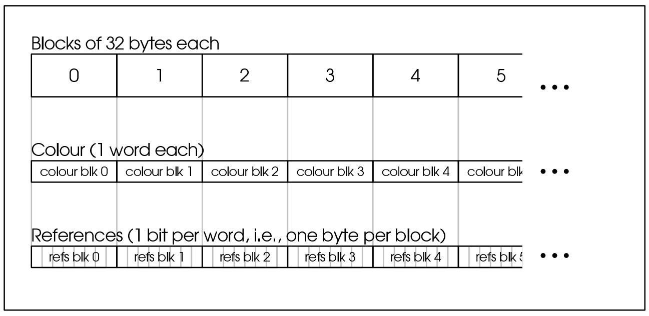

Figure 5: Original heap memory consists of a single array of blocks, a single array of colour entries and one bit per word to find references

However, this approach required the user to specify the heap size statically. Even though such a fixed heap size is required for a configuration of the real-time garbage collector, a typical Java developer expects that the heap size will be set automatically by the virtual machine within the bounds specified via the $-Xms$ and $-Xmx$ options. To be able to run the virtual machine without requiring the user to specify the heap size, a dynamically growing heap is required. This requires a structure that is more dynamic than the original structure using just a single large array of blocks and single arrays for the colour and the reference flags.

To enable dynamic heap growth, JamaicaVM 3.0 now uses a list of memory chunks. Each chunk is a contiguous range of memory allocated from the operating system. Different chunks might reside at completely different memory addresses. The structure of the chunk list is illustrated in Figure 6.

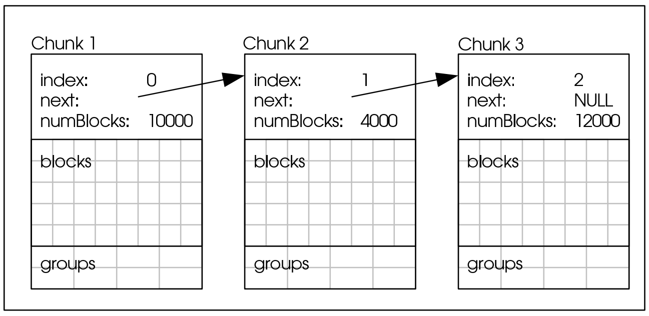

Figure 6: Heap memory consists of several chunks allocated from the OS

Each chunk contains two arrays: one array for the blocks and one array with a structure for each group of blocks. A group of blocks consists of eight contiguous blocks.

The contents of the group structure are illustrated in Figure 7. The group contains one byte for each block of the group (i.e., one bit for each word in these blocks) that indicate where references are stored. One 32-bit word in the group structure contains four flag bits for each block in the group. These flags are used for the colour-bits and the head-flag (see below). Finally, each group contains a *next link* field.

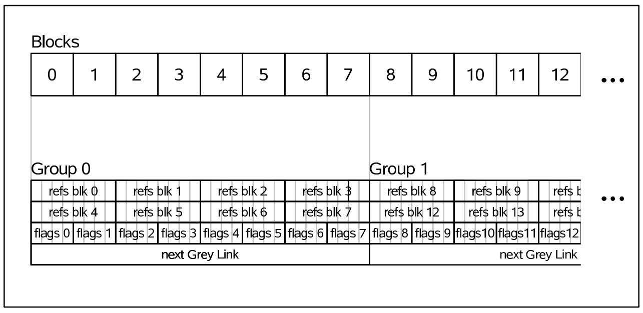

Figure 7: Group

With a dynamically growing heap, an important decision to be made by the implementation is when to request more memory from the operating system to expand the heap. It has to be ensured that the amount of heap requested does not exceed the amount of heap actually required by the application by a large factor, while it also has to be ensured that the heap is sufficiently large to obtain a good performance of the memory management.

Using the automatic pacing presented in the previous section, a trade-off between guaranteed heap use and maximum garbage collection work can be found easily: If the garbage collection work exceeds a certain threshold, a lower bound for the amount of memory used by the application can be determined using the graph in Figure 4. JamaicaVM 3.0 uses $P_{max} = 36$ garbage collection work units as this threshold, which guarantees that at least 81.7% of the heap memory will be used by the application before the heap will be expanded.

6. WRITE-BARRIER CODE

The write-barrier has to be used to ensure that the strong tricolour invariant:

There are no pointers from a black block to a white block

always holds. There are two main situations that require execution of the write-barrier code to ensure this invariant holds: First, at each store of a reference into a *black* block and second, during a mark step of the garbage collector that converts the colour of the scanned block from *grey* to *black*, which requires no block referenced by the scanned block to be *white*.

For any write of a reference that refers to a *white* block into any block on the heap, the written reference is added to the list of *grey* blocks before the write is performed. Since the write-barrier shades an object from *white* to *grey*, this operation will be called shading.

The colour of the block a reference is stored into is not taken into account by the write-barrier, only the reference value that will be stored will be examined and shaded in case it is *white*. This is possible since shading ensures that the stored reference will never point to a *white* block such that the tricolour invariant will remain valid, no matter what the colour of the block that is the target of the write is at this point.

The original shading code required for writing a reference to a field $a.f = r$ can be illustrated as follows:

```
1: if ((r != null) && (colour(r) == white))
2:   {
3:     colour(r) = grey_list;
4:     grey_list = r;
5:   }
6: a.f = r;
```

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

In this original code, the colour of each block was stored in parallel to the array of blocks itself, i.e., for each block on the heap, there is one slot in the colour-array. The index of the colour-slot in the colour-array is the same as the index of the block in the blocks array. Since there used to be only one contiguous memory range for all blocks, determining the index of a block in this range and finding the corresponding slot in the colour-array could be done via very efficient address arithmetic.

An important advantage of having the colour-information separate from the blocks is that the blocks themselves are fully available for the application use. In particular, inner blocks in arrays as in Figures 3 can use all 32 bytes for array data.

To be able to support multiple chunks for a dynamically growing heap, a new solution had to be found since each chunk needs its own array of blocks and colours. This would make it necessary to replace the simple pointer arithmetic for the determination of the colour-slot by more complex code that would first determine the chunk a block resides in. Since the write-barrier code is essential for the performance of the system, making this code more complex was not considered a feasible option.

Instead, one important observation was made: The shading code is most frequently executed by code that works with references to objects or arrays's first blocks. In particular, when references are saved for constant-time root scanning or when an assignment of a reference to a field or array element is made, the write-barrier code needs to be executed on references pointing to these *head blocks*.

The only situation when shading code is executed for references to *inner blocks*, i.e., allocated blocks that are not head blocks, is during the garbage collector's mark phase. Shading of inner blocks consequently occurs less frequently than shading of head blocks. For a high runtime performance, it is therefore essential to optimise the frequent case of shading a head block.

The total number of inner blocks is typically larger than that of head blocks. This has the implication that it pays off to reduce the memory required to store the colour of an inner block, even if this might be paid by poorer runtime performance of shading inner blocks.

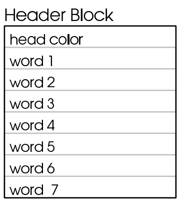

Figure 8: Header Block

For an efficient implementation of the shading of a head block, it was decided to allocate one slot per object for the colour (Figure 8). This slot resides in the head block of each object at the same offset. Using this slot, there is no longer any need for the colours array, the determination of this array's address and the chunk a head block resides in is no longer needed. The new and simplified shading code for head blocks looks as follows:

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

What is still needed is a means to shade inner blocks. Shading of inner blocks is required since the mark phase does not operate on whole objects, but incrementally on single blocks, such that parts of an object may be marked *black* while other parts have not been scanned by the garbage collector yet and are not *black* yet.

To store the colour of an inner block, two bits are required for each block. These bits may be set to one of the values 0 (*free*), 1 (*white*), 2 (*black*) or 3 (*grey*). One additional bit, the head-flag, is used to indicate whether the block is a head block or an inner block. The colour-bits are used only for inner blocks, the colour of a head block is instead stored in the colour field of that block itself.

The shading code for an inner block r now looks as follows:

```
if ((r != NULL) && (colourFlags(r) == white_flags))
  {
    colourFlags(r) = grey_flags;
  }
```

The determination of the colour-flags requires first the determination of the chunk that the block resides in. This operation is more expensive than the determination of the colour of a head block, but the cost is still not prohibitive.

In contrast to the shade code for head blocks, which uses a linked list of *grey* blocks, the shading of an inner block using only the colour flags does not permit the determination of a *grey* block in constant time. In the worst case, the garbage collector would have to scan the flags of all blocks to find one single *grey* block, causing the worst-case execution time to complete a garbage collection cycle to become quadratic. For a deterministic garbage collector, we require the completion of each cycle to be linear in the amount of allocated memory.

Therefore, we need to find a means to find a inner block that is marked *grey* in constant time. The solution that was implemented for JamaicaVM is that for each group of blocks that contains at least one *grey* inner block, the whole group will be added to a linked list of groups that contain *grey* inner blocks. The corresponding shading code for an inner block has to include updating this linked list:

```
if ((r != NULL) && (colourFlags(r) == white_flags))
  {
    colourFlags(r) = grey_flags;
    if (group(r)->nextGrey == NULL)
      {
        (group(r)->nextGrey = firstInnerGrey;
        firstInnerGrey = group(r);
      }
  }
```

7. SWEEP PHASE

The purpose of the sweep phase is to add all blocks that remained *white* after the mark phase to the free lists and to revert the colour of *black* blocks to *white*, such that the next mark phase can start with all allocated blocks being *white*.

Since, during the sweep phase, the data stored in the blocks is not touched, the performance of the sweep phase can be improved significantly if it operates only on the garbage collector data that is stored in the group structures and does not invalidate the cache by accesses to the actual blocks. Unfortunately, with the new write-barrier that uses the first word in header blocks to store the colour, it seems necessary write to this field to reset *black* header blocks to *white*.

JamaicaVM 3.0 avoids touching each block to revert colours from *black* to *white*. Instead of changing all header block's colour entries and all colour flags of inner blocks during the sweep phase, the colour is at first left unchanged during the sweep phase. Only at the end of the sweep phase, a *flip* occurs in which the meaning of *black* and *white* is swapped as originally proposed by Lamport [13] and used by many implementations [10, 3, 6, 11]. This poses a small additional overhead on the shading code, since now the values for *white* and *white_flags* are variables and not constants. However, this additional overhead was insignificant, while the improved cache behaviour of the sweep phase had a very significant effect on the performance of the sweep phase.

8. PERFORMANCE COMPARISON

8.1 Memory requirements

Table 1 illustrates the memory demand for garbage collector related data in JamaicaVM 3.0 compared to earlier

Table 1: Memory Demand for GC related data

Kind of data	JamaicaVM	JamaicaVM
	2.8	3.0
colour array	12.5000%	-
reference flags	3.1250%	3.1250%
group flags	-	1.5625%
group next link	-	1.5625%
head colour (average 2 blocks / object)	-	6.2500%
total	**15.6250%**	**12.5000%**

versions. For an application that has an average object and array size of two blocks, the memory overhead for garbage collector related data was reduced from 15.625% to 12.5%[4]. The reason for this reduction is the replacement of the colour array by the colour flags stored in the new group structure. Even though one word per object is now reserved for an efficient write-barrier for references to header blocks, the additional space required for this *head colour* entry is less than the memory saved by not having the colour array with a word for each block.

8.2 Runtime Performance

To analyse the runtime performance of the new garbage collector implementation, a Java version of the GCBench benchmark by John Ellis and Pete Kovac with modifications by Hans Boehm was used. This benchmark is a micro-benchmark that tests the performance of allocating trees of different depths. This benchmark is useful to measure pure GC performance since it eliminates most other disturbing factors that would influence the results when using real applications. The results on real applications may be different depending on the allocation behaviour of the application.

The benchmark was run on an Intel(R) Core(TM)2 Duo CPU T7300 running at 2.00GHz using openSUSE 10.2 (X86-64). The benchmark was run 25 times using JamaicaVM 2.8 with the previous garbage collector, JamaicaVM 3.0 with the new garbage collector and JDK 1.5 (build 1.5.0-b64) using the incremental garbage collector. For a fair comparison, the heap size was fixed such that the overall memory demand of the JVM was 27MB[5].

The resulting execution time in all runs is illustrated in Figure 9. The average performance of JamaicaVM 3.0 is 30% faster than that of JamaicaVM 2.8. In comparison, the average performance of JDK is another 20% better. However, JDK shows an extreme variation in the execution time, while both versions of JamaicaVM have minor variation only due to the benchmark being run on a Linux system with some background load.

When looking deeper into the results and comparing the different sub-tests of the GCBench benchmarks (Figure 10), it can be seen that the performance of JamaicaVM is largely independent of the structure of the trees created by the benchmark, while JDK has a very good average performance for small trees, but a poor performance for larger structures.

[4]There is additional memory overhead due to internal fragmentation and padding and due to the fact that the amount of reachable memory must be less than the total memory to obtain reasonable GC runtime performance as described in section 4.

[5]JDK could not execute the benchmark with less than 27MB of memory

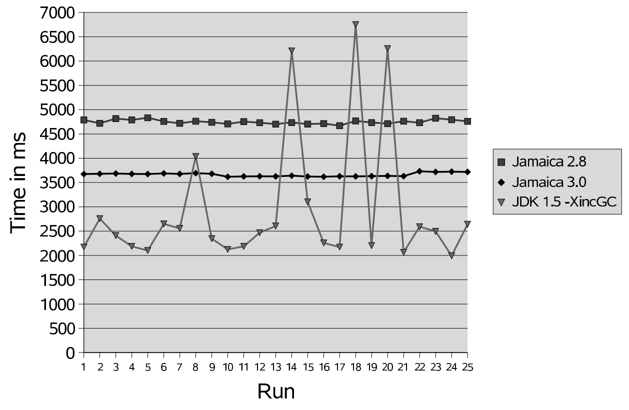

Figure 9: Runtime performance during 25 runs of GCBench.java.

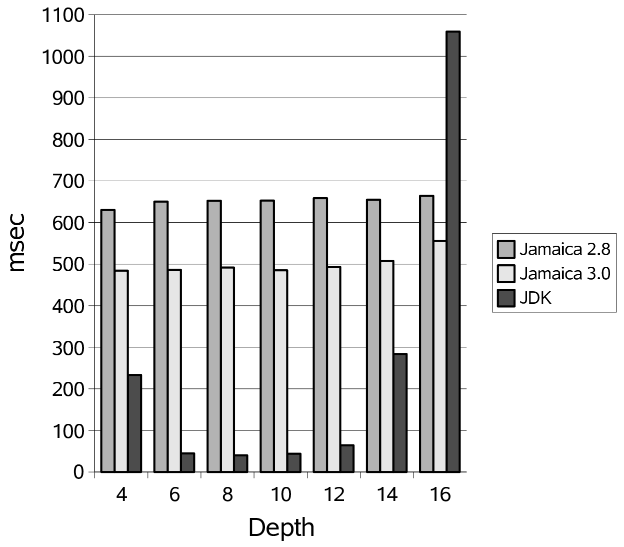

Figure 10: Performance in individual tests of GCBench.java

9. RELATED WORK

IBM's Metronome project [1] had the goal of producing a garbage collector with pause times that are as low as operating system context-switch times in the worst case while providing highly uniform CPU utilisation, low memory overhead, and low garbage collection cost. In contrast to Jamaica, the metronome scheduler is based on time rather than on work. It requires a specific read-barrier and it uses arraylets to break up large arrays to limit cost of scan, update or move operations on large objects. In addition to the heap size, the required input from the developer is the allocation rate and either the CPU utilisation or the space utilisation required by the application.

The PERC VM provides a garbage collector with preemption within 100 μsec [2, 14]. This incremental garbage collector is *paced*, i.e., it also requires the developer to provide the maximum allocation rate in addition to the heap size and the memory utilisation. Then, the scheduler can allocate sufficient CPU time for the garbage collector to reclaim memory fast enough to satisfy allocation requests up to the configured allocation rate.

Both approaches, Metronome and PERC, require more complex user configuration compared with JamaicaVM: JamaicaVM does not need the user to provide the allocation rate nor the heap utilisation of the application. Instead, it is sufficient to provide the heap size. Then, the maximum GC overhead can be determined depending on the memory utilisation (the maximum amount of reachable memory). The worst-case GC preemption time of JamaicaVM is about two orders of magnitude shorted since an atomic step cnsists of scanning only 32 bytes of memory.

10. CONCLUSION

The requirement to make JamaicaVM's realtime garbage collector easier to configure by introducing automatic heap expansions resulted in a major redesign of the internal data structures used by the garbage collector. This had an effect not only on the garbage collector implementation itself, but also on the write-barrier code executed by the interpreter and generated by the compiler.

The most important decision that was made was to implement a distinction between *header blocks* and *inner blocks*. A very efficient write-barrier is required for *header blocks*, since references to these blocks are subject to pointer store operations, while a less efficient write-barrier for *inner blocks* is tolerable.

The resulting implementation managed to reduce the memory required for garbage collector related information by about 20% (from 15.625% to 12.5% of the total heap memory) and to improve the runtime performance of the garbage collector by 30%.

The use of a realtime garbage collector becomes an efficient alternative even for application that do not have strict realtime requirements, but that profit from the lack of garbage collection pauses or larger increments of garbage collection work.

11. REFERENCES

[1] Metronome GC, http://domino.research.ibm.com/comm/research_projects.nsf/pages/metronome.metronomegc.html.

[2] Aonix North America, Inc. (2007). PERC products, http://www.aonix.com/perc.html.

[3] H. G. Baker. The Treadmill, real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, Mar. 1992.

[4] G. Bollela. *Real-Time Specification for Java.* Addison-Wesley, 2001.

[5] R. M. Corless, G. H. Gonnet, D. E. G. Hare, and D. J. Jeffrey. On Lambert's W function. preprint, 1993.

[6] J. DeTreville. Experience with concurrent garbage collectors for Modula-2+. Technical Report 64, DEC Systems Research Center, Palo Alto, CA, Aug. 1990.

[7] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46*. Springer-Verlag, New York, 1976.

[8] A. Diwan, J. E. B. Moss, and R. L. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of SIGPLAN'92 Conference on Programming Languages Design and Implementation*, volume 27 of *ACM SIGPLAN Notices*, pages 273–282, San Francisco, CA, June 1992. ACM Press.

[9] D. G. G. Salagnac, S. Yovine. Fast escape analysis for region-base memory management. In *Proceedings of the 1st Int. Workshop on Abstract Interpretation for Object-Oriented Languages (AIOOL'05)*, jan 2005.

[10] P. R. Hudak and R. M. Keller. Garbage collection and task deletion in distributed applicative processing systems. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 168–178, Pittsburgh, PA, Aug. 1982. ACM Press.

[11] L. Huelsbergen and P. Winterbottom. Very concurrent mark-&-sweep garbage collection without fine-grain synchronization. In Jones [12], pages 166–175.

[12] R. Jones, editor. *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, Vancouver, Oct. 1998. ACM Press.

[13] L. Lamport. Garbage collection with multiple processes: an exercise in parallelism. In *Proceedings of the 1976 International Conference on Parallel Processing*, pages 50–54, 1976.

[14] K. Nilsen. Starting to PERC. *Java Developer's Journal*, 1(2):11, July 1996.

[15] F. Siebert. Guaranteeing non-disruptiveness and real-time deadlines in an incremental garbage collector. In Jones [12], pages 130–137.

[16] F. Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Compilers, Architectures and Synthesis for Embedded Systems (CASES2000)*, San Jose, Nov. 2000.

[17] F. Siebert. Constant-time root scanning for deterministic garbage collection. In *Tenth International Conference on Compiler Construction (CC2001)*, Genoa, Apr. 2001.

[18] F. Siebert. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages.* aicas Books, 2002.

[19] M. Wallace and C. Runciman. An incremental garbage collector for embedded real-time systems. In *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.

APPENDIX

A. PROOF SKETCH FOR UPPER BOUND OF GC WORK ON ALLOCATION

This appendix presents a sketch of a proof for the upper bound P_{max} of GC work required to allocated one block of memory. We first assume that a GC cycle starts when the amount of allocated memory is $A0$. This GC cycle will finish after at most Q0 units of memory were allocated. To determine Q0, we need to solve this inequality:

$$A0 + Q0 \leq \sum_{i=1}^{Q0} P(M - A0 - i) \tag{1}$$

On the left side we have the total allocated memory, while on the right side, we have the amount of memory scanned during this GC cycle so far. When the scanned memory exceeds the allocated memory, the GC implementation guarantees that the cycle is finished. Solving this inequality, we get

$$Q0 \leq U(A0) \tag{2}$$

with

$$U(a) = M - A0 + W((A0/M - 1)/e) \tag{3}$$

where W is Lampert's W function [5].

An important feature of $U(A)$ is that it is monotonically falling for $A > 0.3M$, and that for all A

$$U(A) < M - A = F \tag{4}$$

which means, that no matter when a GC cycle starts, it will finish before we have run out of free memory.

Now, if we assume $A0 <= K$ at the beginning of one cycle, we cannot expect any garbage to be detected during this cycle ($G0 \geq 0$), since all allocated memory may be reachable at the beginning of the cycle and the GC cannot guarantee to find any garbage that will be created during the cycle. Hence, the amount of memory allocated at the beginning of the next cycle will be

$$A1 = A0 + Q0 - G0 \leq A0 + Q0 \leq K + U(K) \tag{5}$$

When the second cycle starts and $K < A1 \leq K + U(K)$, we are guaranteed to find garbage:

$$G1 \geq A1 - K > 0 \tag{6}$$

This second cycle will finish when the amount of GC work done exceeds the amount of allocated memory meanwhile A1 + Q1, i.e., when

$$A1 + Q1 \leq \sum_{i=1}^{Q1} P(M - A1 - i) \tag{7}$$

becomes true. Again, we can estimate

$$Q1 \leq U(A1) \tag{8}$$

And determine the amount of allocated memory $A2$ at the beginning of the next cycle:

$$A2 = A1 + Q1 - G1 \tag{9}$$

$$\leq A1 + U(A1) - G1 \tag{10}$$

$$\leq A1 + U(A1) - (A1 - K) \tag{11}$$

$$\leq K + U(K), \tag{12}$$

ie., the amount of allocated memory at the beginning of a cycle cannot exceed K + U(K). Assuming that memory is freed at the very end of the GC cycle only, the upper bound for A is $K + U(K)$ plus the memory allocated during this cycle:

$$A \leq K + U(K) + U(K + U(K)) =: A_{max} \tag{13}$$

This upper bound for A results in an upper bound $P_{max} = P(M - A_{max})$.