

Eliminating External Fragmentation in a Non-Moving Garbage Collector for Java

Fridtjof Siebert
IPD, Universität Karlsruhe
Oberfeldstr. 34B
76149 Karlsruhe, Germany
siebert@jamaica-systems.de

ABSTRACT

Fragmentation can cause serious loss of memory in systems that are using dynamic memory management. Any useful memory management system must therefore provide means to limit fragmentation. Today's garbage collector implementations often do this by moving objects in a way that free memory is non-fragmented. This paper presents a new object model that is based on fixed size blocks. The model eliminates external fragmentation without the need to move objects. A Java virtual machine and a static Java bytecode compiler that use this object model 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 moving collectors and unparalleled by current implementations.

1. INTRODUCTION

Allocation and deallocation of objects of different sizes in a system with dynamic memory management can cause memory fragmentation. Fragmented memory is free memory that cannot be used by the memory management system to satisfy an allocation request. One generally distinguishes between internal and external fragmentation. Internal fragmentation is memory lost due to the allocator's policy of object alignment and padding. External fragmentation is memory lost because free memory is non-contiguous in a way that an allocation request cannot be satisfied even though the total amount of free memory would be sufficient for the request.

The amount of memory lost due to external fragmentation can be very high, a few objects can prevent large amounts of memory from being used to satisfy larger allocation requests. If no measure against fragmentation is taken, the typical worst-case memory usage is the maximum amount of live data times

the number of allocation sizes [1]. However, actual applications typically cause a relatively low average loss due to fragmentation [2]. This result is nevertheless of little help for safety-critical systems that have to give performance guarantees. In such a system, a guaranteed upper bound on the amount of memory lost due to fragmentation is needed.

2. MOVING COLLECTORS

A common means to fight external fragmentation is to allow the garbage collector to move objects in a way that free memory is contiguous. Examples are compacting mark and sweep collectors like the one used in Sun's Java Development Kit [3], or two-space copying collectors that were first proposed by Fenichel and Yochelson [4] and later enhanced for real-time systems by Baker [5].

To avoid the need to update all references to an object that has moved, so called handles or *forwarding pointers* [6] are used and all accesses to the heap are performed via these indirect references causing additional runtime overhead.

Changing object addresses significantly complicates optimizing compilers since address computations for fields or array elements can become invalid by garbage collector activity.

3. FIXED SIZE BLOCKS

A different strategy to avoid external fragmentation is to divide the heap into a set of blocks of equal size. Small allocation requests can be satisfied by allocating a single block, while larger ones require a possibly non-contiguous set of several blocks to be used to access the required amount of memory. Blocks never move, and any free block is available for any allocation request.

Important questions are what size should be used for these blocks and how larger objects should be built out of several blocks. The next sections present structures for objects and arrays. A Java virtual machine [7] and a static Java compiler have been implemented and the performance of the SPECjvm98 benchmarks suite [8] was analysed using this implementation.

3.1 Building Objects out of Blocks

Java objects consist of a set of instance fields and a fixed number of words for virtual method table, type information, etc.

Instance fields are inherited through class extension. The subclass can add new fields to the set of inherited fields. The position of an inherited field should be the same in objects of the inheriting class as in objects of the parent class, such that field accesses can be performed by the same simple code independent of the dynamic type of a reference.

Since Java objects are typically very small (Dieckmann finds average sizes between 12 and 23 bytes per object [9]), a simple linked list of blocks can be used to represent Java objects of arbitrary sizes, where one word per block is reserved for the link. The first block will contain the required type information and the first fields. If a second block is needed, the link field will point to a second block containing more fields. More blocks can be used if needed.

For a fixed block size of 16 bytes and a word size of 32 bits, **Figure 1** shows the structure of an object with seven fields of one word each. Using this object layout, fields can be added in a child class without changing the position of inherited fields, so that an access to a field does not require knowledge about the actual class of the object.

The time $O(p)$ required to access a field in an object is linear in the position p of the field in the object, instead of constant $O(1)$ for a classical representation of objects. The position p and hence the access time can be determined statically. We will see that an average field access requires little more than a single memory access.

3.2 Building Arrays out of Blocks

Arrays can be very large, so representing arrays as linked lists would impose a very high cost on array accesses. The representation proposed here is a tree structure similar to the list structure proposed in [10]. The number of branches per node is the highest possible power of two permitted by the block size. The array data is stored in the leaf nodes only.

A Java array needs information on its type and length, which are stored in the array header. To ease the access to the elements, it is useful to also store the depth of the tree representation with each array. **Figure 2** illustrates an array of 11 elements of one word each, again for a block size of 16 bytes.

Using this tree structure gives us a performance in $O(\ln(\text{size}))$ for element accesses in an array with size elements. This might be shocking com-

pared to the traditionally constant cost, but a usually low upper bound for the access time can be found for any given system, e.g., in a system with 32-byte (8 words) blocks and a heap of 16MBytes, the tree depth will never exceed 7.

The code required to access an element in such an array can use a small loop to descend the tree to a leaf, a possible implementation in C-code is given in **Listing 1**.

The value node_width used here is a constant power of two specifying the number of words in a node of the tree. The code is for 16-byte blocks, but it can be changed to allow different sizes by changing the $\#define$'s. For a block size of 32 bytes the constants node_width and $\log_2 \text{node_width}$ have to be 8 and 3, respectively.

The array access code can be implemented efficiently in machine code. As an example, **Listing 2** shows the assembler-code that could be produced for the ARM processor [11]. The code is sufficiently short to be inlined, avoiding additional call overhead for array accesses.

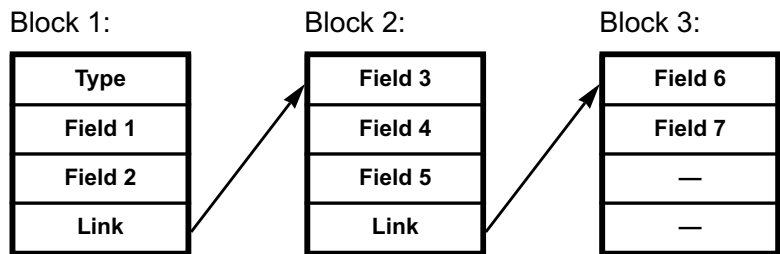


Figure 1: Object with 7 fields composed out of three blocks of 16 bytes each.

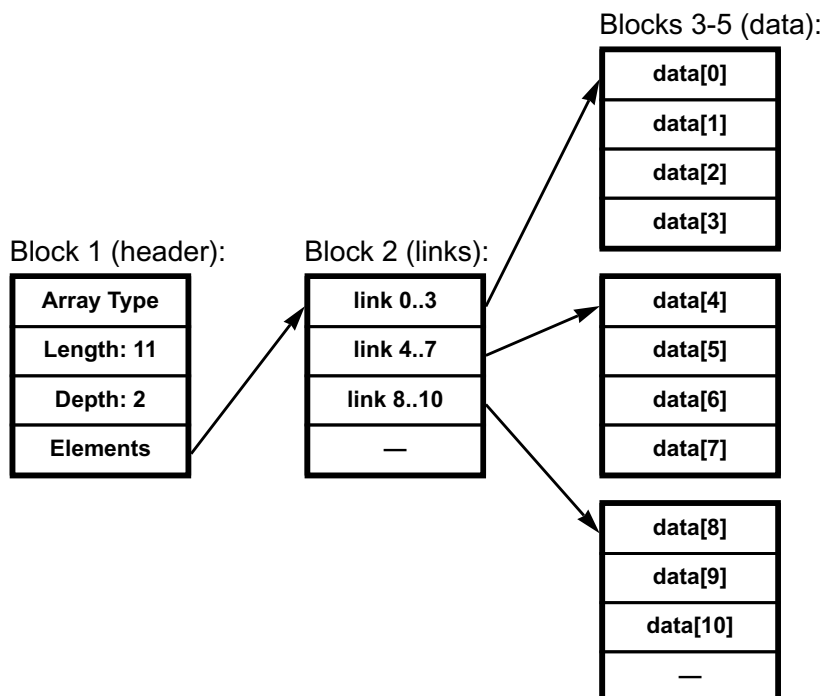


Figure 2: Tree representation of an array of 11 word elements composed out of five blocks of 16 bytes.

3.3 Supporting More Efficient Contiguous Arrays

Even though accesses to arrays that are represented as trees can be implemented in a surprisingly efficient way, the frequent use of arrays in typical Java applications causes the array access code to be one of the most important performance bottlenecks of this approach. Since fragmentation is typically low [2], it would be preferable to use a linear representation for arrays whenever possible. Such a representation is possible when setting the *depth* field in an array to *zero* as shown in **Figure 3** for the array from **Figure 2**. The array access code shown above does not need to be changed, it is correct for both array representations. On an allocation of an array the linear structure can be used whenever a sufficiently large contiguous range of free blocks is available and can be found quickly enough. Else the tree representation can be used as a fall-back whenever the memory is too fragmented or searching for a suitable free range would be too expensive.

4. IMPLICATIONS OF THE USE OF FIXED SIZE BLOCKS

The use of fixed size blocks instead of a moving collector to avoid fragmentation has a number of important consequences for the implementation of a Java virtual machine and a compiler.

```
#define node_width 4      /* 4 words in a block */
#define log2_node_width 2 /* log2(node_width) */

word readArrayElement(block *array, int index) {
    block **ptr = &(array->elements);
    int d = array->depth * log2_node_width;
    while (d != 0) {
        int t = (index >> d);
        ptr = ptr[t];
        index = index - (t << d);
        d = d - log2_node_width;
    }
    return ((word *) ptr)[index];
}
```

Listing 1: C-Code to access array elements

```
-- rA    points to the array
-- rI    contains the index
-- rD, rT temporary values
-- rE    result

        ld     rD, [rA, #depth]    -- rD = rA->depth
        add   rE, rA, #elements    -- rE = &(rA->elements)
        adds  rD, rD, rD           -- rD = rD*log2_node_width
        beq   end                  -- if (rD==0) goto end
loop:   mov   rT, rI, LSR rD       -- rT = rI >> rD
        ld   rE, [rE, rT LSL #2]   -- rE = rE[rT]
        sub  rI, rI, rT, LSL rD    -- rI = rI - (rT << rD)
        subs rD, rD, #2           -- rD = rD - log2_node_width
        bne  loop                  -- if (rD!=0) goto loop
end:    ld   rE, [rE, rI LSL #2]   -- rE = rE[rI]
```

Listing 2: ARM-Code to access array elements

4.1 Updating References for Moved Objects

Moving garbage collectors are usually more complex to implement than non-moving ones. It is important to update all references with the new location of a moved object. This updating requires all reference variables to be known to and modifiable by the collector, while for non-moving schemes it is sufficient if the garbage collector finds one reference to each referenced objects, no matter how many other references to the same object may exist.

If this exact information is not available because, e.g., a compiler that is not garbage collection aware is used, conservative mechanisms have to be employed [12]. Objects must not be moved when a reference seems to exist to that object. In this case, defragmentation can only be partial, and fragmentation can still lead to unpredictable allocation failures.

A scheme using handles can be used to avoid the need to update all references. But this introduces a significant run-time overhead and conservatism might still be needed while direct accesses to objects are in use, e.g., by compiled code.

When fixed size blocks are used, objects are never moved, so references do not need to be updated by the collector. It is sufficient if the garbage collector finds one reference of each object that is in use. There is no need for handles, and the compiler is free to use direct references to objects without informing the garbage collector as long as the compiler assures that at

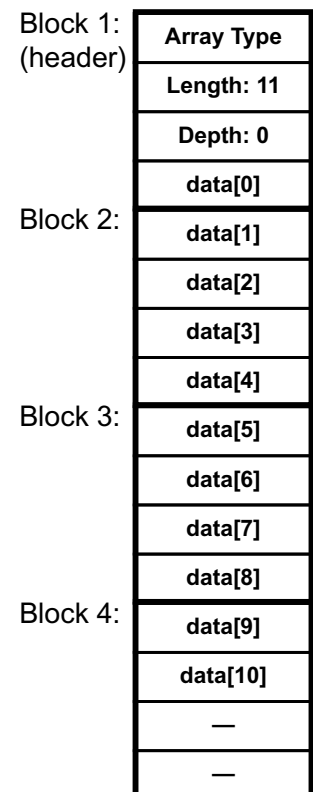


Figure 3: Contiguous representation of an array of 11 elements.

least one reference to each object that is accessed is known to the collector.

4.2 Dealing with Large Objects

An incremental moving collector typically has to move objects atomically. Since an object in a language like Java might be a large array, moving an object atomically will introduce a long pause that is not acceptable for real-time systems.

Incrementally moving an array as proposed in [13] is likely to impose an unacceptable run-time overhead and to complicate both the garbage collector and the compiler. Also, hardware-assisted garbage collection algorithms that incrementally move large objects have been proposed [14].

Using the system's memory management unit for defragmentation has been proposed [15], which might be a solution for some systems.

Another difficulty is scanning a large object for references during the garbage collector's mark phase. It might be desirable to scan one object atomically, and not be interrupted by mutator activity while an object is being scanned. For large objects, this would also introduce long pauses.

In the presented fixed size approach, scanning can be done block-wise. This means that at some point during garbage collection, parts of the same object might have been scanned by the collector, while others have been found to be reachable and yet others have not yet been touched by the garbage collector. Since blocks all have the same small size, the time for scanning a single block has a small worst-case upper bound. Since blocks are never moved, the problems due to moving large objects never occur in this case.

4.3 Ease of Garbage Collector Implementation and Verification

An exact garbage collector requires detailed information on the layout of objects, their sizes and the location of references within each object. Often, it needs access to programming language specific data to obtain this information.

When fixed size blocks are used, the garbage collector does not need to know about the size or structure of objects (as shown in **Figures 1** through **3**). All it has to care about are blocks. To store the information on the location of references in a block, an additional bit array is sufficient that has one bit for every word that indicates which words are references.

Complexity is taken from the garbage collector, the implementation is simpler to verify and easier to prove correct. However, it has to be noted that some complexity is added to the compiler that has to generate additional code for accesses to fields and array elements.

5. THE JAMAICA VIRTUAL MACHINE

Jamaica is a new implementation of a Java virtual machine and a static Java compiler that uses fixed size blocks to represent all dynamically allocated data structures. This includes not only Java objects, but also all internal data structures used by the implementation, e.g., structures for representing classes or method tables for dynamic binding. Arrays are allocated in the contiguous representation only if a free range that is sufficiently large can be found in constant time. The first free range in the free list and the largest free range found during the last garbage collection cycle are checked if one of them is large enough. Otherwise, the tree representation is used. The allocator never searches for a free range of a suitable size such that the allocation time is bounded.

The other aspects of the implementation's garbage collector are described in more detail in earlier publications [16], [17]. *Synchronization points* are used to limit thread switches and garbage collection activity to these points. Root scanning is done by explicitly copying all locally used references onto the heap to avoid the need to scan stacks and registers. 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 [18]. 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 parallel to the array of 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 [19]. 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 [20]. The generated C code is then translated into machine code by a C compiler such as *gcc*.

6. CHOOSING A BLOCK SIZE

When using blocks of a fixed size, the most important decision to be made is to choose the block size. It is not at all clear which size is best for typical Java applications; it is indeed very likely that different applications with different allocation behaviour perform best with different block sizes. The block size

used by Jamaica is therefore configurable, it can be chosen between 16 and 128 bytes at build time.

The run-time performance and heap requirements of seven benchmarks from the SPECjvm98 benchmark suite have been analysed using 17 different fixed block sizes. 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. The programs were then executed on a single processor (333 MHz UltraSPARC-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 [3], [21] and their just-in-time compilers have 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 (real-time vs. non-real-time garbage collection, static vs. just-in-time compilation, etc.).

6.1 Run-Time Performance

First, the run-time performance of the example programs has been measured for 17 different block sizes. The results are shown in **Figure 4**. For the analysis, the heap size was set to 32MB for most tests. Only for *compress*, *javac* and *mtrt* it was set to 64MB, 72MB and 48MB, respectively, since these tests required more memory for some block sizes.

Block sizes that are powers of two cause a significantly better performance than other sizes. The main reason is the simplification of accesses to blocks and entries in the colour-vector and reference-bit-vector, where shift operations can be used instead of slow multiplications and divisions. Additionally, blocks with a size that is a power of two can be aligned with the system's cache lines, reducing cache misses.

Very small block sizes cause bad performance since a smaller block size causes more frequent splitting of an object into several blocks.

For larger block sizes, the performance of some tests either remains more or less constant (*compress*, *mpegaudio*, *jack*), while for the allocation intensive tests (*jess*, *db*, *javac*, *mtrt*) the performance decreases when the block sizes are increased. Larger blocks cause wasting of more memory, which causes more garbage collection work to recycle sufficient memory.

For all benchmarks, the best performance is achieved when using a power of two block size, but the optimal size differs between the tests: 32 bytes is optimal for *jess*, *db*, *javac* and *mtrt*, while 64 bytes is best for *compress*, *mpegaudio* and *jack*.

Compared to Sun's implementation, the performance of most tests for a 'good' block size is similar to that of JDK 1.1.8 or 1.2, while the performance of JDK 1.2.2 was improved signifi-

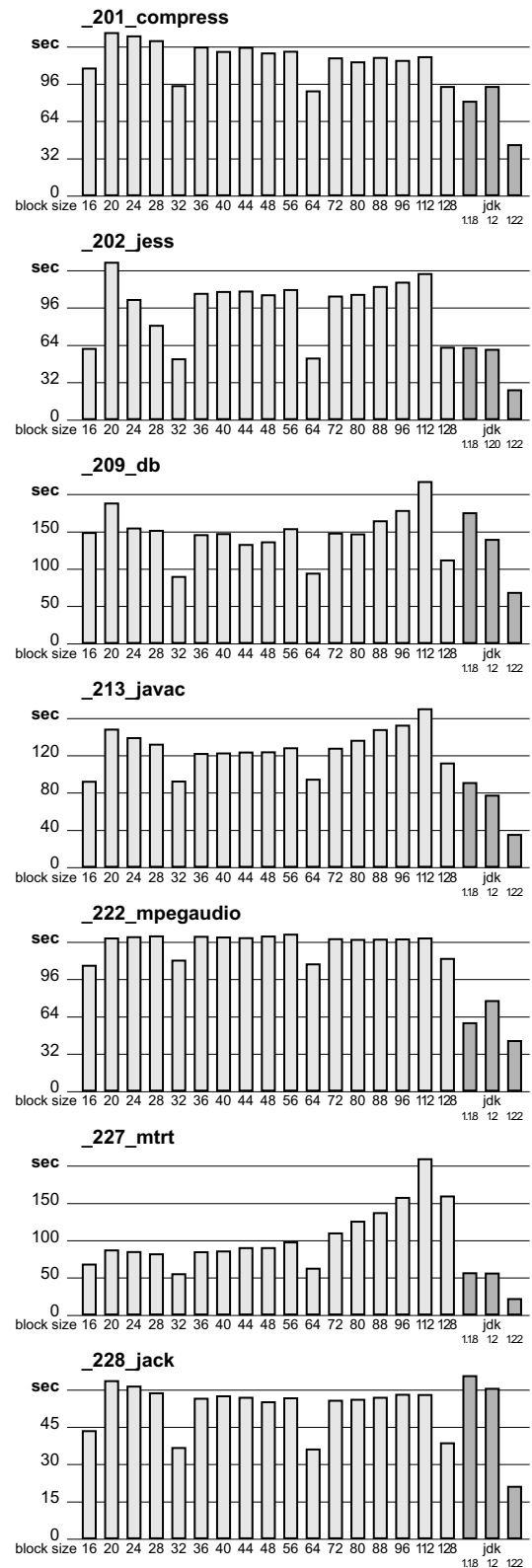


Figure 4: Run-time performance of the SPECjvm98 benchmarks using different block sizes

cantly. One can expect that better optimization will allow improvement of the performance of Jamaica as well.

6.2 Memory Performance

Next, the heap requirements of the test applications were analysed. A special option ‘*-analyse*’ of Jamaica instruments the garbage collector to run sufficiently aggressive to measure the required heap to a given accuracy. For all tests, this accuracy was set to 5%. During this analysis, all arrays are allocated in their tree representation, so that the result is the worst-case heap requirement for the case that no arrays could be allocated in linear space due to fragmentation. The results of the memory analysis are shown in **Figure 5**.

For the JDK’s, the minimum heap requirement was determined by gradually decreasing the options *-ms* and *-mx* until the application failed with an out-of-memory error.

For all test programs, the space for the objects increases with the fixed size that was chosen. The reason for this is that a vast majority of the objects allocated are very small, and bigger fixed sizes are of no use for these objects.

For most tests, the amount of memory required periodically drops at sizes that are powers of two. The reason is the tree representation of arrays that uses only the largest possible power of two number of words in each node or leaf of the tree. All the excess words are unused.

The smallest heap for most tests is possible with a block size of 32 bytes, the exceptions are the array-intensive tests *compress* and *mpegaudio* with the smallest heap for 128 or 64 byte blocks. In *mtrt* most objects that are allocated fit into a block of 20 bytes such that the smallest heap is attained using this block size.

6.3 Allocation and Memory Access Characteristics of the Benchmarks

To better understand the behaviour of the benchmark suite, the Jamaica compiler was instrumented to create additional code to collect information during the execution of the tests. First, the amount of memory allocated for objects and arrays was determined. For arrays, it was also recorded whether an array could be represented as a contiguous range of memory or whether the tree representation had to be used. The results are presented in **Figure 6**.

Some tests (*compress* and *mpegaudio*) allocate most of their memory for arrays, while the other tests also allocate significant amounts for objects. The memory allocated for arrays in tree representation is insignificant for most tests, only *compress* and *javac* allocate a larger fraction of their arrays as trees. This result shows that in the benchmarks, fragmentation is not high during the execution of the benchmarks. Even the constant-time test to find a suitable range of free blocks for an array allocation is typically successful.

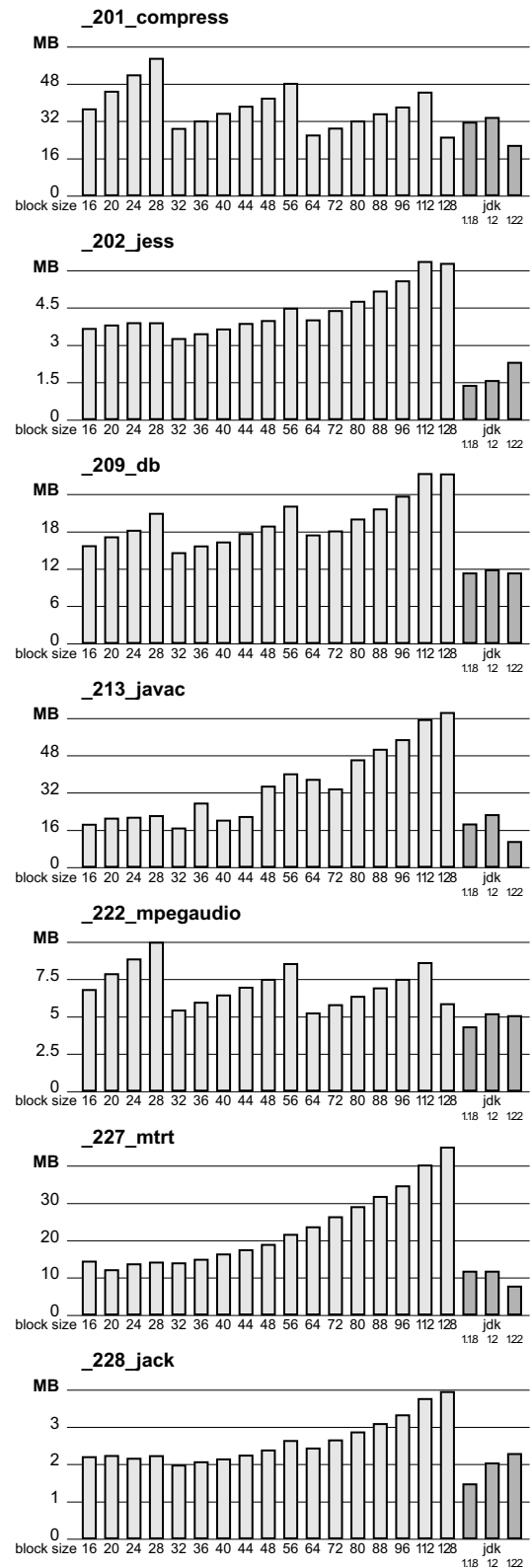


Figure 5: Minimum heap required for different block sizes

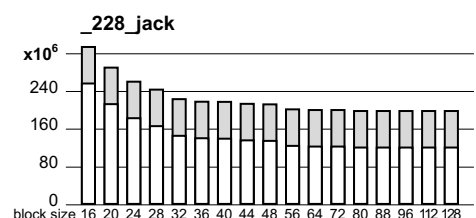
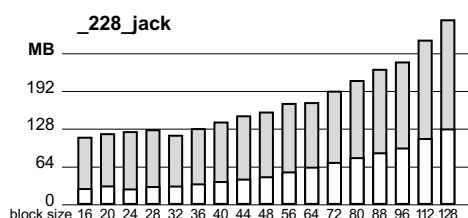
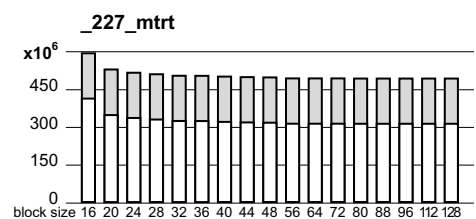
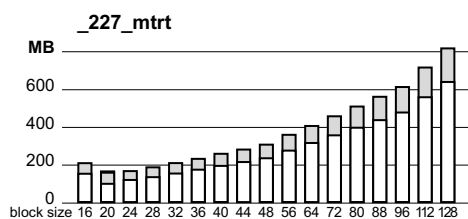
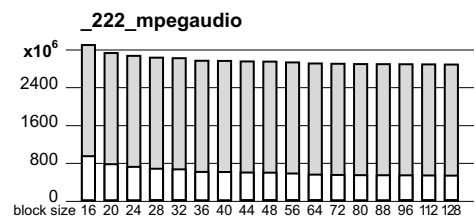
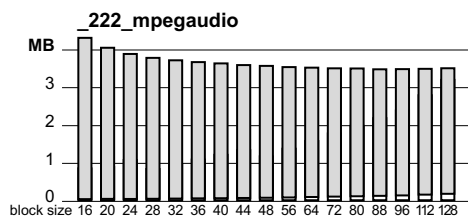
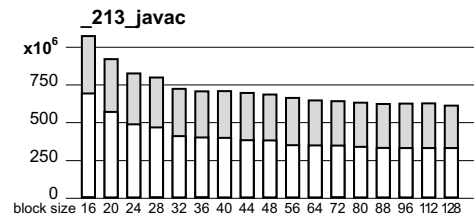
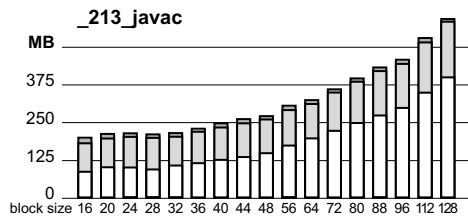
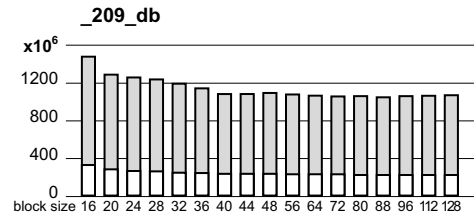
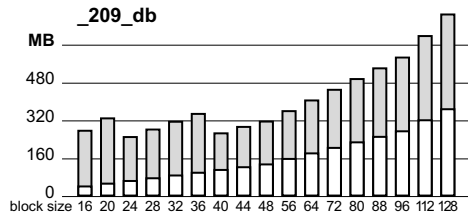
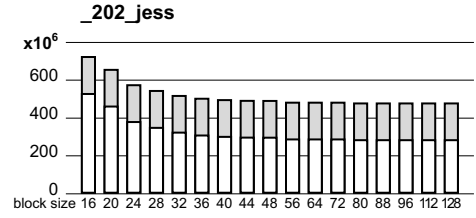
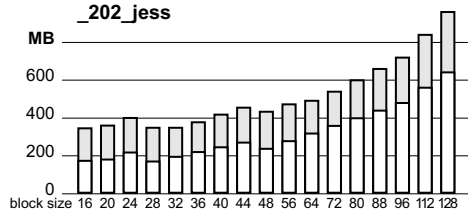
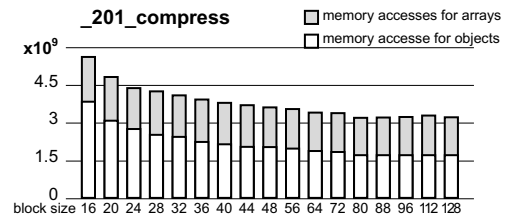
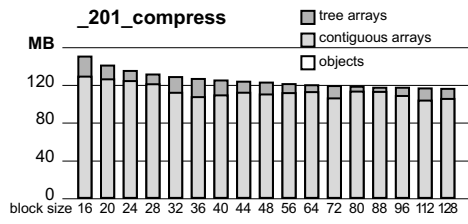


Figure 6: Amount of memory allocated for contiguous arrays, tree arrays and objects for different block sizes

Figure 7: Number of memory accesses performed for array elements and object fields using different block sizes

Next, the number of memory accesses required by the benchmarks to access objects and arrays on the heap was analysed. For an access to a field of an object, a single memory access is sufficient if the field resides in the first block used to represent the object. Two memory accesses are needed for fields in the second block (to read the link from the first block and to access the field itself), etc.

For accesses to arrays, the number of memory accesses required depends on the representation of the array: In contiguous representation, the array's depth needs to be read and checked, next the element itself can be accessed, so two memory accesses are needed. For an array in tree representation, the tree needs to be traversed. This traversal requires d memory reads for a tree of depth d . The depth and the element itself need to be accessed as well, so we get $2+d$ memory accesses in this case. **Figure 7** illustrates the total number of memory accesses required by the benchmarks. The number of memory accesses for objects changes significantly with the block size, while that for arrays is less affected by a change in the block size. Very small block sizes causes significantly more memory accesses for objects.

The average number of memory accesses required to access a field of an object or an array element was determined. The results are presented in **Figure 8**.

For object accesses, the average number of memory accesses is very close to 1 for most block sizes, only very small blocks below 32 bytes cause the average number of accesses to rise significantly, up to around 2 memory accesses for a block size of 16 bytes. The low average number of memory accesses is due to the typically small size of objects in Java. Additionally, when fields of large objects are accessed, those fields with small offsets tend to be accessed more frequently than those with larger offsets.

For most tests, most array accesses are to arrays in contiguous representation, so the average number of memory accesses is 2 or just above 2. Only *compress*, *db* and *javac* have a significant number of accesses to arrays in tree representation. For very small block sizes, the average number of memory accesses reaches values close to 3, but for blocks of at least 32 bytes this value remains close to 2.

If we compare these results to a memory management system that moves objects and uses handles, we see that object accesses need significantly fewer memory accesses in our approach: Close to a single memory access compared to 2 accesses to read the handle reference and the field itself. The average number of memory accesses required for array elements in our approach is slightly worse: Just above 2 accesses compared to exactly 2 when using handles.

6.4 A Good Standard Block Size

A block size of 32 bytes has a good run-time performance and heap size requirement for most cases, so it seems to be a good size for most applications. However, since some applications

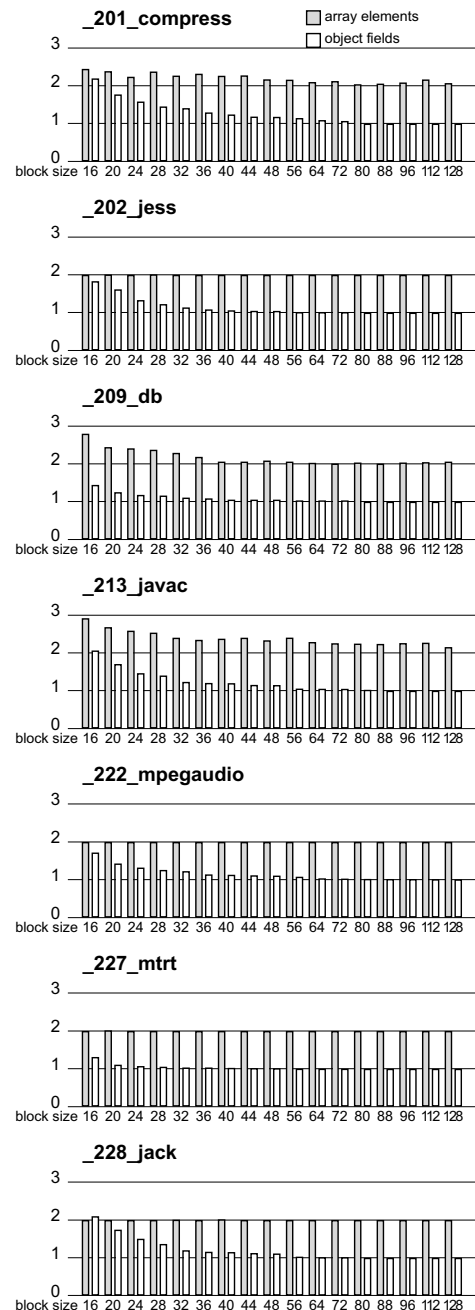


Figure 8: Average number of memory accesses required to access object fields and array elements for different block sizes

achieve their optimal run-time or heap size performance with different sizes, an implementation should allow selection of a different default size to best serve the application's needs.

Selection of a power of two is strongly recommended due to the better run-time and heap-size performance in most cases.

7. CONCLUSIONS

A new object model that uses fixed size blocks to avoid fragmentation has been presented. It has been applied to a new implementation of a Java virtual machine and a static Java compiler to analyse its behaviour. The results show that one can achieve performance that is comparable to that of current Java implementations. Furthermore, it has been shown that, using this object model, one can give hard real-time guarantees that are difficult to achieve with the traditional mechanisms to fight fragmentation, that the amount of reference tracing information needed is reduced, and that garbage collector implementation is simplified.

Even though the original goal was to provide deterministic behaviour for a hard real-time implementation of Java, the elimination of fragmentation and the simplified garbage collector increase the total reliability of the implementation, so that some applications that do not require hard real-time behaviour can be expected to benefit from it as well.

REFERENCES

- [1] Paul R. Wilson and Mark S. Johnstone: *Real-Time Non-Copying Garbage Collection*, ACM OOPSLA Workshop on Memory Management and Garbage Collection, 1993
- [2] Mark Stuart Johnstone: *Non-Compacting Memory Allocation and Real-Time Garbage Collection*, PhD Dissertation, The University of Texas at Austin, December 1997
- [3] *Java Development Kit 1.1.8*, SUN Microsystems Inc., 1999
- [4] Robert R. Fenichel and Jerome C. Yochelson: *A LISP Garbage-Collector for Virtual-Memory Computer Systems*, Communications of the ACM, Volume 12, 11, pp. 611-612, November 1969.
- [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] Rodney A. Brooks: *Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware*, Lisp and Functional Programming, pp. 256-262, ACM Press, 1984
- [7] Tim Lindholm, Frank Yellin: *The Java Virtual Machine Specification*, Addison-Wesley, 1996
- [8] *SPECjvm98 benchmarks suite, V1.03*, Standard Performance Evaluation Corporation, July 30, 1998
- [9] Sylvia Dieckmann and Urs Hölzle: *A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks*, 13th European Conference on Object-Oriented Programming (ECOOP'99), Lisbon, 1999
- [10] P. A. Rounce: *A Processor with List Structured Memory*, Proceedings Advanced Computer Technology, Reliable Systems and Applications, 5th Annual European Computer Conference, Bologna, May 1991
- [11] *ARM Architecture Manual*, Advanced Risc Machines Ltd, Prentice Hall, 1996
- [12] Joel F. Barlett: *Compacting Garbage Collection with Ambiguous Roots*, Digital Equipment Corporation, 1988
- [13] Danny Dubé, Marc Feeley and Manuel Serrano: *Un GC temps réel semi-compactant*, Journées Francophones des Langages Applicatifs, JFLA, Janvier 1996
- [14] Kelvin Nilsen: *Reliable Real-Time Garbage Collection of C++*, Computing Systems Vol 7, no. 4, 1994
- [15] Michael Hicks, Luke Hornof, Jonathan T. Moore, Scott M. Nettles: *A Study of Large Object Spaces*, International Symposium on Memory Management (ISMM'98), Vancouver, 1998
- [16] Fridtjof Siebert: *Real-Time Garbage Collection in Multi-Threaded Systems on a Single Processor*, Real-Time Systems Symposium (RTSS'99), Phoenix, December 1999
- [17] Fridtjof Siebert: *Hard Real-Time Garbage Collection in the Jamaica Virtual Machine*, Real-Time Computing Systems and Applications (RTCSA'99), Hong Kong, December 1999
- [18] Edsger 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
- [19] Fridtjof Siebert: *Guaranteeing non-disruptiveness and real-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>
- [20] Michael Weiss, François de Ferrière, Bertrand Delsart, Christian Fabre, Frederick Hirsch, E. Andrew Johnson, Vanial Joloboff, Fred Roy, Fridtjof Siebert, and Xavier 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
- [21] *Java Development Kit 1.2.2*, SUN Microsystems Inc., 2000