Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2016) 24: 398 – 411

doi:10.3906/elk-1304-206

A switchable approach to large object allocation in real-time Java

Veysel Harun ŞAHİN*, **Ümit KOCABIÇAK**
Department of Computer Engineering, Faculty of Computer and Information Sciences, Sakarya University, Sakarya, Turkey

Received: 22.04.2013 • **Accepted/Published Online:** 14.11.2013 • **Final Version:** 05.02.2016

Abstract: Over the last 20 years object-oriented programming languages and managed run-times like Java have been very popular because of their software engineering benefits. Despite their popularity in many application areas, they have not been considered suitable for real-time programming. Besides many other factors, one of the barriers that prevent their acceptance in the development of real-time systems is the long pause times that may arise during large object allocation. This paper examines different kinds of solutions that have been developed so far and introduces a switchable approach to large object allocation in real-time Java. A synthetic benchmark application that is developed to evaluate the effectiveness of the presented technique against other currently implemented techniques is also described.

Key words: Real-time systems, Java virtual machine, Ovm, Minuteman, garbage collection, memory management

1. Introduction

Object-oriented programming languages have spurred great interest in the programming world because of their software engineering benefits. The object-oriented programming paradigm enabled developers to model and represent real-world problems in computing systems with great success by introducing the object concept. Reusability, inheritance, and encapsulation are three main building blocks of object-oriented programming that helped to ease the construction and maintenance of large-scale projects.

One of the object-oriented programming languages, Java has become quite popular in the last two decades. There were two key factors of Java's popularity besides its implementation of the object-oriented programming paradigm. The first one was the rich set of class libraries that increases developer productivity. The second was the managed run-time, which takes on most of the developer duties like managing resources. One of the key benefits of using a managed run-time is its memory management system, called a garbage collector (GC) [1]. GCs handle all of the memory management tasks of the application. By doing this they free developers from writing memory management code and thus increase developer productivity. They also ensure memory safety by preventing the most encountered memory management bugs like memory leaks, dangling pointer bugs, and double-free bugs.

Despite its success in many application areas, Java has not been considered as an alternative programming language for real-time applications for a long time. Ironically, one of the key reasons for not using Java in real-time applications was one of its strongest points: garbage collection. Unlike traditional computer systems, a real-time system's correctness is determined not only by producing the correct results but also by producing them in a bounded amount of time. In other words, a real-time system must be deterministic. The execution

*Correspondence: vsahin@sakarya.edu.tr

time of each task and the deadlines should be known a priori. Each task must deliver its results within a bounded amount of time, which is determined by its deadline. On the contrary, GCs, by their nature, have nondeterministic timing behaviors. There are a number of uncertainties in a GC's work: a) the exact starting time of a garbage collection cycle, b) the time that is spent in a garbage collection cycle, and c) the response time to an allocation request not being known beforehand. The uncertainties mentioned in (a) and (b) are critical because in real-time systems we need to know the timing of tasks to make sure that the system will work consistently and the real-time threads will not miss their deadlines. As for (c), the response time to allocation requests directly affects the timing of each task that makes allocation. Because of this, the uncertainty mentioned in (c) has a direct impact on the timing of the real-time system. From the point of view of the GC, these uncertainties have been the main roadblocks before the adoption of Java for real-time applications.

Thanks to the extensive research that handles these problems, many solutions have been introduced for each area in the last decade. As a product of these studies, many real-time Java virtual machines have been successfully developed and used in real-time systems. This paper focuses on problem (c) in the context of Ovm [2], an open-source real-time Java virtual machine. In this paper we present a technique called the jumping search algorithm, which is a specialization of Boyer and Moore's [3] string search algorithm for contiguous object allocation and a switchable approach that aims to speed up the allocation times of large objects. We implemented our solution in the Minuteman RTGC framework [4–6], which is the GC framework of Ovm. For evaluation, we developed a benchmark application that allocates large-sized arrays and measures the allocation times. We ran our benchmark application for all the solutions that are currently implemented in Ovm in addition to our solution and then compared the results. To our knowledge this is the first study that specializes Boyer and Moore's algorithm in this context.

In the second section, background information on real-time Java is given and different approaches to large object allocation are explained. In the third section, we explain the Minuteman RTGC framework of Ovm in detail. Our technique is described in the fourth section, and in the fifth we share and interpret our benchmark results.

2. Real-time Java

To overcome the problems stated above and make Java a suitable environment for real-time application development, many solutions have been proposed and successfully implemented by researchers and engineers. The products of these studies are generally referred to as real-time Java [7]. In this section we explain different approaches to the real-time Java concept and describe well-known solutions to the abovementioned problems.

The real-time studies can be at the highest level categorized into two approaches, depending on their interpretation of the Java memory model. The first approach is based on the idea of limiting or completely disabling the GC and instead offering a region-based memory model in which developers would be able to manage memory themselves. Real-time specification for Java (RTSJ) [7–9] and safety critical Java (SCJ, JSR-302, Safety Critical Java Technology Specification) [7,10] are the two well-known examples of this approach. Region-based memory models are not examined further as they are beyond the scope of this paper. On the other hand, the second approach is based on the idea of developing GCs that enjoy predictable timing behaviors. GCs of this sort are called real-time garbage collectors (RTGCs).

The main theme of real-time garbage collection studies is to solve the ambiguity problems mentioned in the previous section and to make GCs more suitable for real-time application development. To be able to solve problems (a) and (b), precise timing behaviors should be brought into a collector's work. This is achieved by

applying different incremental garbage collection algorithms. RTGCs are in fact incremental garbage collectors [11].

The main aim of the incremental approach is to split the collection work into small increments and make the GC and application threads run sequentially. Unlike the stop-the-world [1] approach, incremental GCs do not perform the whole collection work in one step and do not block application thread(s) for long time periods. They do some amount of collection work in divided time intervals and then let the application thread(s) run between these intervals.

The key point when implementing an incremental approach to RTGC is to decide when to start collection work and how much time to spend on that work. Different scheduling strategies have been developed to be able to decide these timings. There are mainly three kinds of scheduling strategies. Slack-based scheduling was introduced by Henriksson [12] and used in Mackinac [13]. Time-based scheduling was introduced by Bacon et al. [14]. Lastly, the most well-known work-based scheduling was introduced by Baker [15] and used in Jamaica VM [16]. As this study is about the large object allocation we will not examine these strategies further.

On the other hand, for problem (c), there are studies that strive to improve the response time of allocation requests to a reasonable level that is suitable for real-time applications. One of the major hurdles when dealing with allocation requests is large objects. Finding a suitable location for large objects in a fragmented heap can be a very time-consuming process. During this long allocation period, real-time threads of the application may miss their deadlines. This is an unacceptable situation for real-time applications. Even worse, in the case of a heavy fragmentation, a suitable location for the object may not be found, thus forcing the RTGC to defragment the heap before allocation. Finding a suitable location for normal (small) objects can, however, be handled very efficiently by using bump pointer allocation with the help of segregated free lists as an example. For this reason, RTGCs mostly tend to use different allocation algorithms for each group of objects. Two different kinds of approaches are mainly used for large object allocation today.

In the fragmented allocation approach, large objects and arrays are split into small parts and every part of the object is considered as a normal object. Hence, it can be fairly easy to find a suitable location for each part of the object. The other advantage of this approach is to reduce the fragmentation by eliminating contiguous large object allocation, again resulting in efficient object allocation. Two well-known variants of this approach were presented by Bacon et al. [14] and Siebert [16].

Bacon et al.'s approach is based on the idea of splitting arrays into small parts called arraylets and then treating each arraylet as a normal object. In this approach, a pointer array-like structure called a spine is placed just after the header of the array and used to hold the address of each arraylet. Bacon et al.'s approach was implemented in Metronome RTGC [14] of the IBM Websphere real-time product [17]. The same approach was also implemented in the Minuteman RTGC framework of Ovm.

Siebert's approach is, however, more comprehensive. In this approach all objects larger than 32 bytes are treated as large objects and split into blocks of 28 bytes. For regular objects, the last word of each block stores the address of the next block. For arrays a tree-like structure is used to hold the address information. This approach is used in Jamaica VM. These two approaches were also implemented in Schism [18], a RTGC of Fiji VM [19,20].

These two variants of the fragmented allocation approach have the advantage of efficient memory allocation. On the contrary, access times to the fields of the objects would be slower because of the fragmented allocation of the object.

In the traditional contiguous allocation approach, all objects are placed in a contiguous memory area. The main disadvantages of this approach are the long allocation times and fragmentation stated above. Contrarily, access to the fields of the object can be handled very efficiently because of the contiguous placing of the object. Jamaica VM uses contiguous allocation for arrays whenever possible. Sun's RTS [21] uses contiguous allocation, but in case of fragmentation it switches to the fragmented allocation scheme. The Minuteman RTGC framework of Ovm also can be configured to use contiguous allocation.

It is clear that an RTGC that uses the contiguous allocation approach should have two key characteristics: efficient search algorithms for faster object allocation and support for defragmentation. Minuteman also has defragmentation support and can be configured to use it. As this study is about large object allocation we will not cover the details of defragmentation support of Ovm. On the other hand, for efficient memory allocation, one can prefer to use state-of-the-art allocators like binary-buddy [22–24], half-fit [25], Masmano et al.'s TLSF [26,27], or Doug Lea's (a memory allocator, Unix/Mail 1996) allocators. More information on memory allocators can be found in Wilson et al.'s survey [28]. A review of the most well-known allocators can also be found in [27].

This paper focuses on contiguous allocation and presents a switchable approach to large object allocation, partially based on Boyer and Moore's string search algorithm. We aim to achieve two key objectives with our approach: simple to implement and effective enough to improve the run-time performance of the allocator.

3. The Minuteman RTGC framework

The switchable approach presented in this paper was implemented in the Minuteman RTGC framework. Before going any further, readers might find it helpful to understand the principles of the operation of this framework. In this section we will give a brief introduction to Ovm and the Minuteman framework and then explain Minuteman's memory organization and memory allocators in detail.

Ovm is an open-source real-time Java virtual machine, mostly written in the Java programming language. It supports real-time garbage collection and also has an RTSJ implementation. It has been deployed on the ScanEagle UAV of Boeing Company [2]. Ovm features an ahead-of-time compiler that compiles virtual machine, application, and library codes to C code. Then the C code can be compiled using a stock C compiler generating the executable for the target platform.

Minuteman is the RTGC framework implemented in Ovm. The main purpose of Minuteman is to provide a stable base to implement and configure different kinds of GCs and therefore to enable the benchmarking of GCs on the same platform. Currently, different kinds of GCs can be configured on Minuteman by selecting different scheduling, defragmentation, barrier, array representation, and incrementality options. Because of its and Ovm's open-source nature, one can also implement his own solutions and GCs to the Minuteman framework.

3.1. Memory organization

Minuteman manages the heap by partitioning it into small parts called blocks. The size of the blocks is stored in a field in Minuteman and can be configured. The default value of the block size is 2048 bytes.

With the help of the heap size and block size information, Minuteman calculates the number of blocks that can be placed in heap. Then Minuteman creates a bit vector to store the usage information of each block. A set bit in the bit vector represents a used block, and a clear bit means a free block. This bit vector is defined as a 32-bit integer array named usedBits.

The representation of the blocks in the heap by using a bit vector is shown in Figure 1. In this figure, the first 10 elements of the usedBits bit vector, bit representation of the first element, and the corresponding blocks to these bits are illustrated by using sample values.

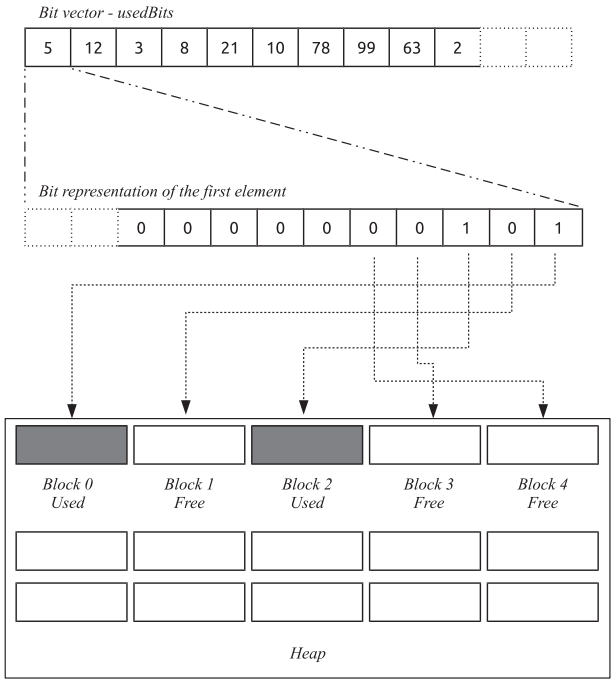

Figure 1. Memory organization of the Minuteman framework.

3.2. Memory allocators

A GC, if taken literally, might be considered as excluding memory allocation procedure. The truth, however, is different. Besides their garbage collection work, GCs are also responsible for memory allocation. One of the main duties of a GC is to find suitable locations in the heap for the newly created objects and return the addresses of the locations to the virtual machine.

Because a long response time to an allocation request may cause the real-time threads to miss their deadlines, memory allocation is also a critical piece of RTGCs. This is an unacceptable situation for most of the hard real-time applications, and for most of the soft real-time applications this behavior is undesired. RTGCs must respond to memory allocation requests in a reasonable amount of time.

To achieve effective memory allocation, Minuteman includes three allocators: a normal object allocator, a large object allocator, and an arraylet allocator. It uses segregated free lists for normal object allocations and linear search for large object allocations. For arrays, Minuteman can use contiguous or fragmented allocation approach. For contiguous allocation, Minuteman treats arrays as any other objects and therefore selects one of the first two allocators depending on the size of the array. As for fragmented allocation (arraylet allocator), Minuteman divides arrays into small pieces and treats each small piece as a normal object, and therefore employs segregated free lists. These three allocators are explained in detail in the following paragraphs.

Minuteman considers objects that can be placed in a single block as normal objects and thus it employs the normal object allocator for these objects. The normal object allocator uses a variation of segregated free lists. This variation is called sizeClass structure in Minuteman.

Whenever a normal object allocation request is received, Minuteman checks the sizeClass of the object and gets the block address. After getting the block address, the object is allocated in the block using bump pointer allocation. If the block is full, or if currently block information in the sizeClass is lacking, then a free block is picked and employed by that sizeClass. A representation of this approach is shown in Figure 2.

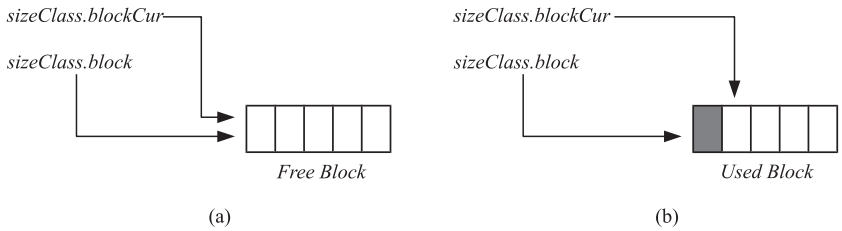

Figure 2. Size class and bump pointer representation.

For objects larger than the block size, Minuteman employs the large object allocator. This allocator first calculates the number of required blocks in order to place the object. Then it searches the usedBits bit vector for a number of adjacent clear bits equal to the calculated block count. This search operation is called linear search.

The large object allocator starts the search operation from the first element of the usedBits bit vector. It visits the elements of the bit vector one by one until it finds a clear bit. Upon finding a clear bit, it stores the index value of this bit. Then, starting from this index value, the allocator visits the elements of the bit vector one by one until it finds a set bit. Finding a set bit, it subtracts the index value of the clear bit from the index value of the set bit. By doing this the large object allocator finds the number of adjacent free blocks. If the number of adjacent free blocks is equal to or greater than the required block count for object, the start address of the first free block is returned as the address of the newly created object. Otherwise, the search operation continues starting from the next element of the set bit.

A graphical illustration of this search operation is shown in Figure 3. In this figure, a sample of the first 15 bits of the usedBits bit vector is shown. In this bit vector, a search operation of a large object that spans 4 blocks is illustrated.

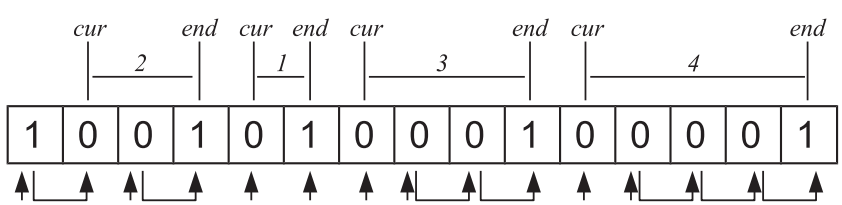

Figure 3. Large object allocation in Minuteman.

Minuteman can use two different approaches to array allocation: contiguous allocation and fragmented allocation. Minuteman can be configured so that it is potent to employ each of the approaches if need be.

In contiguous allocation, Minuteman places the whole array on one contiguous memory area. Before doing this it first checks the size of the array. If the size of the array is smaller than block size, the array is considered as a normal object, and the normal object allocator is employed. Otherwise, the array is considered as a large object, and in this case Minuteman switches to large object allocator.

In fragmented allocation, Minuteman uses arraylet allocator. This allocator is Minuteman's own implementation of the arraylet approach. In this approach, the array is split into equally sized pieces called arraylets.

The size of the arraylets by default is the block size. Therefore, each piece of the array is allocated using the allocation strategy for normal objects by employing the normal object allocator. To be able to access the array elements, the address of each piece is stored in a structure called a spine. The header of the array, the spine, and the last arraylet (if the size of the last arraylet is smaller than the block size) together are placed in a contiguous memory area. This type of arraylet is called the mixed form [17]. More information about arraylets can be found in [14] and [17]. If the size of this trio (header, spine, and last arraylet) is smaller than block size, the normal object allocator is used. If not, it is considered as a large object and the arraylet allocator employs the large object allocator. We should yet be aware that arraylet implementation of Minuteman does not always guarantee the use of the normal object allocator.

In Figure 4 contiguous and fragmented array representations are shown.

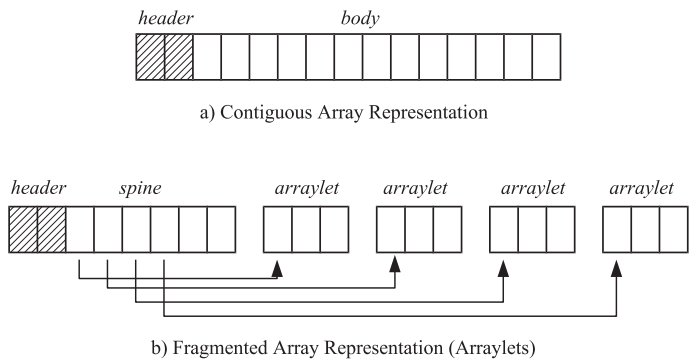

Figure 4. Array representations used in Minuteman.

4. The switchable approach

In this section we introduce the jumping search algorithm, which is a specialization of Boyer and Moore's string search algorithm for contiguous allocation and a switchable approach that combines the jumping search algorithm with linear search algorithm for efficient memory allocation.

The jumping search algorithm is an alternative of the linear search algorithm currently implemented in Minuteman. We had two key objectives with this algorithm: easy implementation that requires minimum modification to Minuteman code, and being effective enough to improve the run-time performance of the current linear search algorithm. Unlike linear search, the jumping search algorithm does not visit all the bits in the bit vector one by one. It performs the search operation by jumping a number of bits forward and then visiting the bits backwards.

The jumping search algorithm performs its search operation in two phases: the jumping phase and the reverse lookup phase.

Before starting the search operation, the required number of blocks for the object is calculated. Then the jumping phase starts. In this phase, the first element of the usedBits bit vector is visited. This is the start index of the jumping operation. This index value is called the starting point. Then the starting point is added to the required number of blocks for the object. The result is the end index value. We call this value the jumping point of the jumping operation. After gathering this information, our algorithm jumps to the element of the usedBits bit vector, which is at the jumping point. This operation is called the jumping operation.

Following the jumping operation, the reverse lookup phase starts. In this phase, all elements of the bit vector from the jumped point down to the starting point are visited one by one. During this visit operation a set bit is looked for. If a set bit cannot be found during this operation, this means the required number of

blocks for the object has been found. However, if a set bit is found, the reverse lookup phase is stopped and our algorithm switches to the jumping phase. This time the starting point of the jumping operation is set as the next element of the set bit. These operations continue until the required number of free blocks is found.

The jumping search algorithm is illustrated in Figure 5. In this figure, the first 15 bits of the bit vector are shown. In this bit vector, a search operation using the jumping search algorithm of a large object that spans 4 blocks is illustrated. In Figure 6, the pseudocode of the jumping search algorithm is shown.

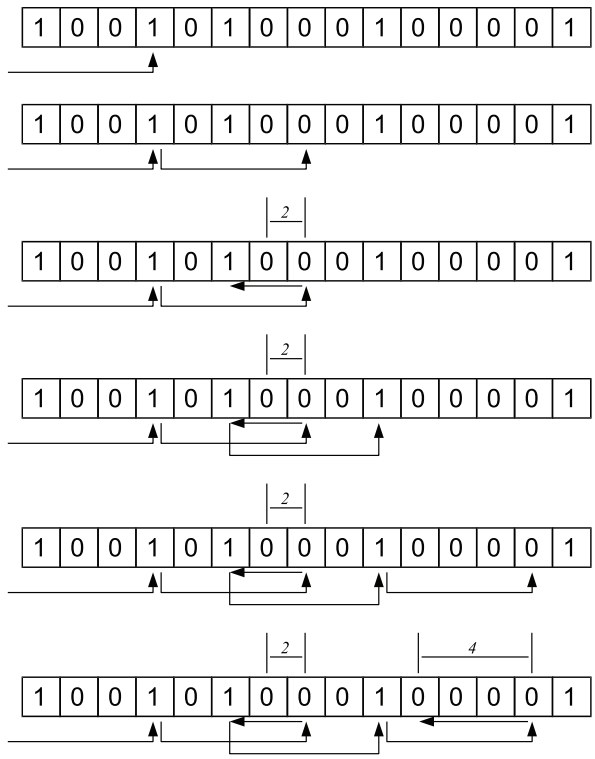

Figure 5. The jumping search algorithm.

Developing an algorithm that could replace the current linear search algorithm in Minuteman was our first intent in this study. However, in our early benchmarks we saw that, although the average allocation time performance of the jumping search algorithm was better than the linear search algorithm, the worst case execution time performance of the jumping search algorithm was worse. After a careful analysis, we found that the lower worst case execution time performance of the jumping search algorithm was occurring during the search of two adjacent free blocks. In other words, when searching two free blocks, the linear search algorithm was performing better. In order to increase both the average allocation time and worst case execution time performances, we therefore attempted to develop a switchable approach that is able to use both the linear search algorithm and the jumping search algorithm alternately.

Our switchable approach first calculates the required block count. If the block count is less than 3 blocks it switches to linear search. If not, it continues to work with the jumping search algorithm. The pseudocode of the switchable approach is shown in Figure 7.

5. Evaluation

To demonstrate the efficiency of the switchable approach we implemented it in the Minuteman framework. For the evaluations we developed a benchmark application and carried out tests of each approach in the Minuteman

framework. We then compared the allocation time performances of the switchable approach, the current linear search algorithm, and the arraylet approach. We also tested the array access times of each approach.

```
//Get total block count
elementCount = usedBits element count;

//Get required block count for the object
blockCount = required block count;

//Set starting point of the jumping operation
startIndex = 0;

//Set ending point of the jumping operation
endIndex = startIndex + blockCount - 1;

while (startIndex != (endIndex + 1)) {

    //Not enough free blocks found: Abort
    if (endIndex > elementCount)
        return error;


    if (usedBits[endIndex] == 1) { //Set bit found: Make jump
        startIndex = endIndex + 1;
        endIndex = startIndex + blockCount - 1;
    } else { //Clear bit found: Make reverse lookup
        endIndex = endIndex - 1;
    }
}

//Found free blocks
return startIndex;
```

Figure 6. Pseudocode of the jumping search algorithm.

```
blockCount = required block count;
if (blockCount >= 3)
    perform jumping search;
else
    perform linear search;
```

Figure 7. Pseudocode of the switchable approach.

Our experiments were run on an Intel Core i5 2.53GHz dual core machine with 4 GB of RAM, running Ubuntu Linux 12.04 with the 3.2.0-24 generic 32-bit kernel. Although our test machine is a dual core system, we ran our experiments on only one core because of the uniprocessor nature of Ovm. We achieved this by setting the CPU affinity of our application's process to only one core.

5.1. Benchmark

The benchmark application developed for these tests, a Java 1.4 compatible Java application, is a synthetic benchmark. Its main purpose is to allocate arrays and calculate the spent time in allocations. Our benchmark application allocates 1000 predefined and hardcoded sized arrays and reports the allocation time of each array. We preferred to use predefined sizes in order to create a similar memory usage pattern for all tested approaches. The sizes of the arrays were defined by using a random number generator before hardcoding them to our

benchmark application. By doing this we aimed to generate different loads for the allocator and different fragmentation patterns after each collection cycle and thus make a better measurement of the allocators.

Minimum array size of our benchmark application is 600 elements (2.34 KB) and maximum array size is 99229 elements (387.61 KB). The midrange of our array sizes is 194.97 KB. The size of the heap assigned to the application is 8 MB.

The bound on the allocable memory during a GC cycle that we used in this study was introduced by Robertz and Heriksson [29] and is shown in the following equation.

$a_{max} = \frac{H-L_{max}}{2}$ Here, a_{max} is the maximum amount of memory that we can allocate during a GC cycle without the risk of running out of memory. H is the heap size and L_{max} is the maximum amount of live memory. Besides Robertz and Henriksson's work, interested readers can also look at Schoeberl's [30] and Kalibera et al.'s [5] studies about schedulability analysis.

For our benchmark application we can ignore L_{max}, because the maximum live memory throughout the application lifetime is very small. It is the one object that allocates arrays. This means that in a GC cycle we can allocate a maximum 4 MB of data safely (a_{max} is 4 MB). Arrays are our objects that we will allocate during GC cycles. As our sample set consists of arrays whose sizes are distributed randomly, we chose to use the midrange of the array sizes as the base size. Depending on the midrange of the array sizes and a_{max} we calculated that we can safely allocate maximum 20 arrays in a GC cycle.

As a result of our calculations we configured our benchmark application to sleep in every 20th iteration of allocations to be able to give the GC some time for collection and to finish its cycle. In doing so we had two purposes: to get rid of the out-of-memory errors and to create a fragmented heap. By selecting this value we achieved our goal for our tests. However, one can choose to use the maximum array size instead of the midrange value to calculate the maximum amount of memory that can be allocated reliably during a GC cycle for different sample sets or for mission or safety critical real-life applications.

Besides allocation times, array access times are also critical data to see the differences between contiguous and fragmented array allocations. For this purpose, the benchmark also measures array access times for each array. After allocating an array and getting the allocation time, our benchmark application visits all the elements of the array and performs a store operation on them. It measures the spent time in this operation and divides the time according to the array size, and it calculates the average access time to a single element in the array.

Following the completion of the benchmark we have two kinds of information for each of the 1000 arrays: allocation time and array access time.

5.2. Results and discussion

In this study we ran this experiment for each of the three approaches. The first approach was to use contiguous array allocation with the current linear search algorithm, the second approach was to use contiguous array allocation with our proposed switchable approach, and the third approach was to use arraylet representation. For each approach we ran our benchmark three times and used the arithmetic mean of the gathered data from these runs. After doing this we got the allocation times and array access times of 1000 arrays.

The distribution of the allocation times of each approach is seen in Figures 8–10. In Figures 11–13, the histogram of the allocation times for each approach is given. Average allocation time for linear search is 119 μs, for the switchable approach is 36 μs, and for the arraylet representation is 44 μs. As can also be seen in Figures 8–10, these results show that the switchable approach increased the average allocation time performance of linear search to the level of arraylet representation. The frequencies of allocation times shown in Figures

11–13 demonstrate the improvement of the presented technique in a more clear way. This performance gain of the switchable approach over linear search can also be verified by the median values of the execution times. The median execution times for linear search, switchable approach, and arraylet representation are 135 μs, 25 μs, and 26 μs, respectively. The worst-case execution time of the switchable approach is 37% faster than linear search and 38% faster than arraylet representation. These results show that the worst-case execution time performance of the switchable approach is better than both linear search and arraylet representation. The allocation times of each approach are given in Table 1.

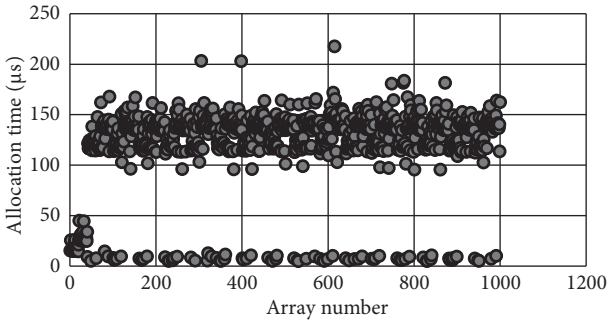

Figure 8. Distribution of allocation times (linear search).

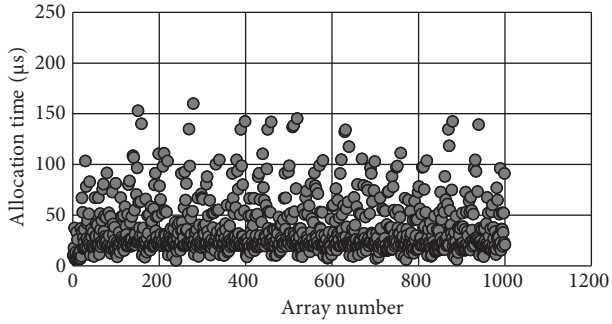

Figure 9. Distribution of allocation times (switchable approach).

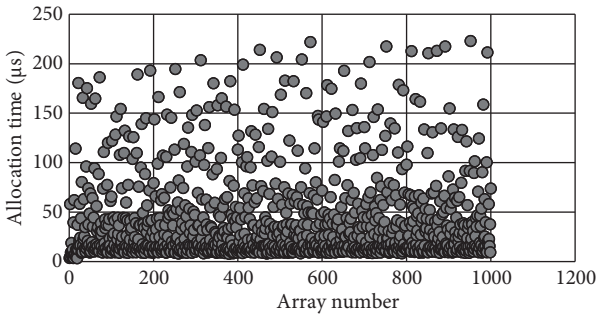

Figure 10. Distribution of allocation times (arraylet representation).

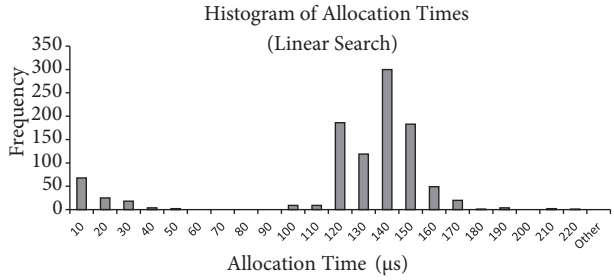

Figure 11. Histogram of allocation times (linear search).

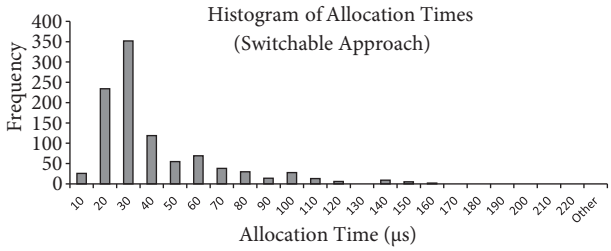

Figure 12. Histogram of allocation times (switchable approach).

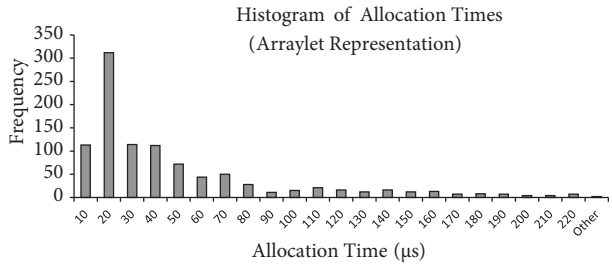

Figure 13. Histogram of allocation times (arraylet representation).

Depending on these results and our early benchmarks we can say that the jumping search algorithm alone performs better than linear search on average-case allocation times and performs worse than linear search on worst-case allocation times. However, with the combination of the two algorithms, the switchable approach performs better than linear search both on average- and worst-case allocation time performances.

Table 1. Allocation times of three approaches.

Approach	Average (μs)	Median (μs)	Worst case (μs)
Linear search	119	135	217
Switchable	36	25	159
Arraylets	44	26	222

The time complexity of our switchable approach is $O(n)$. This is the same as the current linear search algorithm. Although time complexities of the two approaches are the same, these results clearly show that the run-time performance of our approach is better. What is surprising in these results is that the arraylet representation performed worse than our switchable approach on average-case and worst-case allocation times. Again the arraylet representation performed worse than the linear search on worst-case allocation time.

Looking at the philosophy of the arraylet approach one can easily expect better results from it. When we analyzed our benchmark data closely we saw that most of the worst-case allocation times of arraylet representation were generated because of a switch operation to the large object allocation strategy (switchable approach). As we explained above, Minuteman uses the mixed form of arraylet representation. In this form, the header, spine, and last arraylet (if the last arraylet is smaller than the block size) are together placed on a contiguous memory area. Whenever the sum of these trio is greater than the block size, the arraylet allocator of Minuteman switches to the large object allocation strategy. In our test case for 164 arrays, the block size was exceeded and our large object allocation strategy (switchable approach) was run. In addition, for each arraylet, the normal object allocation strategy was run. Thus, the worst case performances of arraylet representation are the results of the performance of the switchable approach and the normal object allocation strategies. When we excluded the data of these 164 arrays from our results the worst-case allocation time of arraylets decreased to 142 μs, which is better than the other two approaches. We did not do a new comparison for average allocation times because our result set had changed. Based on these observations we can say that a more efficient arraylet implementation should perform better than contiguous allocation.

In Table 2, average array access times of each approach are shown. We can easily say that the average array access times of the switchable approach and linear search performed well compared to arraylet representation. This is because the switchable approach and linear search use contiguous allocation, while arraylet representation uses fragmented allocation. For contiguous allocation, access to an array element can be handled by simple pointer arithmetic. As for the fragmented allocation, access to an array element becomes a little bit more complex and time-consuming process.

Table 2. Average array access times.

Approach	Time (ns)
Linear search	27
Switchable approach	26
Arraylets	124

6. Conclusion

This paper presented a switchable approach to large object allocation in real-time Java partially based on Boyer and Moore's string search algorithm. The presented approach is realized by implementing it in the Minuteman framework of Ovm, a real-time Java virtual machine. We also presented a synthetic benchmark application to evaluate the large object allocation and array access performances. We have successfully evaluated three

different approaches to large object allocation on Ovm by using our synthetic benchmark. We have shown that the switchable approach has increased the performance of the large object allocator. We have also observed that although allocation times of fragmented allocation approaches could be better than contiguous allocation approaches, the average array access times of fragmented allocation are slower than those of contiguous allocation in Ovm.

Although the proposed approach is applicable to other real-time Java virtual machines, our work was realized on Ovm for two reasons. The first is that the BSD licensed open-source code of Ovm gave us the opportunity to freely get and modify the virtual machine code. The second is the highly configurable Minuteman RTGC framework, which allowed us to configure and test different object representations.

References

[1] Jones R, Hosking A, Moss E. The Garbage Collection Handbook: The Art of Automatic Memory Management. Boca Raton, FL, USA: CRC Press, 2011.

[2] Armbruster A, Baker J, Cunei A, Flack C, Holmes D, Pizlo F, Pla E, Prochazka M, Vitek J. A real-time Java virtual machine with applications in avionics. ACM T Embed Comput S 2007; 7: 5:1–5:49.

[3] Boyer RS, Moore, JS. A fast string searching algorithm. Commun ACM 1977; 20: 762–772.

[4] Kalibera T, Pizlo F, Hosking AL, Vitek J. Scheduling hard real-time garbage collection. In: Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS'09); 1–4 December 2009; Washington, DC, USA. Washington, DC, USA: IEEE Computer Society. pp. 81–92.

[5] Kalibera T, Pizlo F, Hosking AL, Vitek J. Scheduling real-time garbage collection on uniprocessors. ACM T Comput Syst 2011; 29: 8:1–8:29.

[6] Baker J, Cunei A, Kalibera T, Pizlo F, Vitek J. Accurate garbage collection in uncooperative environments revisited. Concurr Comp-Pract E 2009; 21: 182–196.

[7] Higuera-Toledano MT, Wellings AJ, editors. Distributed, Embedded and Real-Time Java Systems. New York, NY, USA: Springer, 2012.

[8] Bollella G, Brosgol B, Dibble P, Furr S, Gosling J, Hardin D, Turnbull M. The Real-Time Specification for Java. Boston, MA, USA: Addison-Wesley, 2000.

[9] Wellings AJ. Concurrent and Real-Time Programming in Java. West Sussex, UK: John Wiley & Sons, 2004.

[10] Plsek A, Zhao L, Sahin VH, Tang D, Kalibera T, Vitek J. Developing safety critical Java applications with oSCJ/L0. In: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'10); 19–21 August 2010; Prague, Czech Republic. New York, NY, USA: ACM. pp. 95–101.

[11] Jones R. Dynamic memory management: challenges for today and tomorrow. In: Proceedings of the International Lisp Conference (ILC'07); 1–4 April 2007; Cambridge, UK. Cambridge, UK: The Association of Lisp Users. pp. 115–124.

[12] Henriksson R. Scheduling garbage collection in embedded systems. PhD, Lund University, Lund, Sweden, 1998.

[13] Bollella G, Delsart B, Guider R, Lizzi C, Parain F. Mackinac: Making Hotspot™real-time. In: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05); 18–20 May 2005; Seattle, WA, USA. Washington, DC, USA: IEEE Computer Society. pp. 45–54.

[14] Bacon DF, Cheng P, Rajan VT. A real-time garbage collector with low overhead and consistent utilization. SIGPLAN Not 2003; 38: 285–298.

[15] Baker HG Jr. List processing in real time on a serial computer. Commun ACM 1978; 21: 280–294.

[16] Siebert F. Realtime garbage collection in the JamaicaVM 3.0. In: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES'07); 26–28 September 2007; Vienna, Austria. New York, NY, USA: ACM. pp. 94–103.

[17] Auerbach J, Bacon DF, Blainey B, Cheng P, Dawson M, Fulton M, Grove D, Hart D, Stoodley M. Design and implementation of a comprehensive real-time java virtual machine. In: Proceedings of the 7th ACM & IEEE international conference on embedded software (EMSOFT'07); 1–3 October 2007; Salzburg, Austria. New York, NY, USA: ACM. pp. 249–258.

[18] Pizlo F, Ziarek L, Maj P, Hosking AL, Blanton E, Vitek J. Schism: fragmentation-tolerant real-time garbage collection. SIGPLAN Not 2010; 45: 146–159.

[19] Pizlo F, Ziarek L, Vitek J. Real time java on resource-constrained platforms with Fiji VM. In: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'09); 23–25 September 2009; Madrid, Spain. New York, NY, USA: ACM. pp. 110–119.

[20] Pizlo F, Ziarek L, Blanton E, Maj P, Vitek J. High-level programming of embedded hard real-time devices. In: Proceedings of the 5th European conference on Computer systems (EuroSys'10); 13–16 April 2010; Paris, France. New York, NY, USA: ACM. pp. 69–82.

[21] Bruno EJ, Bollella G. Real-Time Java Programming: With Java RTS. 1st ed. Upper Saddle River, NJ, USA: Prentice Hall, 2009.

[22] Knowlton KC. A fast storage allocator. Commun ACM 1965; 8: 623–624.

[23] Knuth DE. The Art of Computer Programming, Volume 1: Fundamental Algorithms. 3rd ed. Redwood City, CA, USA: Addison Wesley, 1997.

[24] Peterson JL, Norman TA. Buddy systems. Commun ACM 1977; 20: 421–431.

[25] Ogasawara T. An algorithm with constant execution time for dynamic storage allocation. In: Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications (RTCSA'95); Tokyo, Japan. Washington, DC, USA: IEEE Computer Society. pp. 21–25.

[26] Masmano M, Ripoll I, Crespo A, Real J. TLSF: A new dynamic memory allocator for real-time systems. In: Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04); 30 June–2 July 2004; Catania, Italy. Washington, DC, USA: IEEE Computer Society. pp. 79–86.

[27] Masmano M, Ripoll I, Real J, Crespo A, Wellings AJ. Implementation of a constant-time dynamic storage allocator. Softw Pract Exper 2008; 38: 995–1026.

[28] Wilson PR, Johnstone MS, Neely M, Boles D. Dynamic storage allocation: a survey and critical review. In: Proceedings of the International Workshop on Memory Management (IWMM'95); 27–29 September 1995; Kinross, UK. London, UK: Springer-Verlag. pp. 1–116.

[29] Robertz, SG, Henriksson R. Time-triggered garbage collection: robust and adaptive real-time GC scheduling for embedded systems. In: Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03); July 2003; San Diego, California, USA. New York, NY, USA: ACM. pp. 93–102.

[30] Schoeberl M. Scheduling of hard real-time garbage collection. Real-Time Syst 2010; 45: 176–213.