A Comparison of Fixed and Variable Block Allocation in two Java Virtual Machines

by

Markus Goffart, Gerhard W. Dueck, and Rainer Herpers

TR 12-218, July 20, 2012

Faculty of Computer Science
University of New Brunswick
Fredericton, NB, E3B 5A3
Canada

Phone: (506) 453-4566
Fax: (506) 453-3566
Email: fcs@unb.ca
http://www.cs.unb.ca

Abstract

This paper compares the memory allocation of two Java virtual machines, namely Oracle Java HotSpot VM 32-bit (OJVM) and Jamaica JamaicaVM (JJVM). The basic difference of the architectures is that the JJVM uses fixed size block allocation on the heap. This means that objects have to be split into several connected blocks if they are bigger than the specified block-size. On the other hand, for small objects a full block must be allocated. The paper contains both theoretical analysis and experimental results on the memory-overhead. The theoretical analysis is based on specifications of the two virtual machines. The experimental analysis is done with a modified JVMTI Agent together with the SPECjvm2008 Benchmark. The results are summarized in a diagram. From this diagram, it can be seen which block size should be used for a given data size.

Contents

1 Introduction **2**

2 Methodology **4**
2.1 Architecture of Objects in OJVM and JJVM 4
2.2 JVMTI-Tool 7
2.3 Benchmarks 7

3 Results of SPECjvm2008-Benchmarks **9**

4 Optimal Parameter Setting **11**
4.1 Comparison of Overhead Between all Block-Sizes 11
4.2 Block-Size n for Specific Standard Objects 12
4.3 Block-Size n for Specific Array Objects 14
4.4 Which Block-Size for a Specific Program? 14

5 Conclusion **16**

6 Future Work **18**

Chapter 1

Introduction

A very important part of Java virtual machines is the garbage collector. It is responsible for the reorganization of the objects on the heap during the runtime of the JVM.

Two of the most recent developed and standard garbage collectors (GC) (i.e. Java virtual machines) that were developed for real-time applications, are the JJVM of the company *aicas* [1] and the *Concurrent Mark and Sweep Collector* (CMS) in the OJVM [13]. The JJVM tries to get short and bounded pause times in the first step in the GC-process in hard real-time systems [8]. This means that there are predictable execution times of stop-the-world phases. Whereas the OJVM avoids long stop-the-world phases with a concurrent garbage collector. This means that most of the garbage collection is done concurrently with the execution of the program. Most GCs need four steps to clean and compact the heap. These are the Root Scanning, Mark, Sweep, and Compaction. The basic difference between the garbage collectors of OJVM and JJVM is that the JJVM does not have a compaction phase. It becomes redundant due to the use of fixed-size blocks for the allocation of new objects on the heap [5][6]. However, the allocation of objects becomes more complex because the GC needs to track the free space and also has to split bigger objects into several fixed-size parts if an object is larger than the fixed-size block. Siebert measured the allocation time and could show that the time-overhead is low ($< 1\ \mu s$) [7].

Thus, the runtime of the JJVM seems to be quite good but what are the disadvantages is this case? The new architecture of the JJVM causes some added costs. First, there is wasted space due to the fact that the object may not fill an exact multiple of the fixed-size blocks. Second, links must be kept for split objects. These structures also affect the runtime because the GC may need to scan several blocks for one object and keep track of the additional information in chunks. Although Siebert conducted a first analysis on the JJVM, more research on this topic is necessary. He mentioned that the benchmarks used do not reflect real-time applications and this should be changed in future work.

This paper provides a detailed analysis of the JJVM memory allocation scheme. For this purpose, a profiler tool was implemented with the *Java Virtual*

Machine Tool Interface (JVMTI) [3] to analyze and compare the heap of the both virtual machines. This paper also describes how it is possible to find the best block-size n in the JJVM for any object.

The paper starts with some background information necessary to understand the basic differences between the object-sizes in JJVM and OJVM. Then, a short description of the modified JVMTI-profiler "hprof" that was used for the analysis follows. The two benchmarks that were used in this work are presented. After this, the results of the theoretical and experimental analysis are presented. The paper ends with some concluding remarks.

Chapter 2

Methodology

2.1 Architecture of Objects in OJVM and JJVM

Figures 2.1 and 2.2 show the architecture of objects in the OJVM and JJVM respectively. There is a difference between standard objects and array objects in both machines. In the OJVM, the standard object has a two words-header followed by the actual data, whereas the array-objects has a three words-header followed by the data. While the first two fields in both structures contain information such as hash-code, color-flag and a reference to the class of the object, the third field of an array object contains the number of elements of the array itself [4]. Together with the condition that the total size of the object must be a multiple of 8 bytes the total size for java objects in the OJVM is as follows:

$$\begin{aligned} sso_{OJVM} &= \left\lceil \frac{2 \cdot s_{mp} + s_{data}}{8} \right\rceil \cdot 8 \ [B] \qquad \text{(standard object)} \\ sao_{OJVM} &= \left\lceil \frac{3 \cdot s_{mp} + n_{el} \cdot s_{data}}{8} \right\rceil \cdot 8 \ [B] \qquad \text{(array object)} \end{aligned} \tag{2.1}$$

where s_{mp} represents the size of machine-pointer (also known as word), s_{data} the size of the actual data, and n_{el} denotes the number of elements in the array. The result is given in bytes $[B]$.

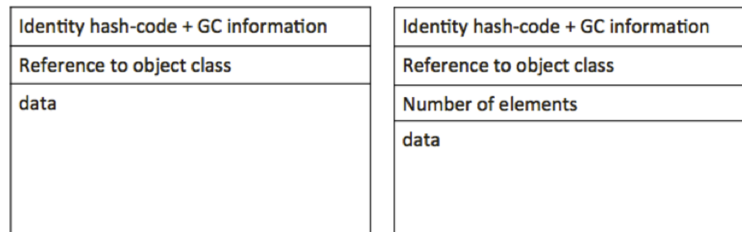

Figure 2.1: Architecture of a standard object and array object in the OJVM.

The storage of objects is different in JJVM and OJVM. In JJVM objects are split into fixed-size blocks with a length of n words (Part A of Figure 2.2 shows an empty example block). The size of the word is the same as for the OJVM – the size of a machine pointer. If an object is larger than the block-size, the data of the object has to be split into several blocks. Also, the JJVM stores standard objects differently than array objects. Both structures do have a so called head-block that contains additional information of the object itself and in case of an array of the sub-structure. The head-block for standard-objects requires three words for information such as color-flag of the object (used during the garbage collection for marking visited objects), type of the object, and monitor of the object (used for the monitoring-process that is available in the JJVM). Then, the data follows and it can be split over successive blocks that are connected by a pointer to the next block (see Part B of Figure 2.2). The head-block of the array-object contains one more reserved word that has information about the length and depth of the array-structure-tree. The data is separated in fields that are located in blocks at the last level of the tree. Each block is connected with link-fields up to the head block (see Part C of Figure 2.2). Only the size of the actual data s_{data} of an object in the OJVM is the same as in an object in JJVM. For standard object the total number of fields (including the head-block information and link-fields) that are used, is as follows:

$$numOfFieldsTotal = \left\lceil \frac{3 \cdot s_{mp} + s_{data}}{s_{mp}} \right\rceil \tag{2.2}$$

The following equation calculates the number of *total blocks* (TB) that are necessary to provide enough space for the number of fields of Equation (2.2):

$$TB_{so} = \left\lceil \frac{numOfFieldsTotal - 1}{(n-1)} \right\rceil \tag{2.3}$$

where n is the number of fields that has been chosen for the block-size.

Then, the total size of a standard object on the heap of the JJVM (sso_{JJVM}) is as follows:

$$sso_{JJVM} = (TB_{so} \cdot n \cdot s_{mp}) \ [B] \tag{2.4}$$

The memory allocation of array object in the JJVM is a little bit more complex. To explain how the equations for the memory-allocation of an array-object in the JJVM were derived, would go beyond the scope of this paper (for details see [2]). First the depth of the tree is calculated regarding number of fields that are used for the data s_{data}. The number of fields that is necessary for s_{data} is calculated first:

$$numOfFieldsData = \left\lceil \frac{s_{data}}{s_{mp}} \right\rceil \tag{2.5}$$

Since the number of necessary fields is known, the *depth* of the array-structure can be calculated with the equation as follows:

$$depth = \left\lceil \frac{\log_2(numOfFieldsData) - \log_2(n-4)}{\log_2(n)} \right\rceil \tag{2.6}$$

with n = number of fields in one block and *numOfFieldsData* from Equation (2.5).

Now, the number of fields for an array (2.5) and the depth of the array-structure (2.6) are known. With this information it is possible to calculate the total number of *total blocks* (TB) that are used for head-block, linking blocks, and the data blocks for an array-structure as follows:

$$TB_{ao} = getNumBlocks(depth, numOfFieldsData) \tag{2.7}$$

where $getNumBlocks(d, b) = gNB(d, b)$ is a recursive function as follows:

$$gNB(d, b) = \begin{cases} 1 & (\text{if } d = 0) \\ gNB\left(d-1, \lceil \frac{b}{n} \rceil\right) + \lceil \frac{b}{n} \rceil & (\text{if } d > 0) \end{cases} \tag{2.8}$$

Thus, the total size sao_{JJVM} for one array-object on the heap in the JJVM is then as follows:

$$sao_{JJVM} = (TB_{ao} \cdot n \cdot s_{mp}) \ [B] \tag{2.9}$$

This looks the same as for the standard object, but the derivation for the number of $TotalBlocks$ is different ($TB_{so} \neq TB_{ao}$). For each block on the heap, the JJVM stores additional information in so called groups. This information requires some space that belongs the the size of the object. Thus, if the number of *TotalBlocks* is known, the additional space used by the groups can be calculated. The additional space is reserved for the reference-flag for each word in the block ($n \cdot 1$ *bit*), the color-flag (4 *bit*) and the grey-link that shows for 8 blocks the reference to the next group that has at least one grey-marked object. The grey link is as big as one machine pointer and is divided by the number of the blocks in one group (which is 8). So, the total amount of memory for an object is:

$$TotalMemory = TB \cdot \left((n \cdot s_{mp}) + \frac{s_{mp} + n + 4}{8} \right) \ [B] \tag{2.10}$$

with

$$TB = \begin{cases} TB_{so} & (\text{for standard object}) \\ TB_{ao} & (\text{for array object}) \end{cases} \tag{2.11}$$

Compared to the object-size in the OJVM, it should be mentioned that there is some memory-overhead in the JJVM that is produced by linking fields, remaining empty fields (because of the fixed-size blocks), and additional group information. The memory-overhead is the $TotalMemory$ minus the three (for standard objects) or four fields (array objects) of header information, and minus the data-size of the object itself. This leads to the following memory-overhead-equations for standard objects (oh_{so}) and array objects (oh_{ao}):

$$\begin{aligned} oh_{so} &= (TB_{so} \cdot n - 3 - nOFD) \cdot s_{mp} \ [B] + oh_{gen} \cdot TB_{so} \ [B] \\ oh_{ao} &= (TB_{ao} \cdot n - 4 - nOFD) \cdot s_{mp} \ [B] + oh_{gen} \cdot TB_{ao} \ [B] \end{aligned} \tag{2.12}$$

where oh_{gen} is the general size of memory that is used for each block for the group information as follows:

$$oh_{gen} = \left(\frac{s_{mp} + n + 4}{8} \right) \ [B]$$

2.2 JVMTI-Tool

The *Java Virtual Machine Tool Interface* (JVMTI) provides a library to create so called agents to analyze the states of any program during the runtime of the virtual machine. In this paper, the agent "hprof" given by the demos of the JVMTI was modified. This agent only needs minor changes to cover the necessaries for the analysis of the memory between the OJVM and the JJVM. In this case is used the possibility to get information such as size and type of each object that has been allocated by the profiled application. This information is saved and later used for the recalculation of standard and array object-sizes to the memory needs of the JJVM because the profiling takes place only during the runtime with the OJVM. Since it was not possible to get a runnable JJVM, the recalculation step was necessary to get a comparable base. This means that the size of standard objects and array objects in the OJVM is recorded and then recalculated to the memory-needs in the JJVM with the help of the theoretical calculations in Section 2.1. With the help of the type of the object and the known header information (2 fields for standard objects and 3 fields for array objects) it is possible to separate the actual data-size s_{data} and use it for the calculations of memory used and overhead in the JJVM.

2.3 Benchmarks

The objective of this paper is to check how the memory allocation for different block-sizes n looks like for real life applications and for specific cases where it is possible to specify the general size of the allocated objects, because Siebert did not test with real life benchmarks (as mentioned in the introduction). Two real life applications were found in the SciMark and Sunflow of the SPECjvm2008 Benchmark [11, 12]. This is also the reason why only two of the whole SPECjvm2008-Benchmark-set were used for this paper. The SciMark is a performance test on the CPU and the Sunflow is a multi-threaded renderer to test graphic visualization.

The memory usage for the two benchmarks (in both virtual machines) and the memory-overhead for four different block-sizes ($n = 8, n = 16, n = 32$, and $n = 64$) in the JJVM are calculated.

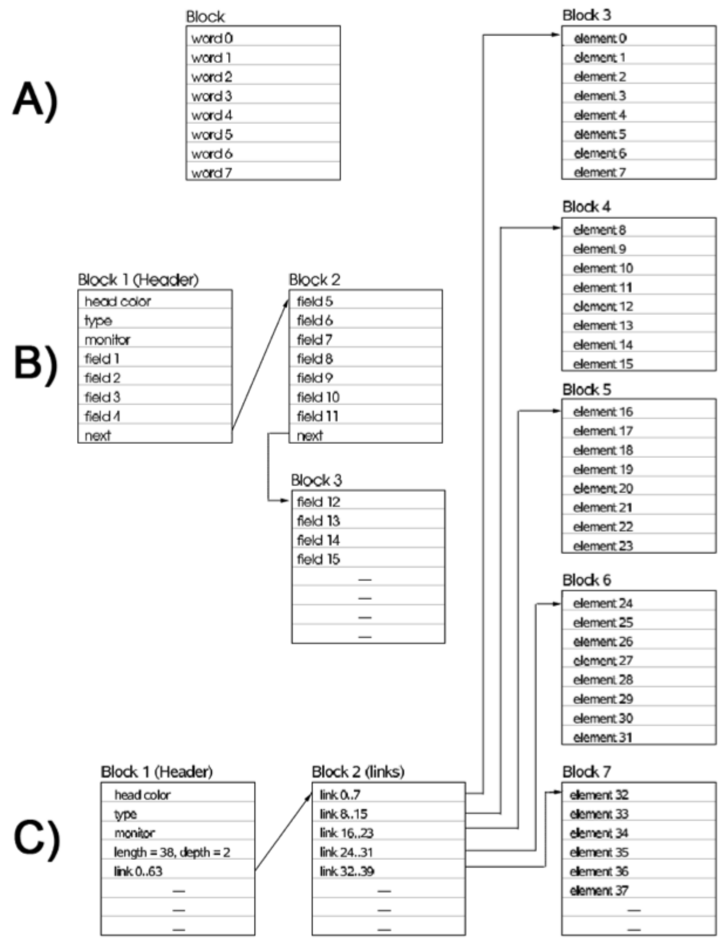

Figure 2.2: Architecture of a fixed-size block A), standard object B) and array C) object in the JJVM.

Chapter 3

Results of SPECjvm2008-Benchmarks

Figure 3.1 shows the results for the Sunflow- and the SciMark-benchmark. Each figure shows a diagram with the size of the used memory for the OJVM and for each defined block-size ($n = 8$, $n = 16$, $n = 32$, and $n = 64$ fields) in the JJVM. The diagram also shows how much of the used memory is overhead that was created by the JJVM (grey bar charts).

There is a small difference of the used data-size together with header-information (black part of the bar charts) between the OJVM and the JJVM because both types of objects in the JJVM (standard object or array object) use one more word than in the OJVM. In summary, the difference between both machines becomes significant. It can also be seen that the memory-overhead increases with the size of the blocks. A closer look into the distribution of the size of the objects shows that there are much more small object allocations than big objects allocations (around 50% of objects are lower than 25 bytes; the rest is located in higher ranges up to 20 kilobyte). The logical result is that there is less memory-overhead for smaller block-sizes than for larger block-sizes. It should be noted that the memory-overhead for the block-sizes $n = 8$ and $n = 16$ is mostly the same in the SciMark-benchmark. The reason for this is explained in the next Section 4.

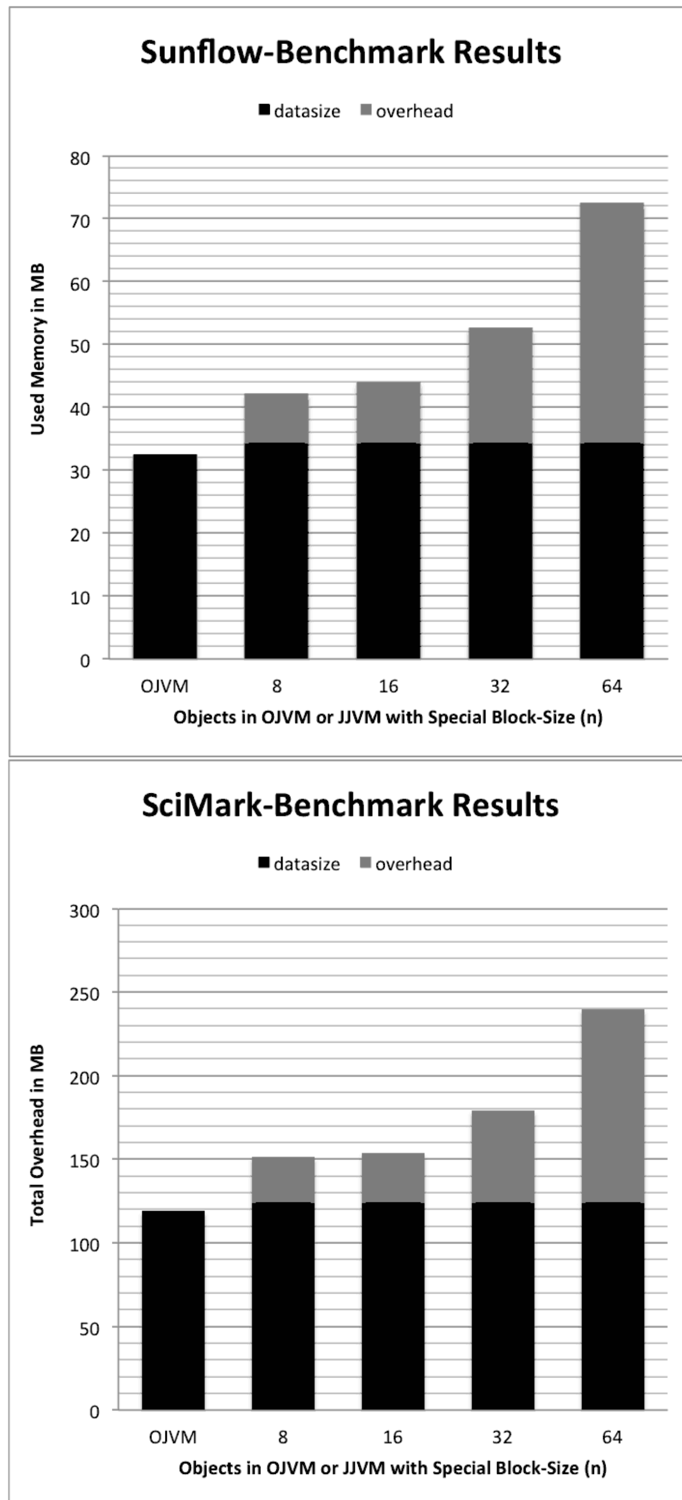

Figure 3.1: Used memory and memory-overhead of Sunflow and SciMark benchmark for different block-sizes ($n = 8$, $n = 16$, $n = 32$, and $n = 64$ fields) and for both virtual machines. For both cases, the best block-size is $n = 8$ but in case of the SciMark benchmark it is also possible to use the block-size of $n = 16$ because there are only minor differences in the memory-overhead.

Chapter 4

Optimal Parameter Setting

In previous chapter, some results of the memory-overhead investigation according to specific programs that the agent analyzed during the runtime were presented. Now, the objective is to find the best block-size for any given program so that the memory-overhead stays as small as possible. Thus, this chapter explains how to get the best block-size for a given program. Section 4.1 presents some further analysis on the memory-overhead for the first four successive block-sizes with $n = 8$, $n = 16$, $n = 32$, and $n = 64$. It shows that there are some ranges that have the same memory-overhead for two or more block-sizes at specific data-sizes. This means that it does not matter which block-size is chosen because the memory-overhead would be the same for some data-sizes. This fact makes the exact calculation for the correct block-size a little more challenging. Sections 4.2 and 4.3 show how to get the block-size for a specific data-size for each type of data structures in the JJVM. Finally, the overview for the first four block-sizes is presented and the experimental results are compared to this diagram.

4.1 Comparison of Overhead Between all Block-Sizes

The Figures 4.1 and 4.2 show the memory-overhead for both types of data structures in the JJVM. It can be seen in both cases that in most cases the memory-overhead varies according to the different block-sizes for one specific data-size but there are also some ranges, where the memory-overhead does not change for all four tested block-sizes. For example in the range between 416 and 440 bytes the memory-overhead for all block-sizes of the standard objects has mostly the same size for the first four block-sizes ($n = 8$, $n = 16$, $n = 32$, and $n = 64$ fields). A synthetic test-program that creates objects with the data-size of 424 bytes (this data-size lays in the previous mentioned range) showed that the overhead indeed became the same size for all four block-sizes.

There are also some data-sizes where the memory-overhead gets smaller for

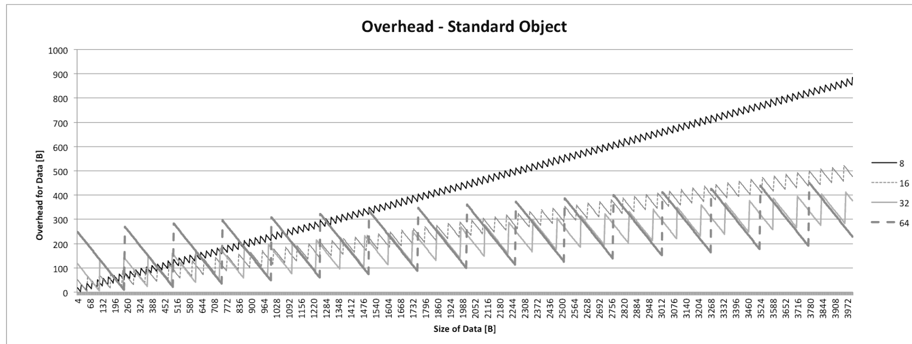

Figure 4.1: Comparison of overhead in standard objects between different block-sizes ($n = 8$, $n = 16$, $n = 32$, and $n = 64$ fields) and the same data-size. It shows that it makes no sense to use smaller block-sizes for bigger data-sizes in general because the memory-overhead becomes bigger than for bigger block-sizes (e.g., an object with the data-size around 3000 bytes creates an memory-overhead of around 650 bytes for the block-size of $n = 8$ and only around 400 bytes for $n = 64$).

one block-size than for another one, but for the next few data-sizes the size of the memory-overhead for the same block-sizes switches their order. For example, for the data-size of 496 bytes, the memory-overhead for the blocksize $n = 32$ is around 158 bytes and for the block-size $n = 64$ around 2 bytes, but for the data-size of 500 bytes, the memory-overhead for the block-size $n = 32$ is almost constant and the memory-overhead of $n = 64$ increases up to around 283 bytes. Thus, in the first case the block-size $n = 64$ would have been chosen, whereas in the second case the block-size of $n = 32$ would have been chosen because of the lower memory-overhead in both cases.

Because of the previously mentioned overlapping area and the switching of the memory-overhead-sizes, it is not possible to predict exactly for which data-size-range the best block-size should be taken. For one specific data size the best block-size can be chosen but not for a whole range that is actually more applicable for general programs because objects of different sizes are usually allocated and not only from one specific size. Some more specific analysis on the memory-overhead and the results for such ranges is explained in the next Sections 4.2 and 4.3.

4.2 Block-Size n for Specific Standard Objects

To get the recommended blocksize n for any data-size s_{data}, there are several steps necessary. First, to evaluate how big the differences between the neighbouring block-sizes in the memory-overhead would be, the difference between the memory-overhead of the current block-size n and the memory-overhead of

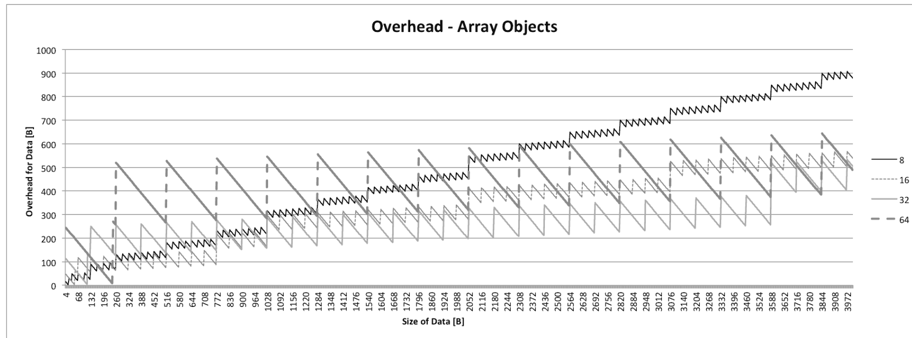

Figure 4.2: Comparison of overhead in array objects between different block-sizes (8, 16, 32, and 64) and the same data-size. It shows that it makes no sense to use smaller block-sizes for bigger data-sizes in general because the memory-overhead becomes bigger than for bigger block-sizes (e.g., an object with the data-size around 3000 bytes creates an memory-overhead of around 700 bytes for the block-size of $n = 8$ and only around 300 bytes for $n = 32$

the next successive block-size ($n \cdot 2$) is calculated. If the difference gets below zero, the memory-overhead of the successive block-size is bigger than the current one. With this information it can be seen at which data-size it is better to switch from the smaller block-size n to the next block-size $n \cdot 2$ to avoid high memory-overhead. There are always two thresholds that define three operating regions for block-size allocations:

- All data-sizes up to the first threshold:
  For all data-sizes up to the first threshold, should definitely be taken the smaller block-size because then the memory-overhead is also smaller for the smaller block-size.

- All data-sizes bigger than the second threshold:
  For this, the memory-overhead never becomes smaller than zero, which means that the bigger block-size ($n \cdot 2$) has a lower memory-overhead than the smaller block-size n and should be taken.

- All data-sizes between the two thresholds:
  The difference of the memory-overhead of the data-sizes between those two thresholds varies from negative to positive values and vice versa. This means, for this range, a general decision cannot be made if the smaller or bigger block-size should be taken. Usually, implementations do not allocate objects with exactly the same size, but might be located in a specific range. For this, a more simplified representation of the data is used. One simplified representation is found by the linear regression line that shows the memory-overhead-difference between two block-sizes as linear slope. The regression line intersects with the x-axis at one point.

This point is chosen as threshold for switching the block-size if the data-size becomes bigger in this paper.

For example, in the analysis to this paper the linear regression of the memory-overhead-difference between the block-sizes $n = 8$ and $n = 16$ for standard objects shows an intersection with the x-axis at 16 bytes ($0 = 0.098 \cdot x - 15.914 \Leftrightarrow x = \frac{15.914}{0.098} = 163,4$). This is then the recommended maximum of the data-size for the block-size $n = 8$.

4.3 Block-Size n for Specific Array Objects

The same steps as given for the calculation to get the threshold for standard object apply for the threshold of array objects.

4.4 Which Block-Size for a Specific Program?

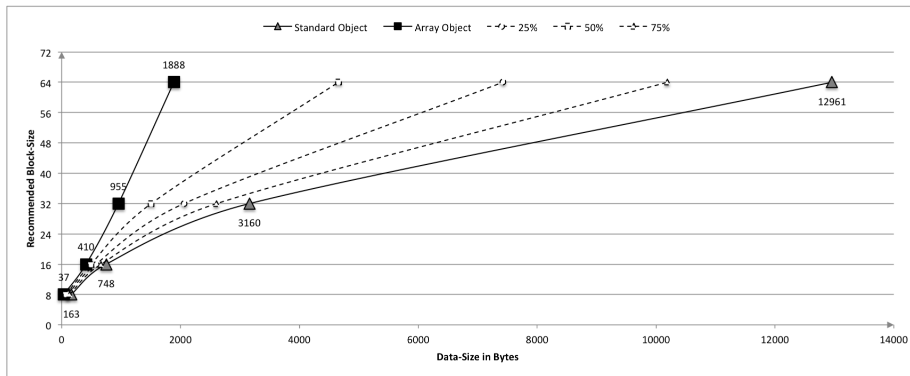

Figure 4.3: Recommended maximum block-size n for a specific data-size s_{data}. For the known distribution of the data-size it can be looked up which block-size is recommended regarding the ratio of standard- and array-objects.

Figure 4.3 shows the recommended block-size for a specific data-size for standard and array objects for the first four block-sizes ($n = 8$ up to $n = 64$). The numbers are the intersection points with the x-axis of the regression-lines that are explained in previous two sections. For any object-size, the suggested block-size can be read by following the slope to the next block-size of 2^n (with $n \in \mathbb{N}_{>2}$). For example, if there is an array object allocation of the size around 500 bytes, it is recommended to use the block-size of $n = 32$. In general, it can be said that for array objects the recommended size of the block-size increased faster than for standard objects. This is because of the different architecture of both types of objects. Standard objects produce less memory-overhead than array objects with the same data-size. To get the correct block-size for a program, it is necessary to analyze first the object allocations of

the program. One option is to analyze the distribution of the object sizes for standard and array objects (i.e., histogram). If the size of the typical allocation of a standard and an array object is known, it can be looked up in the diagram which block-size matches the best.

If the problem arises that the analysis for each object type would lead to two different block-size recommendations, it should be weighted which block-size is better regarding the memory-overhead for the other object type. For example, if the data-size of standard objects would be 200 bytes but for array objects around 8000 bytes then the recommended block-size would be $n = 8$ for standard objects and $n = 64$ for array objects. If the block-size of 64 is chosen, the memory-overhead of standard objects is in general too high because there is always a high number of unused fields in smaller object-allocations. Otherwise, if the block-size of $n = 8$ is chosen, the memory-overhead for the array-list increases a lot because of additional depths and linking fields that cause a memory-overhead. If the tendency of the object allocations is known (e.g., 25% array objects and 75% standard objects or vice versa), it is possible to find the block-size by just using the assisting lines (e.g., 75% for the previous mentioned case). The higher the percentage of the distribution between array-object and standard-object is, the higher is the number of standard object allocations. 50% means that the number of allocations for standard and array objects is the same. If the tendency is not known, it is recommended to use the 50% line which is a good assumption, because this works well for all mentioned examples in this paper.

If larger block-sizes are chosen, the total runtime of the garbage collection cycle decreases because there are less blocks that has to be marked. This is actually an advantage for the purpose of the JJVM to have small and predictable run-times in the GC. On the other hand, objects that are smaller than the size of the block, create more memory-overhead. So, again, this remains at hands of the programmer to choose a block-size and he has to decide if he wants less memory needs or less total GC times.

In general, it can be said that the block-size of $n = 8$ is a good size because in most applications, the size of the objects do not increase beyond 37 bytes for standard objects or 163 bytes for array objects. These are maximum sizes for the data that fits for the block-size $n = 8$ for standard and array objects, respectively (compare Figure 4.3). This can also be seen in the two tested real life applications SciMark and Sunflow. All test results show that the block-size of $n = 8$ is the best because of the smallest memory-overhead. Only in special cases, it is necessary to choose another block-size or it does not matter (e.g., the synthetic test program that allocates objects with the size of 424 bytes).

Chapter 5

Conclusion

The results presented in this paper lead to the conclusion that the block-size of $n = 8$ works the best for the most programs because they usually have many objects that are below the size of 163 bytes. This is the upper bound for standard objects and the block-size $n = 8$ in the JJVM (see Figure 4.3). The upper bound for the recommended block-size of standard objects is always higher than for the array objects. This is because of the higher memory-overhead that array-objects produce than for standard-objects for the same data-size.

If the tendency of the distribution between standard and array objects is known, the programmer can use the assisting lines in Figure 4.3. Unfortunately, objects are seldom uniform, but it turns out that all programs in this paper and the results of them show that they are good matches for the 50%-line. More experiments are required to confirm that the distribution of standard and array objects is equal for the general case.

Although the JJVM provides a garbage collection for real-time systems with critical time constrains, the costs for the necessary memory of the garbage collection are higher than in the OJVM. There was a minimum average of 20% increase in memory-use in the JJVM when it is compared with OJVM for all programs and benchmarks used in this paper and when the block-size was chosen that produces the smallest memory-overhead. This is definitely a disadvantage of the JJVM.

For programs that have more larger size objects, larger block sizes are recommended, since the memory-overhead for accessing large blocks is small. This might also be a problem because if there are two main distributions of objects (e.g., one small and one big object-distribution) that actually causes two different block-size recommendations, the programmer has to decide which block-size he should use in his implementation.

Regarding the two tested JVMs the conclusion is as follows: If an implementation does not need a virtual machine that fits the hard real-time goals, it is recommended to use the standard OJVM. The JJVM should be taken in the other case. The JJVM should also be taken if short and predictable GC pauses are necessary, although more memory is used. Then, the programmer

just needs to provide a bigger memory for the heap.

Chapter 6

Future Work

Siebert [8] already gave an overview of the runtime-overhead of the GC in the JJVM but also mentioned that the benchmarks that were used do not reflect the characteristic of real applications. The benchmarks that were used by Siebert show that the GC of the OJVM runs in average 20% faster than the JJVM [8]. Some future work should be the testing with real applications (or benchmarks that reflects those ones) on the JJVM and compare the run-time with the OJVM.

There should also be a more specific performance analysis regarding the change of the the block-size. For example, it can happen that the programmer chooses a small block-size that leads in case of big objects to many small blocks-divisions that has to be checked during the GC. The bigger the objects are the more connections hast to be checked. This leads to a more time-consuming fact than it would be if the block-size would not have been chosen too small.

Bibliography

[1] aicas, **JamaicaVM - Java Technology for Realtime**, URL: http://www.aicas.com/jamaica.html, Retrieved: 8 July 2011

[2] Markus Goffart **Comparison of Memory Allocation in the Jamaica and Oracle Java Virtual Machine**, Master's Thesis, University of New Brunswick, Fredericton, January 2012

[3] Oracle, **Java Virtual Machine Tool Interface - Version 1.2**, URL: http://download.oracle.com/javase/6/docs/platform/jvmti/jvmti.html, Retrieved: 8 July 2011

[4] Oracle, **Whitepaper - The Java HotSpot Performance Engine Architecture**, URL: http://java.sun.com/products/hotspot/whitepaper.html, Retrieved: 25 September 2011

[5] Siebert F., **The Impact of Realtime Garbage Collection on Realtime Java Programming**, Journal: Object-Oriented Real-Time Distributed Computing, 2004. Proceedings, Date: 14 May 2004, Page(s): 33 - 40, ISBN: 0-7695-2124-X, INSPEC Accession Number: 8302198

[6] Siebert F., **Slides to: The Impact of Realtime Garbage Collection on Realtime Java Programming**, URL: http://www.cs.uni-salzburg.at/announcements/Slides_Siebert.pdf, Retrieved: 8 July 2011

[7] Siebert F., **Concurrent, Parallel, Real-Time Garbage-Collection**, Journal: ISMM '10 Proceedings of the 2010 International Symposium on Memory Management, Year: 2010, ACM New York, ISBN: 978-1-4503-0054-4

[8] Siebert F., **Realtime Garbage Collection in the JamaicaVM 3.0**, JTRES'07 - The 5th International Workshop on Java Technologies for Real-time and Embedded Systems - JTRES 2007, 26-28 September 2007, Vienna, Austria

[9] Standard Performance Evaluation Corporation, **Java Virtual Machine Benchmark**, URL: http://www.spec.org/jvm2008/, Retrieved: 8 July 2011

[10] Standard Performance Evaluation Corporation, SPECjvm2008 User Guide; Version 1.0; Last modified: 16 April 2008, URL: http://www.spec.org/jvm2008/docs/UserGuide.html, Retrieved: 2 November 2011

[11] Standard Performance Evaluation Corporation, **Sunflow - Description**, URL: http://www.spec.org/jvm2008/docs/benchmarks/sunflow.html, Retrieved: 2 November 2011

[12] Standard Performance Evaluation Corporation, **SciMark - Description**, URL: http://www.spec.org/jvm2008/docs/benchmarks/scimark.html, Retrieved: 2 November 2011

[13] Sun Microsystems, **Memory Management in the Java HotSpot(TM) Virtual Machine**, Date: April 2006, URL: http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf, Retrieved: 17 October 2011