Evaluating the Importance of Virtual Memory for Java

Yolanda Becerra Toni Cortes Jordi Garcia Nacho Navarro
Computer Architecture Department. Universitat Politècnica de Catalunya
C/ Jordi Girona,1-3, 08034 Barcelona, Spain
{*yolandab,toni,jordig,nacho*}*@ac.upc.es*

Abstract

The Java language has rapidly become widespread and it is being used to implement a broad range of applications, including applications with high resource requirements. For this reason, it is important to evaluate the suitability of the Java environment to execute such applications. This paper presents an evaluation of the effects of memory management in the context of memory intensive Java applications executed on a virtual memory system. The goal of this work is to detect the most critical memory management issues for Java applications performance. We measure the overhead that each memory management task adds to the application execution, and we determine which part of this overhead is due to the memory access pattern of the application and which part is due to the interaction between the different memory management tasks.

1. Introduction

The specific features of the Java language make it appropriate for usage in very different fields. Portability, object-oriented paradigm or security are powerful reasons for writing applications in Java. Thus, the Java language has rapidly become widespread and we can find projects working with it in the areas of business applications [6], data mining [16] or scientific applications [15].

As a result, Java is used to implement applications with high resource requirements, and, in particular, with a high consumption of memory. For example, applications solving engineering problems or physics problems often work on large data set inputs that do not fit into physical memory [4]. These applications are referred to as out-of-core applications and due to virtual memory limitations they often can not rely on this technique to obtain good performance for their data accesses.

We have to remark that the accuracy of the results of this kind of applications usually depend on the size of the data set inputs, and this size is limited by the resource availability. Therefore, although adding more physical memory could alleviate the memory preasure for a fixed data set input, it is important to offer good memory management in order to allow larger data set inputs and, therefore, to improve the accuracy of the results.

Moreover, adding more physical memory is not possible for all scenarios. For example, grid computing is an emerging technique that allows the execution of applications distributed on an heterogeneous computer network and that it is also used to implement Java applications [9]. Computers in the grid have very different resource availabilities, and therefore, an application piece can be assigned to a machine with a small amount of physical memory.

Therefore, in order to obtain good performance, it is fundamental to offer these applications an execution environment with a good memory management. In this paper we evaluate the memory management of Java environments in a virtual memory system, and the effects of this management on memory intensive application performance.

Java applications are executed on the environment provided by a Java Virtual Machine (JVM) [14], which isolates applications from the underlying system. The method used to offer such an isolation consists in first compiling the application code into an intermediate bytecode which is machine independent. Then, the JVM executes the resulting bytecode either using an interpreter or compiling the bytecode with a Just-In-Time compiler (JIT).

Memory management of Java environments on virtual memory systems

The memory management tasks on virtual memory systems can be grouped into two different levels: implementation of the virtual memory technique and maintenance of the logical address space layout.

On the one hand, the operating system implements the virtual memory management. It decides which logical memory is loaded on physical memory, solving page faults and replacing the contents of physical memory when needed. A page fault involves selecting a physical address to support the data required and loading it from backing storage (typically the disk). Due to the different access time between disk and memory, the goal of the virtual memory management is to reduce the number of page faults generated by processes.

On the other hand, the JVM implements the address space of the Java applications, and manages the allocation and the deallocation of logical memory. Allocation of memory is application driven, and it happens whenever the application creates a new object. This process consists basically in deciding the placement for each new object in the logical address space. Deallocation of memory can be requested explicitly by the applications, or can be launched by the JVM if it detects that free logical memory is needed. In any of these two cases, the selection of which memory areas are deallocated is transparent to the application because the JVM uses the garbage collection mechanism to detect non-referenced objects and to free the logical memory they use.

Both levels of memory management are usually implemented separately. However, their execution is closely related and can affect the performance of each other. Thus, in order to analyze the effects of memory management on application performance, we evaluate the different memory management tasks, as well as the interaction between tasks from different memory management levels. In addition, as the JVM interferes with the application execution, such interferences have to be studied and analyzed.

The rest of this paper is organized as follows. The next section shows some related work. In section 3 we present our working environment. Section 4 describes the experiments that we have executed as well as the results obtained. Finally, in section 5 we present the conclusions from our evaluation.

2. Related work

There is a lot of previous work focused on each of the different memory issues: virtual memory technique, allocation and deallocation of logical memory. They have studied a single issue in isolation or the relationship between two of the memory issues, although none of them has tried to layout the whole picture. In addition, memory management performance depends heavily on memory access patterns, and these patterns are closely related to the language characteristics. For this reason, memory management evaluation has to be done for a given language, and therefore it has to be done for Java applications.

2.1. Virtual memory technique

The research on virtual memory management follows the current research trends on operating systems. Traditionally, operating systems implemented general purpose policies that fit well with average application requirements. However, current operating system research is oriented to offer application specific resource management. The goal is to adapt policy decisions to the application behavior to get better resources exploitation [19, 8, 2]. However, in the operating systems scope, there is no work on considering particular features of the Java environment, as, for example, interleaving the execution of the garbage collector and the execution of the application code.

2.2. Allocation of logical memory

One aspect of the logical address space management is allocation. An important issue of allocation is to decide which free logical address is assigned to a new object. There are a lot of studies analyzing the influence on performance of this placement decision. For example, there are studies that showed how to group objects in order to improve garbage collection performance [17, 22, 18]. Other placement studies worked on improving the performance of the virtual memory technique. These studies proposed different criteria for grouping the new objects in pages in order to help the operating system to maintain the data in use in physical memory [21, 17, 25]. Wolczko and Williams went even further proposing dynamic relocation of objects to adapt the object placement to changes in the execution characteristics [27].

In addition to placement decisions, the effects of the allocation execution have also been studied. Wilson et al. stated that, as this process executes with a cyclical pattern, it may disrupt operating system replacement algorithm [26].

2.3. Deallocation of logical memory

In the Java environment, deallocation of memory is implemented through garbage collection. There are many proposed garbage collection techniques, each of them with a different goal [11]. For example, incremental garbage collectors attempt to reduce the application execution pauses due to garbage collecting; collectors with compaction try to speed up the allocation process. There are also some garbage collection techniques that take into account the impact on memory hierarchy performance and try to maintain the locality of the applications. Generational garbage collectors follow this goal by reducing the amount of objects checked for deallocation.

2.4. Analysis of Java applications behavior

As the Java environment is gaining in popularity, plenty of work has analyzed the execution of Java applications in order to understand the interactions between the Java environment and the underlying system. For example, Kim and Hsu have studied the interaction between caches and Java applications running with a JIT compiler [12]. Moreover, they have studied the influence of the initial heap size on cache performance and they have pointed that this value may also affect the virtual memory performance.

Li et al. analyzed executions with a JIT compiler as well [13]. Their work was centered on understanding the operating system activity during Java applications executions. As virtual memory is part of the operating system activity, it also quantified the time spent on this management, but they did not distinguish which part of the work is done on behalf of the applications and which part on behalf of the JVM.

The study of Dieckmann and Hölzle [7] was oriented to help garbage collector implementors, and showed the kind of objects the Java applications allocate and the distribution in time of these allocations.

2.5. Our memory management evaluation

In this paper, we evaluate the effects of the memory management on memory intensive Java applications. We quantify the overhead of this management measuring the time spent solving page faults. In addition, we determine which part of this overhead is due to the application code and which part is due to interaction between the virtual memory technique and the JVM memory management.

Previous work that analyzes the Java applications behavior works on applications from the SPECjvm98 benchmarks suite [20]. Although these applications execute a lot of memory requests, they have small working sets that fit on a reduced physical memory size. Moreover, Dieckmann and Hölzle showed that these applications can execute on less than 8Mb of heap [7]. Therefore, for these applications the percentage of time taken by the page fault management is negligible, if compared with the total application execution time [13].

We work on the applications from the JavaGrande benchmarks (see section 3.3). Unlike SPECjvm98 benchmarks, these applications execute with large working sets requiring a lot of physical memory, and therefore the effects of execution on virtual memory are more relevant.

3. Working environment

The platform we used is composed by the Java Virtual Machine from Sun running on top of the Linux operating system. The main reason for our selection is that we need a free distribution source in order to modify both the Java Virtual Machine and the operating system. The benchmarks we have chosen for the experiments are part of those provided by the JavaGrande Forum [10]. We have run all this software on a 500 MHz Pentium III processor, with 128 Mb of physical memory. The following subsections describe each software component of our working environment.

3.1. Linux operating system

We have used the 2.2.12 version of the Linux operating system kernel [3]. One of the main aspects of virtual memory management is the algorithm used to map logical pages into physical ones. The version of Linux we have studied uses a page replacement algorithm that approximates the Least Recently Used policy. This approximation is based on a process that periodically sweeps all physical memory, aging those pages not referenced since the last time this process was executed. When free pages are needed, the victim pages are selected from the older ones. It first tries to deallocate pages not modified, because such pages do not need to be written to the swap space. If more free pages are required, then modified pages are replaced.

3.2. Java 2 virtual machine

We have used the classic virtual machine of the Java 2 SDK Standard Edition from Sun version 1.2.2 for Linux. In order to describe the memory management of this JVM we have to focus on three main issues: heap organization, allocation, and deallocation of logical memory.

In this JVM version, the application heap is divided into two separate zones: one for handles and one for objects. The JVM requests the operating system enough memory to hold a maximal heap length, although it initially offers the application a minimal heap length. Both minimal and maximal heap length can be decided by the user. During execution, both the handle zone and the object zone can grow or diminish following the application requirements, while maintaining the heap between the limits established (see Figure 1).

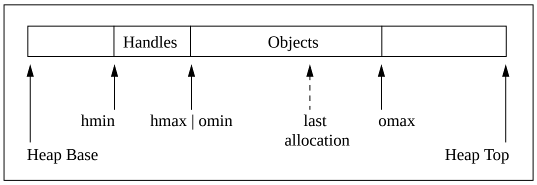

Figure 1. Heap layout in the classic JVM

Memory allocation is done in a sequential and cyclic fashion. When an application creates a new object, the JVM looks for enough contiguous memory to hold that object, starting from the address where the last allocation was satisfied, and finishing when such a block is found or when it reaches the initial searching point. This cyclic search through all the heap tries to reuse holes left by previous deallocations before starting the garbage collector. Therefore, it may involve accesses to many logical pages just to check if they have enough contiguous free memory for the new object.

Memory deallocation is done through garbage collection. The algorithm this JVM uses is mark and sweep, compacting the free memory in the heap and modifying the heap limits when it can benefit the application execution [11].

First of all, the garbage collector traverses the handle area in the heap, detecting and marking referenced objects (mark phase). Then it accesses the non-marked objects, freeing the memory they occupied (sweep phase). During this last phase, the garbage collector accesses pages with data no more in use.

This process may generate free memory holes over the heap, that may involve external fragmentation and may also slow down the allocation process. External fragmentation appears when there is enough free memory for satisfying a request, but it can not be used because it is not contiguous. Thus, in order to prevent this situation, after the mark and sweep process, the JVM checks if it is appropriated to compact the free memory in the heap.

Compacting the heap implies several heap traverses: for every referenced object it tries to move it to the first free zone in the heap large enough to hold it.

Once the mark and sweep process and the heap compaction are finished, the JVM checks whether a heap space expansion is needed. The expansion process consists in just modifying the variable that points to the end of the heap, because as explained before, the JVM asks the operating system for memory to hold the maximum heap size when it starts execution.

3.3. JavaGrande benchmarks

The JavaGrande group is working on the use of Java for high performance computing. They provide a benchmark suite for comparing Java environments when executing applications with high resources requirements (*grande applications*) [5]. In this suite there are some kernels frequently used in that kind of applications and a set of applications. For each kernel benchmark they supply three versions of different size (they are referred to, from the smallest to the biggest one, as SIZEA, SIZEB and SIZEC), and for each application benchmark they supply two versions (referred to as SIZEA and SIZEB). We have selected some of the tests with higher memory requirements. In this section we show a brief description of the selected tests (a more complete description can be found in [10]).

- CRYPT: This is a kernel benchmark that implements encryption and decryption on an array. It uses the International Data Encryption Algorithm (IDEA). We have selected the largest version (SIZEC).
- HEAPSORT: This kernel sorts an array of integers using the heap sort algorithm. We have chosen the largest version (SIZEC).
- MONTECARLO: This benchmark is a complete application that implements a financial simulation based on Monte Carlo techniques. We have selected the smallest version (SIZEA), because it consumes enough memory to stress our memory system.
- FFT: This kernel implements a one-dimensional forward transform on a set of complex numbers. We have chosen the smallest version provided for this benchmark (SIZEA), because it consumes enough memory.
- SPARSE: This kernel multiplies sparse matrices stored in compressed-row format. We have run the largest version of this benchmark (SIZEC).

4. Experiments and results

We have performed several experiments in order to evaluate the effect the memory management has on memory intensive applications. First of all, we show the significance of the memory management overhead on the overall application execution time, as well as the kind of memory accesses that are more affected by this overhead. Then we evaluate the different sources of overhead, in order to determine which part of this penalty is due to the application code and which part is due to the memory management code.

We also present the results we obtain simulating the execution of the applications on an optimal memory system. These results offer us the upper bounds for the performance of the memory management.

Finally, we present the results from an initial experiment on the HotSpot JVM that allows us to compare the performance of the memory management of this virtual machine with the memory management of the classic JVM.

4.1. Evaluation of memory management performance

4.1.1. Methodology

We have evaluated the memory management performance counting and classifying the page faults generated while executing the applications. In addition, we have measured the time spent by the operating system to solve each group of page faults. For this reason, we have added to the Linux kernel several counters and configuration variables. Moreover, we have added to the JVM the code that configures the

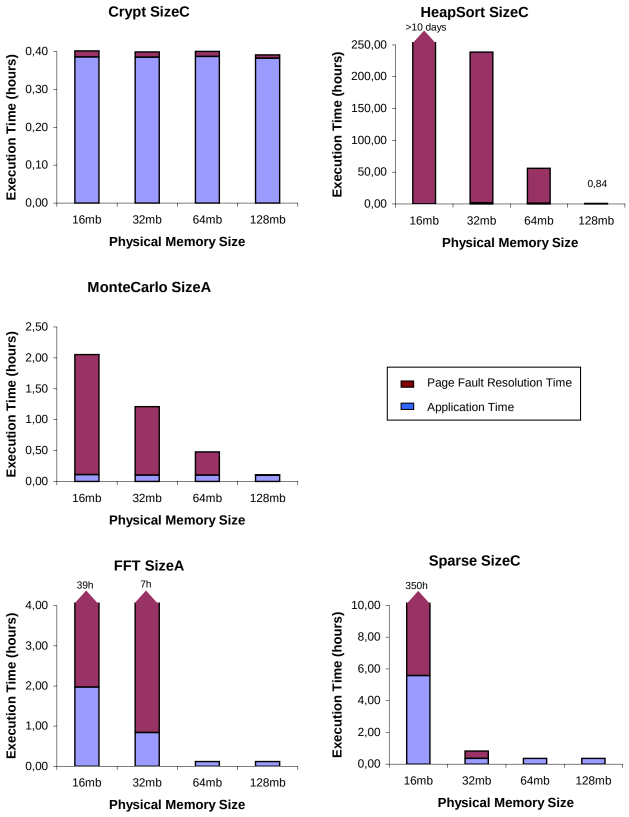

Figure 2. Time dedicated to solve page faults

page fault counting and that consults the results. The details of all these modifications can be found in [1].

As we are measuring real executions on a real environment, we have to take into account the system processes running in the machine, which can affect the measures. In order to mitigate these effects, the numbers presented in this paper are the average of several executions results. We boot the machine before executing a different application to maintain initial conditions for all applications.

We are interested on executing the benchmarks on various physical memory sizes, to evaluate the execution under different conditions. This can be measured in the same machine booting the Linux kernel with a parameter that configures it with the physical memory available in the machine. We have executed our experiments on 16Mb, 32Mb, 64Mb and 128Mb of physical memory.

Another execution parameter that may influence the memory management performance is the heap size, because this value affects the number of the garbage collector executions. We have selected as the minimum heap size the JVM default value, and as the upper limit the minimal value enough for executing the application.

4.1.2. Page fault management time

This experiment shows the significance of the memory management overhead on application performance.

Figure 2 presents the time spent by the operating system to solve the page faults that each benchmark generates. We have only taken into account those page faults requiring access to disk (*hard page faults*), because we have seen that the effect of the page faults solved without accessing the disk is negligible for application performance.

We can see in the graphs that the computing time of CRYPT makes insignificant the time spent solving its page faults. Moreover, the total page fault time is very stable through changes on physical memory size, and therefore, CRYPT execution time is almost the same for all the sizes of physical memory that we have tested.

The execution time of HEAPSORT and MONTECARLO increases for each reduction on the physical memory size, and this increment is due to the page fault management time. Furthermore, except for executions on 128Mb of physical memory, the time dedicated to solve page faults clearly dominates the total execution time for both applications. HEAPSORT shows this behavior clearer. It executes on 128Mb of physical memory without generating any page fault and its execution time is around 50 minutes. If the physical memory size is reduced to 64Mb, the execution time of this application increases to more than 2 days. And reducing the physical memory size to 16Mb increases the execution time of HEAPSORT to more than 10 days.

Finally, FFT and SPARSE require less amount of physical memory to fit. They execute on 128Mb and 64Mb without generating hard page faults. However, if they are executed on 32Mb of physical memory, their page fault solving time increases considerably, dominating completely the total execution time.

The results from this experiment show that, unlike the SPECjvm98 benchmarks ([13]), the applications from the JavaGrande benchmark suite spend a non-negligible time solving page faults. This is because they have larger working sets, consuming a lot of physical memory. Thus, if physical memory is reduced, it is essential to offer these applications a good memory management implementation in order to reduce the number of hard page faults generated, and therefore, the percentage of time spent solving page faults.

4.1.3. Page faults distribution on the address space

The goal of this experiment is to determine which regions of the application address space have more accesses that involve page faults. We have considered three different areas: objects, handles, and out of the heap (code, stack, ...).

The results from this experiment show that, for all the benchmarks, most page faults are caused by accesses to the object region (see table 1). Therefore, accesses to handles

Programs / Mem. Size	16Mb	32Mb	64Mb	128Mb
Crypt SizeC	96.99%	97.50%	98.05%	97.73%
HeapSort SizeC	100%	100%	99.99%	n/a
MonteCarlo SizeA	90.29%	95.81%	96.96%	95.60%
FFT SizeA	99.96%	99.93%	n/a	n/a
Sparse SizeC	100%	99.90%	n/a	n/a

Table 1. Page faults accessing objects (%)

or to the region of the application code are not relevant to the memory performance of these benchmarks. Thus, the best effort would be oriented to improve memory management for the object region, because this is the region where improvement could be more significant.

4.1.4. Sources of memory management overhead

The goal of this experiment is to separate the page faults generated by the application code from the page faults caused by the memory management execution. These results allow us to determine the overhead that the memory management is adding to the application performance.

We distinguish between page faults caused while executing the application code from page faults caused while executing any task of the JVM memory management (object allocation, object deallocation, or heap compaction). However, this classification is not enough to determine the whole impact of the JVM memory management, because the execution of this code can originate some of the subsequent page faults generated by the application code. For example, the execution of the JVM code alters the information about the application locality used by the operating system when replacing memory and, therefore, the operating system may evict application active pages from physical memory.

In order to evaluate this effect, we also execute the applications reducing as much as possible the execution of the JVM memory management code. Execution of applications with minimal memory management is achieved launching them with an initial heap large enough, in such a way that no garbage collection nor heap compaction is required. Moreover, in this scenario, allocation of objects becomes a very simple process with minimal overhead for performance, because it consists in just assigning the logical address next to the last allocation. We have also added to the JVM an option to ignore application explicit calls to garbage collection.

Although this method of execution offers the applications a memory management with minimal impact, when we analyze the results of the experiments we have to take into account that the application address space becomes larger. This could increase the amount of physical memory needed and therefore the number of page faults caused by the application.

Figure 3 shows, for each benchmark, the number of hard page faults generated, both with minimal memory management (referred to as *gcnull*) and with regular memory management (*gc* in the graphs), and classified by the code that generates them (*application code*, *allocation code*, *mark and sweep code*, and *compaction code*). We only show the results for accesses to objects, because as presented in section 4.1.3 this region is the most affected by page faults.

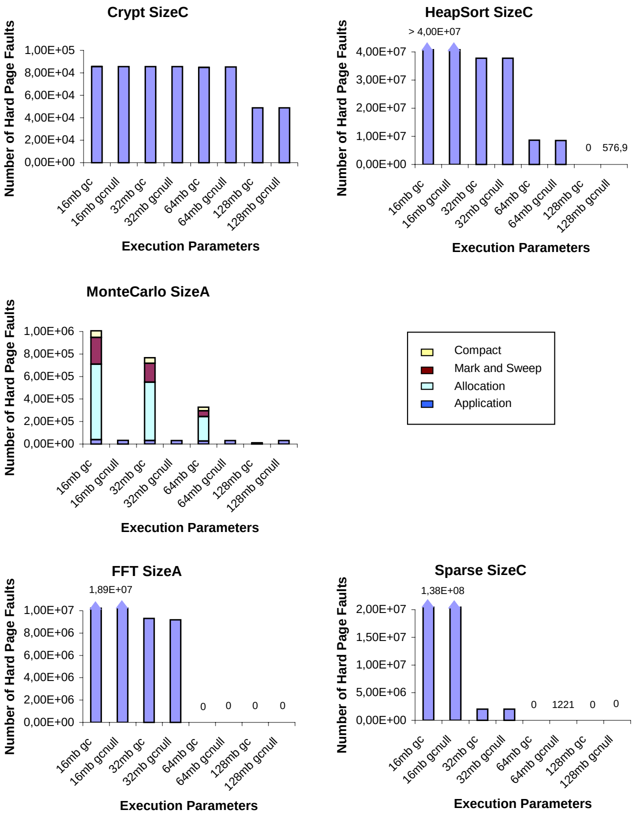

Figure 3. Page faults distribution by code

First of all we observe that, for all the applications except for MonteCarlo, the application code is the main source of page faults. In addition, whether we execute them with regular JVM memory management or with minimal JVM memory management, the number of page faults generated is almost the same. The explanation to this behavior is that these applications need few executions of the garbage collector (this number varies from 3 for FFT and HeapSort to 7 for Sparse), and therefore the execution with regular memory management is very similar to the execution with minimal memory management.

These results suggest that for these applications it makes no sense to work on improving the deallocation management code, because this code is seldom executed.

As we have mentioned before, if no object deallocation occurs, the process of allocating an object is very simple and does not disturb the application performance. However, allocating a new object involves the object placement decision, and the distribution of objects on the address space affects the virtual memory effectiveness. Thus, improving the memory management performance of these applications would be possible if we find a better grouping of objects on pages.

Particular behavior of MONTECARLO

The behavior of MONTECARLO is different than the rest of the benchmarks. This application requires more garbage collector participation, and therefore, results from the application execution with minimal JVM memory management differ from results from regular JVM memory management. Furthermore, the minimal JVM memory management clearly outperforms the regular one, except for executions on 128Mb of physical memory.

Analyzing the execution with minimal memory management, all the page faults generated are due to the execution of the application code, and we can see that the number of page faults is almost identical for each physical memory size. This is due to the memory access pattern of the application: each working set fits in a small amount of physical memory. The page faults occur when the application changes from one working set to another.

In the execution with regular JVM memory management, we observe that a reduction on the physical memory size increases considerably the number of generated page faults. However, we also observe that the number of page faults caused by the application is maintained quite similarly. This means that the execution of the JVM code has a little effect on the virtual memory performance of the application code. Thus, the increment in the number of page faults is mostly due to page faults generated when the JVM memory management code is executing. The effects of each memory management task are the following:

- The object **allocation** is the one that generates more page faults when physical memory is reduced. This is due to the cyclic search for room that requires it to traverse the object region, and potentially incurs many page faults. In the worst case, the heap has no room for the new object and, after traversing the whole object region, the garbage collector is activated.

- The **Mark and sweep** executions generate around 30 percent of the total JVM memory management page faults. As we have explained in section 3.2, this task marks the non-referenced objects traversing the handle area, and the only accesses to the object region are to deallocate the marked objects. Moreover, this task executes only when the allocation task fails and once called explicitly by the MONTECARLO code.

- The page faults generated by the **heap compaction** code only represent less than 10 percent of the total JVM memory management page faults. Although the algorithm of this task is potentially an important source of page faults, its execution is only required once for MONTECARLO.

Finally, if we compare the results from both kind of executions, we observe that the number of page faults the application code generates is slightly higher when it executes on minimal memory management than when it executes on regular memory management. This is due to the reduction of the object area size that achieves the garbage collection, diminishing the amount of physical memory needed to fit it.

4.2. Evaluation of an optimal memory management

This paper evaluates the memory management performance in order to determine the aspects of this management that penalize more the application execution. Therefore, the efforts to improve the Java application memory performance should be focused on these aspects. However, it is important to obtain an upper bound for this performance to quantify the potential benefits from an improved memory management. We obtain this upper bound simulating the execution of the applications on an optimal memory system.

4.2.1. Methodology

We have implemented a simulator of an optimal memory management system, which is composed by both optimal virtual memory management and optimal logical address space management. This optimal memory management requires a perfect cooperation between the operating system and the JVM.

This optimal system maintains the most utilized objects in physical memory. This is achieved through optimal dynamic placement of objects and through perfect deallocation of objects. An optimal object placement groups in evicted pages those objects later referenced in time, and when loading a referenced object groups in its logical page the objects next referenced. On the other hand, a perfect deallocation consists in detecting immediately and deallocating an object with no references left.

Figure 4 shows the algorithm of the reference management routine in our simulator (a more complete description of this simulator can be found in [1]).

The simulation of the execution on an optimal memory system is based on knowing in advance the next reference instant for each allocated object. For this reason, our simulator works on traces that describe the application memory behavior. Thus, we have modified the JVM in order to generate these traces while it is executing the applications.

This optimal management only considers accesses to memory holding objects. We adopt this simplification be-

```
if (!present(object)) {
  required_size = object_size(object);
  if (required_size > free_memory){
     free_memory+=deallocate_inactive_objects()
     while (required_size > free_memory){
        select object further accessed in time
        mark victim object as non-present
        increment free_memory
     }
  }
  if (this is not the first page fault)
     major_page_faults+=required_size/PAGE_SIZE

  mark object as present
  free_memory -= required_size
}
update next reference info
if (this was the last reference to the object)
  mark object as inactive
```

Figure 4. Optimal management algorithm

cause results from our experiments show that this application region is clearly the one centering most of the application page faults (see section 4.1.3).

The page faults generated on this system are unavoidable. Thus, this number represents the best possible behavior of the memory management. Although this bound is unreachable because the system is not implementable, it accomplishes the purpose of showing the chance for performance improvement. Other previous studies [21, 24] used the same method to estimate the benefits of their proposals.

4.2.2. Results

In figure 5 we can see the number of hard page faults involved in accesses to objects both with real execution and with simulated execution in our optimal memory system.

These graphs show that for all the benchmarks, if virtual memory is actively in use, real memory management performance is far from the optimal memory management results. Thus, although in practice it is not possible to reach the performance of the optimal memory system, we can try to improve memory management to get closer to that bound.

4.3. Experiments with the HotSpot JVM

We are currently starting the evaluation of the memory management performance of the HotSpot virtual machine [23], i.e. the current JVM implementation from Sun. We have used the server configuration of HotSpot 1.3.1 for Linux. This configuration is oriented to applications with high consumption of resource and without user interaction. Although this virtual machine has been designed to be more memory conscious than the classic JVM, the effects on virtual memory performance for memory intensive applications are not clear.

The main differences in memory management between the classic JVM and the HotSpot consist in the method for accessing objects and in the garbage collection algorithm.

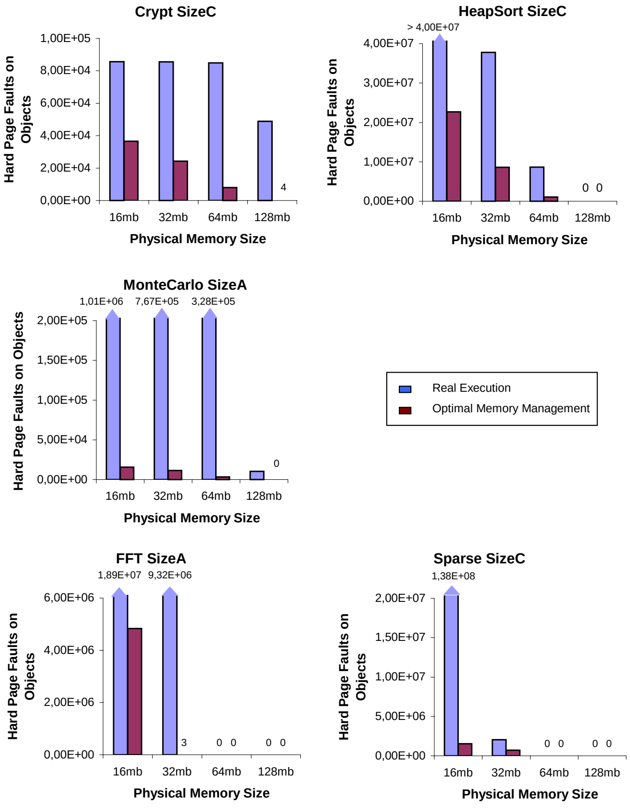

Figure 5. Real vs. optimal management

Unlike the classic JVM, HotSpot does not have separated handles for accessing the objects. Thus, an object reference is solved with a single access to the heap. However, this heap layout involves that any management task that requires checking of the object state has to traverse the whole heap.

The HotSpot uses a generational garbage collector. This kind of garbage collector tries to maintain the locality of applications and is based on the assumption that most allocated objects die young. Thus, it executes frequent *minor collections* that consider just the most recently allocated objects. Only when a deeper cleaning of the heap is needed, it executes a *major collection* that considers also the old objects.

The HotSpot always uses a mark and copy algorithm to collect young objects. However, the user can decide, through an execution argument, which algorithm to use for the major collections. The default option is to use mark and sweep with compact over the whole heap. The alternative is to use an incremental mark and copy. This algorithm divides the heap into small blocks and a single block is traversed on each garbage collector execution.

In order to obtain a first approximation to the HotSpot behavior, we have executed the benchmarks and we have

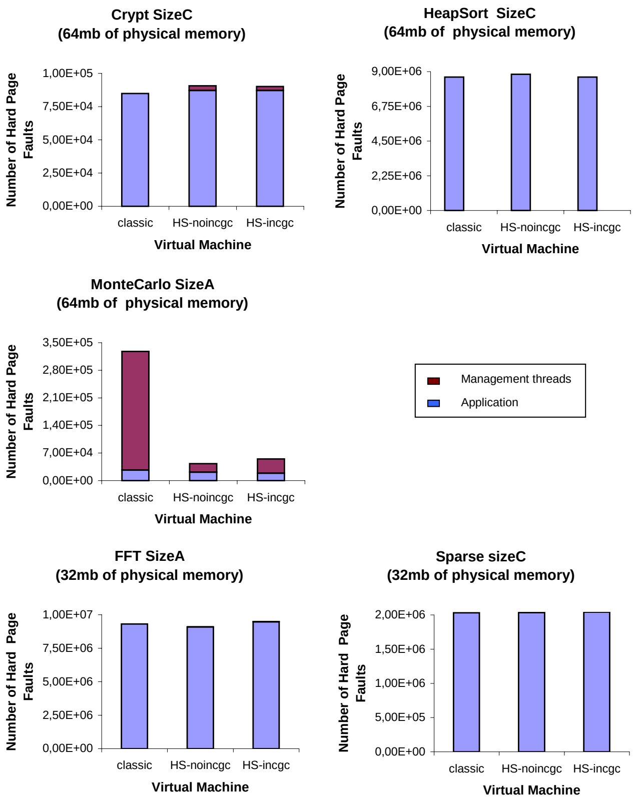

Figure 6. HotSpot vs. classic JVM

counted separately the hard page faults generated by the application code and by any HotSpot management task. Figure 6 shows the results obtained, both with non-incremental (*HS-noincgc*) and incremental (*HS-incgc*) garbage collection, compared to the results obtained with the executions on the classic JVM. We only show the numbers obtained while executing the applications on a physical memory size that stresses the virtual memory system.

We can see in the graphs that for all benchmarks but MONTECARLO, the number of hard page faults is quite similar for both classic JVM and HotSpot. As we have explained in 4.1.4, these applications require few executions of the garbage collector and, therefore, their execution does not take advantage of an improved garbage collector.

In contrast, MONTECARLO requires an active participation of the garbage collection and, therefore, it benefits from a garbage collection algorithm oriented to improve locality. However, the number of hard page faults generated by the management code in HotSpot is still higher than the number of page faults generated by the application code.

In summary, although a deeper evaluation of the HotSpot is needed, these first experiments show that in spite of the improvements implemented on the HotSpot, the behavior of the virtual memory system whether we execute the HotSpot or we execute the classic JVM is similar.

5. Conclusions

This paper presents an evaluation of the effects of memory management in the context of memory intensive Java applications executed on a virtual memory system. The work describes clearly the different tasks of memory management in the Java execution environment, and it analyzes both the overhead introduced by each task and the possible interferences between them.

One contribution of this paper is that we show that there are scenarios and applications where virtual memory is a key issue for performance. Although previous work stated that virtual memory has no effects on Java applications, they used applications from the SPECjvm98 that have small working sets that fit on a reduced physical memory size.

However, the main contribution of this paper is that we identify the issues in memory management that are most critical for Java applications performance.

First of all, we determine which area in the application address space has more accesses that involve page faults. Results from our experiments show that most page faults are caused by accesses to the object region. Therefore accesses to other areas, such as the object handles or the application code, have a negligible impact on the memory management performance.

We also have analyzed which part of the slowdown is due to the memory access pattern of the application and which part is due to any task of the logical memory management implemented by the Java Virtual Machine, such as object allocation, object deallocation or heap compaction.

This analysis shows that not all the applications require an active participation of the garbage collector, as assumed in previous work. Therefore, improving the deallocation algorithm does not revert into an improved performance for these applications. Our initial experiments with the HotSpot JVM confirm that using a more memory conscious garbage collector reduces considerably the number of page faults generated by the execution of the deallocation task; however, the number of page faults generated by the application code is maintained quite similar.

Thus, the memory allocation is the only JVM memory management task that may affect the performance of applications with few executions of the garbage collector. Results from our experiments show that most page faults for this kind of applications are generated by the application code. However, the simulated execution of the benchmarks on our optimal memory system shows that real performance is quite far from an optimal performance. This suggests that it is possible to offer a memory management more suitable to the memory access pattern of the applications, for example, a better object placement policy in the allocation task.

In summary, in this paper we present a detailed study of the memory system performance that shows the relevance of memory management on memory intensive Java applications. The results obtained will be useful to orientate future research in order to improve those aspects of the memory system with the highest impact on the applications performance.

6. Acknowledgments

We thank Todd Austin for his help in the final version of this paper. This work has been partially supported by the Spanish Ministry of Science and Technology and by the European Union (FEDER funds) under the TIC2001-0995-C02-01 contract.

References

[1] Y. Becerra, T. Cortes, J. Garcia, and N. Navarro. Evaluation of memory management performance in memory intensive java applications. Technical Report UPC-DAC-2002-9, Computer Architecture Dept., UPC, Barcelona, Spain, 2002.

[2] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the spin operating system. In *Proceedings of 15th Symposium on Operating Systems Principles*, Saint-Malô, France, December 1995.

[3] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Really, October 2000.

[4] A. D. Brown, T. C. Mowry, and O. Krieger. Compiler-based i/o prefetching for out-of-core applications. *ACM Transactions on Computer Systems*, 19(2):111–170, May 2001.

[5] J. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. A benchmark suite for high performance java. *Concurrency, Practice and Experience*, (12):21–56, 2000.

[6] R. Christ, S. Halter, K. Lynne, S. Meizer, S. Munroe, and M. Pasch. Sanfrancisco performance: A case study in performance of large-scale Java applications. *IBM Systems Journal*, 39(1):4–20, February 2000.

[7] S. Dieckmann and U. Hölzle. A study of the allocation behavior of the SPECjvm98 java benchmarks. In *Proceedings of the 1999 European Conference on Object-Oriented Programming*, June 1999.

[8] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of 15th Symposium on Operating Systems Principles*, Saint-Malô, France, December 1995.

[9] J. G. Gregor von Laszewski, Ian Foster. Cog kits: A bridge between commodity distributed computing and high-performance grids. In *Proceedings of the ACM 2000 Java Grande Conference*, San Francisco, CA, June 2000.

[10] Java Grande Forum. `http://www.javagrande.org`, 2001.

[11] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley and Sons Ldt., 1996.

[12] J. Kim and Y. Hsu. Memory system behavior of Java programs: Methodology and analysis. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems*, 2000.

[13] T. Li, L. John, V. Narayanan, and A. Sivasubramaniam. *Characterizing Operating System Activity in SPECjvm98 Benchmarks*, chapter 3, pages 53–82. Kluwer Academic Publishers, 2001.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, second edition*. Addison Wesley, April 1999.

[15] J. Moreira, S. Midkiff, M. Gupta, P. Artigas, M. Snir, and R. Lawrence. Java programming for high-performance numerical computing. *IBM Systems Journal*, 39(1):21–56, February 2000.

[16] J. Moreira, S. Midkiff, M. Gupta, and R. Lawrence. High performance computing with the array package for Java: A case study using data mining. In *Proceedings of the Supercomputing '99 Conference*, Portland, OR, November 1999.

[17] M. Seidl and B. Zorn. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the ACM ASPLOS VIII*, San Jose, CA, October 1998.

[18] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh. Exploiting profilic types for memory management and optimizations. In *Proceedings of the ACM POPL 2002*, Portland, OR, January 2002.

[19] C. Small and M. Seltzer. A comparison of OS extension technologies. In *Proceedings of the 1996 Usenix Conference*, 1996.

[20] SPEC JVM98. `http://www.spec.org/osg/jvm98`, 1998.

[21] J. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Transactions on Computer Systems*, 2(2):155–180, May 1984.

[22] D. Stefanović, K. S. McKinley, and J. E. B. Moss. Age-based garbage collection. In *Proceedings of ACM 1999 SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

[23] Sun Microsystems, Inc. The Java HotSpot Virtual Machine. Technical white paper, April 2001.

[24] I. Williams, M. Wolczko, and T. Hopkins. Dynamic grouping in an object oriented virtual memory hierarchy. In *Proceedings of the 1987 European Conference on Object-Oriented Programming*, Paris, June 1987.

[25] P. Wilson, M. Lam, and T. Moher. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proceedings of the 1991 ACM Conference on Programming Language Design and Implementation*, Toronto, June 1991.

[26] P. Wilson, M. Lam, and T. Moher. Caching considerations for generational garbage collection. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 32–42, San Francisco, CA, June 1992.

[27] M. Wolczko and I. Williams. Multi-level garbage collection in a high-performance persistent object system. In *Proceedings of the Fifth International Workshop on Persistent Object System*, Pisa, September 1992.