Java Virtual Machine: the key for accurated memory prefetching

Yolanda Becerra **Jordi Garcia** **Toni Cortes** **Nacho Navarro**

Computer Architecture Department
Universitat Politècnica de Catalunya
Barcelona, Spain

Abstract

The popularity of the Java language is easily understandable analyzing the specific features of the language and the execution environment. However, there are still aspects of the Java execution environment that are not exploited. One of the most distinctive features of this platform is the knowledge that the Java Virtual Machine (JVM) has about the runtime characteristics of the applications. In this paper we show an example of how the JVM could use the applications runtime information to help the operating system in some tasks related to resources management, improving significantly the performance of the Java applications. In particular, we focus on the memory management and on the memory prefetching strategy. We show that it is possible to improve the performace from traditional prefetch approaches. In order to prove this idea we have designed and implemented a prototype to add memory prefetching into de JVM. This prototype achieves to overlap the time required to read the memory from disk with the computing time of the application, improving the final performance of the applications.

1 Introduction

The growing popularity of the Java language is easily understandable analyzing the specific features that this language and its execution environment offer to the programmers. However, there are aspects of this platform that are not exploited and that could revert into better performance for the execution of applications. For example, one of the most distinctive features of the Java execution environment is that applications are not executed directly on the underlying hardware, but they are executed on a Java Virtual Machine (JVM). This is the key principle for the Java paradigm of portability. The Java compiler translates the Java code into an intermediate bytecode independent of the physical platform. Then, at runtime, the JVM is responsible for interpreting and translating the application intermediate bytecode into native machine code. Thus, the JVM has the whole knowledge about the code and the runtime characteristics of the Java application. With this information it is possible to estimate the application requirements during execution and, therefore, it could help to take management decisions that favor the application performance. This particular feature of the Java execution environment is an important difference compared to execution environments for traditional compiled languages. In these environments, the available runtime information of the applications is very limited and, therefore, the operating system (OS) is only able to offer general management policies that do not consider specific requirements of the applications.

In this paper we show an example of how the JVM could use the runtime information to help the OS in some tasks related to resources management, improving the performance of the Java applications. In particular, we focus on the memory management and on the memory prefetching strategy. The effectiveness of a memory prefetching strategy depends on the accuracy of the information about the runtime behavior of the application and about the runtime system conditions. We show that using the information available to the JVM it is possible to improve the performace from traditional prefetch approaches. In order to prove this idea we have designed and implemented a prototype to add memory prefetching into de JVM.

The rest of the paper is organized as follows. First, we justify in Section 2 that the JVM is the appropriate component to develop the prefetching task. In Section 3 we present our prototype to prefetch in Java and in Section 4 we discuss some interesting aspects related to the implementation. In Section 5 we present the evaluation of our prefetching prototype. Finally, we present some related work in Section 6 and in Section 7 we present the conclusions from this work and our future work.

2 Preliminary Discussions

In our work we propose to take benefit from the knowledge that the JVM has about the runtime behavior of the applications to add memory prefetch to the Java Execution Environment. In this section we argue that the JVM is the most suitable component of the Java Execution Environment to direct the prefetch operations, compared to other optimization strategies. Then, we show that there are Java applications that can improve its memory performace if they execute in an environment provided with an effective prefetch.

2.1 Prefetch directed by the Java Virtual Machine

The JVM is the responsible for executing the applications code. Therefore, at runtime, it has the whole control about which is the current instruction and the parameters of such instruction. That is, the JVM is able to determine if a given instruction is going to access an object, what characteristics have the object (addresses, type and size), and the position of the object to be referenced. With this information, it is possible to get an accurate accesses pattern for the objects. This accesses pattern can be used to predict the next pages referenced by the application and, therefore, to prefetch them. We have to remark that the JVM deals with the application binary code and, therefore, it is not necessary to have available the application source code in order to get this pattern. In addition, as we are considering a runtime prefetch strategy, it is also possible to get enough information about the execution environment to get the ideal prefetch distance, that is, the anticipation needed to ask for the predicted pages with time enough to be loaded before they are referenced.

An alternative to this strategy could be to modify the Java compiler to implement a static prefetch. This strategy consists on analyzing the application code and inserting the appropriate prefetch operations into the generated bytecode. Although the compiler is able to implement prediction algorithms with a reasonable hit rate, the characteristics of the physical execution platform have to be known. Therefore, this approach is not portable because it would be necessary to recompile the applications for each different hardware platform. This recompilation requirement could be avoided by encapsulating the prefetch code into dynamic libraries and linking the application with them. This way it could be possible, for each target machine, to have a suitable version of the prefetch code, and the application could use it without recompiling it again. However, this alternative requires the user to have the application source code available, and this is not always possible.

Finally, we could consider to add a dynamic prefetch strategy into the OS. However, although the OS has a whole knowledge about the execution conditions, it lacks enough information to characterize the behavior of the application accesses. On the one hand, it is not aware of each access to memory, but only of those causing page faults. This reduces the amount of information available to deduce the memory access pattern of the application. On the other hand, the OS does not know the layout of the application heap. That is, it does not know which logical addresses belong to each object, because the JVM manages the heap transparently to the system. Thus, the OS is unable of deducing the object accesses pattern and, without this information, it is complicated to offer an efficient prefetch of pages for objects. In summary, in order to implement an effective prefetch, both the compiler and the OS have limitations that the JVM overcomes.

2.2 Benefits from memory prefetching in Java

In previous work, we have evaluated the performance of virtual memory management for memory intensive Java applications [1]. We worked on the benchmark suite provided by the JavaGrande group [5], which works on the use of Java for high performance computing. In that work, we showed that there are Java applications with an execution time heavily influenced by the memory management time. In previous research, all the efforts to improve the memory management performance for Java applications have been focused on the JVM memory management code; however, we showed that there are applications that seldom require executions of this code and, therefore, they require another kind of optimizations in order to speedup their execution.

In order to determine if page prefetching is a feasible method to speedup this kind of applications, we have analyzed deeply the use of the memory that have these applications. In particular, we have analyzed the kind of allocated objects, their size and type. We have presented the results of this analysis in [2] and they show that, although these applications allocate few objects larger than 16 pages, these objects fill most of the heap and they are usually large arrays.

Arrays are usually accessed within loops and with an strided pattern [7], which is a behavior easy to detect in runtime. Therefore, given a large array it is possible to diminish the cost of accessing it, if we detect the stride between the accesses of the application, and we load on physical memory the next page of the array that the application is going to access before this access becomes effective.

3 Designing effective page prefetch in Java

In this section we present our first prototype for the pages prefetch in Java, which will show the benefits from adding a prefetch directed by the JVM. We focuss on accesses to large arrays and we discuss about the requirements needed to get an effective prefetch, and how our design takes advantage of the Java execution environment to accomplish these requirements. The requirements for an efficient prefetch are:

- A good prediction algorithm: we need to predict which is the next page of the array that the application is going to reference. Therefore, it is important to have a high hit rate in these predictions. As we have said before, arrays are usually accessed with an strided pattern. Therefore, we just need to keep for each instruction the stride used to access the array.

- To anticipate the application access: we need to load the predicted pages before the application requires them. Usually, the instruction that accesses an array is part of a loop. Therefore, for each instruction that accesses a large array, we can prefetch the page that it will reference in the next loop iterations. The distance of the prefetch, that is, how many iterations to anticipate, is a parameter of the prefetch that needs to be tuned.

- To overlap the page loading time with the application computing time, in order to actually save the time required to load them. As the OS does not offer this functionality, we have added to the JVM an execution flow (*prefetcher*) that loads the predicted pages while the JVM continues the execution of the application.

In order to achieve these goals, we have modified the routine that initializes the JVM to add the creation of the *prefetcher*, and we have added the *prediction code* to the JVM routines that manage the bytecodes of array access.

In Figure 1 we show the steps involved in each access to a large array. When an array access is being executed (1), the prediction code initially updates the information about the stride used in the current instruction to access the array. Then, this information is used to decide whether to prefetch and, in this case, which page to prefetch (p with a distance d in the figure). The prediction code emits the request to load in advance the selected pages (2) and continues the execution of the original array management code for that bytecode (3). Concurrently, the prefetcher gets the prefetching request and executes an array access (3). If such memory page is not present in memory this access will cause a page fault so that the OS will load it from the swap area (4).

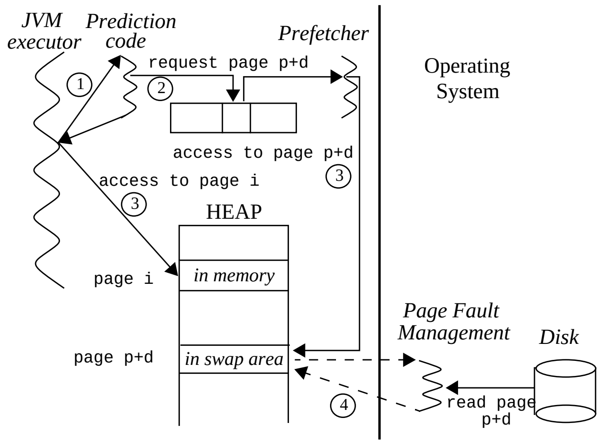

Figure 1. Prefetch design

4 A prototype implementation for prefetching

An additional requirement to achieve an effective prefetch is an efficient implementation for the mechanism. That is, the prefetch code added to the original JVM should be executed fast enough not to hide the benefits of the prefetch itself.

We have developed this prototype on the Java 2 SDK Standard Edition from Sun for Linux. We have selected the server configuration of HotSpot Virtual Machine version 1.3.1., because this configuration is oriented to applications with high resources consumption. We have used the 2.4.18 kernel version of the Linux OS.

4.1 The first step: prediction code

This part of the prefetch mechanism is very simple, because of the selected prediction algorithm: strided based on objects. However, there are some implementation decisions that we have taken in order to get an efficient code.

Pages prediction

The prediction code is executed for each bytecode that accesses a large array. Therefore, to diminish the overhead of this additional code it is important to write it carefully and to reduce the number of instructions executed. The stride computation is an example of how to save instructions. Usually, the stride that a loop instruction uses to access an array is fixed for all the iterations. Thus, we use the two initial consecutive pages that each instruction references to compute just once the stride of the accesses, and we store this value for consulting it in the rest of iterations.

In order to be able of managing the accesses to matrices we consider the possibility of having nested strides for each instruction. Usually, the accesses to a matrix are row or column based. Thus, an internal stride is used to access all the elements of a given row or column, and an external stride is used to change the target row or column. Our code detects this kind of accesses and keeps both strides as well as the number of consecutive accesses to a given row or column. This number of accesses is used to decide when it is necessary to use the external stride to predict.

Using the current accessed address, the appropriate stride, the distance of prefetch, and the bounds of the array, our code predicts which pages this instruction will reference in the next iterations. In our current prototype both the prefetch distance and the number of pages to request for each prediction, are execution parameters that we have added to the JVM. However, in future versions of this implementation we plan these variables to be dynamically adjusted at runtime.

Filtering useless requests

Another consideration for achieving an efficient implementation is to try to avoid useless requests for prefetching. For example, when the predicted page matches the last predicted page (and, thus, it is a redundant request), or when the predicted page is already present on physical memory, or when the memory system is overloaded and using prefetch would add more pressure to the system instead of helping it.

First of all, our prediction code filters those redundant requests that would result from consecutive accesses to the same page. We store for each instruction which was the page referenced in the last iteration, and we only predict if the address accessed in the current iteration belongs to a different page. The other two filtering criteria are not easy to implement with the current OS interfaces. In this prototype, we have just implemented a heuristic that help us to eliminate some of the requests related to already loaded pages.

Our heuristic focuses on matrices accesses and consists on detecting some repetitive accesses to the same set of pages to avoid the redundant requests that would involved. Our code determines the number of columns or the number of rows belonging to a given set of pages and, therefore, it is able of deducing if the new target row or column would involve the same set of pages.

4.2 The second step: prefetcher

In order to overlap the loading of the prefetched pages from disk with the computing time of the application, we have added another execution flow (*prefetcher*) that, concurrently with the execution of the applications, accesses the requested pages: if the page is already loaded on memory, the cost of the access is negligible; if the page is not loaded, then the access is a page fault that the OS solves reading the page and updating the page table.

Although the prefetcher code is a very simple loop, there are several points to decide in this implementation that can heavily affect the final performance. We have two different options to implement the new flow of execution into the JVM: we can implement it as a lightweight process or as an independent process. Although it seems to be a very easy decision to take, it is necessary an evaluation to decide which of both approaches would offer better performance for our purposes. As we show in Section 4.3 we have detected that, in some OS implementations, the lightweight process approach may be penalized by internal locks of the OS.

We have evaluated both options and, in our execution platform, the lightweight approach shows better performance than the independent process approach. The main reason for this behavior is the performance of memory accesses which, in the case of the independent process, slowdowns the performance application by a factor of 3,5 (for a more detailed discussion please refer to [2]).

In order to implement the communication between the JVM and the prefetcher we use a shared circular buffer. The prediction code added to the JVM stores in this buffer the data describing each request (initial address, stride, and the number of pages to load), and the prefetcher accesses to this buffer in order to get the requests to serve. In addition, we use a system semaphore to block the prefetcher when the buffer gets empty, until the JVM stores a new request. Although this synchronism method requires the execution of two system call, the overhead added by our code is still affordable for the applications that we have evaluated (see Section 5). The other situation we have to face is the full buffer. In this situation we have decided to prioritize the newest requests, and we overwrite the oldest one. However, the experiments we have evaluated has shown us that the usual situation is to have just one pending request in the buffer and, thus, the prefetcher is able of managing all the requests.

Another important characteristic to consider is the CPU scheduling policy of the OS. This policy decides when the prefetcher gets the CPU and, therefore, it influences the time required to load the page. We have to remark that it is desirable to have low values for the prefetch distance, because this allows the prefetcher to avoid more page faults. For this reason, we increase the scheduling priority of the prefetcher marking it as a *real time thread*, which forces the OS to assign the CPU to it whenever it becomes ready to execute. For the applications we have executed, although the performance is better assigning the real time priority to the prefetcher, the difference is always less than 2%. However, we suspect that this difference may become greater with applications with a high percentage of computing time.

4.3 Influence of the operating system

As usual, the implementation presented depends on the interface provided by the OS. But we also have detected some implementation details of the OS that heavily affect to the efficiency of our code.

Kernel prefetching

The Linux kernel implements some kind of memory prefetching, very limited because it lacks information about the applications access pattern. Linux has a system variable that indicates how many pages to load whenever it has to solve a page fault. The default value for this variable is sixteen pages, although the administrator of the machine can modify this value. Note that changing this parameter affects all the applications and accesses. Thus, whenever a page fault happens, the Linux default behavior consists on loading the target page as well as the consecutive sixteen pages, without any consideration about the access pattern nor the contents of the pages. The concurrent execution of the kernel prefetch and our prefetcher is redundant, in the best of the cases, and may penalize the final performance of the system in a general case. This is because the kind of access pattern that can benefit from the kernel prefetch is a subset of the kind of access pattern that can benefit from our prefetch proposal. For this reason, we have deactivated this kernel prefetch feature in order to evaluate our proposal.

Memory replacement

Another collateral effect that we have observed is related to the execution of the replacement of physical memory. In Linux, this task is developed by a kernel thread (*kswap daemon*). In some situations, the prefetch of pages may interfere with the kswap daemon decisions, increasing the total execution time. In previous versions of the Linux kernel this effect may become very pronounced. However, this memory management task has been redesigned in more up-to-day versions of the kernel, and this has been reverted into a more beneficial interaction. We have evaluated our prefetch proposal on the Linux version 2.4.2. In this version our proposal also benefits the execution of the applications, however, the interferences with the kswap daemon in some cases reduce the benefits from our prefetcher more than 4 times, compared to the benefits obtained on our current Linux version.

Page fault management

Another negative interaction between our prefetch prototype and the OS was present in the page fault management of older versions of the Linux kernel (we have studied the 2.2.12 and 2.4.2 versions). Although this negative interaction has disappeared in our Linux version (2.4.18), this effect is an interesting example of how the OS implementation can condition some decisions of our prefetch implementation. Former versions of Linux used a coarse grain locking mechanism to guarantee the consistency of the memory management data structures. This mechanism allows just one thread of an application to have a page fault pending to solve, even if the page faults are completely independents and could be solved without interferences. This may result into an undesirable behavior for our prefetch mechanism, because the JVM could block due to the page faults caused by the prefetcher.

Lessons learned

The sensitivity of our prefetch prototype to changes into the kernel implementation is something to be expected, as we are adding a memory management task into the user level that executes transparently to the kernel. Thus, this code interacts with the memory management tasks of the kernel and, if the kernel code changes so it does the interaction with our code.

As we show in Section 5, with this prefetch prototype we get the goal of showing the potential benefits from a prefetch directed by the JVM. However, the final design of this strategy should avoid the interferences between both management levels. In order to eliminate these interferences we should design the prefetching mechanism as a cooperative task between the JVM and the kernel. That is, the JVM should decide which pages to prefetch, taking benefit of the knowledge it owns about the application behavior, and the OS should carry out the page prefetch. This implies modifying the kernel to offer this feature.

5 Evaluation

In this section we present the results from the experiments we have executed to evaluate our prefetch prototype and, thus, to show that our prefetch approach is a feasible method for improving the performance of Java applications that use large arrays. We have run our experiments on a 500 MHz Pentium III processor, with 128 Mb of physical memory.

In order to evaluate the memory management performance, we have counted and classified the page faults caused by the JVM. We separate the page faults caused by the application code from those caused by the prefetcher. Moreover, we also separate the page faults caused by accessing a page that has already been requested from the swap area but it is still in transit (this may happen, for example, if the prefetcher requests the page without enough anticipation). In addition, we have measured the time dedicated to solve each page fault type, as well as the total application time. These measurements require us to modify the Linux kernel, to add the counters and the code that manipulates them. We have also added to the JVM the code to configure the counting and to get the results. We have implemented this new interface between the JVM and the kernel through a kernel driver. Finally, we deactivate the Linux memory prefetching for the evaluations of our proposal.

We have chosen for these experiments the matrix multiplication, because this is an often used kernel in programs dealing with matrices. The size of each element of the matrices is 8 bytes (doubles). We have run this algorithm on different sizes for the input matrices, to study the behavior of our prefetch proposal on different conditions of physical memory availability. In Figure 2 we show the results obtained with a data set input small enough to remain on memory during all the program execution. The dimensions of both input matrices is 500x500 and, thus, the multiplication loops execute with no page fault and the kernel prefetch has no effect on its execution (see Figure 2.a). In addition, our prefetch is useless and, therefore, the results show us the overhead added by our code when no benefit is possible (see Figure 2.b). Comparing results from the original JVM with results from the JVM with prefetch, we can see that this overhead is around 12%. We have to remark that at runtime it is possible to detect if the data set of an application fits on physical memory and, thus, the JVM can deactivate for this application the prefetch feature, avoiding this way the overhead of the additional code (see Section 7).

We have also used this case to study how the filters presented in Section 4 affect to the performance of our prefetcher. We have seen that if we do not use the heuristic for detecting some repetitive accesses to a set of pages the execution time of the prefetcher multiplies by a factor higher than 6. In addition, if we do not use the filter to avoid consecutive predictions to the same page, the execution time of the prefetcher multiplies by another factor higher than 3. These

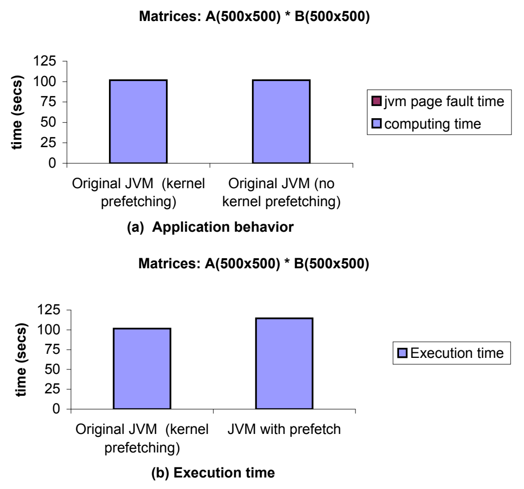

Figure 2. Small matrices multiplication

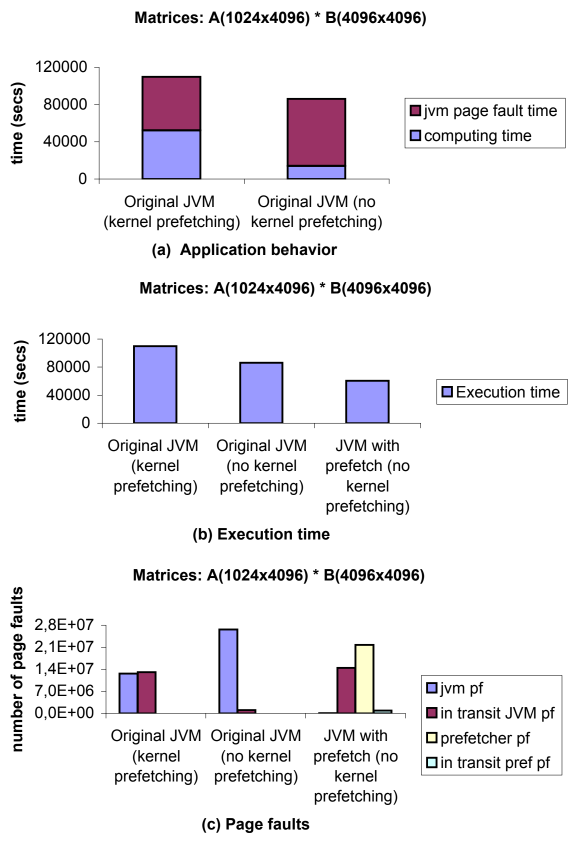

Figure 3. Large matrices multiplication

results confirm to us the importance of reducing as much as possible the number of request to the prefetcher.

In Figure 3 we show the results for a data set input that do not fit on our available physical memory. The dimension of the matrix traversed by rows (A) is 1024x4096 (32Mb). The dimension of the matrix traversed by columns (B) is 4096x4096 (128 Mb). With these matrices, it is possible to maintain simultaneously in memory the target row of matrix A and the columns from several physical pages of matrix B. In this way, while the JVM is accessing the columns from a given set of pages our prefetcher is able of loading the columns for the next set of pages. This is the best scenario that this algorithm can offer to our prefetch approach.

We can see in Figure 3.a that, if we execute this application with the kernel default behavior, 52% of the total application time is spent solving page faults. If we execute the same application deactivating the kernel prefetch, the page fault time becomes 84% of the total time. Ideally, if the prediction algorithm of the prefetcher always hits the prediction and the distance of prefetch is the suitable, then the prefetcher should be able to reduce the execution time of the application by the time it originally spent solving page faults. However, we have to consider that this reduction of the execution time is based on overlapping the time required for loading the pages with the application computing time and, therefore, the potential improvement depends not only on the application page fault time but also on the application computing time.

In Figure 3.b we show the total execution time of the applications in the three considered cases: executed with the original JVM and the default kernel behavior, executed with the original JVM but deactivating the kernel prefetch feature, and executed with the JVM that includes our prefetch feature. We can see that the kernel prefetch penalizes the execution of this application, and we can improve significantly its performance only deactivating this feature. However, if we execute the application with our prefetch proposal, the performance is still improved, reaching an improvement of around 45% with respect to the original execution, and an improvement of around 30% with respect to the original JVM deactivating the kernel prefetch.

In Figure 3.c we show the number of page faults caused in each case. We can see that the execution with our prefetch proposal achieve the JVM only to cause *in-transit page faults*, that is, the prefetch always requests the page for the JVM, but not with enough anticipation. However, the reduction on the total execution time indicates that the slowdown due to these *in-transit page faults* is not significant.

Finally, we have selected a third data set input to increase the stress of the memory system. With these matrices, it is possible to keep simultaneously in physical memory the target row of A, and the columns of B of just one set of pages. With this scenario, our prefetcher should request a page for the next set of columns only when it detects the last reference to a page for the current set of columns. However, the elapsed time between this situation and the access to the next columns is very small and, therefore, it is difficult for our prefetcher to be on time. We have seen than, although our prefetch is not able to load on time the required pages, the performance of the application does not diminish [2].

6 Related work

As Java is growing in popularity we can find a lot of work that study the memory management of the JVM execution environment. However, all this work do not consider applications working on large objects that survive through all the execution. On the contrary, it is focused on improving the performance of applications that execute a lot of allocations and deallocations of small objects and, therefore, garbage collection is a key issue for these studies. For example, there are proposals for garbage collection algorithms that do not alter the data locality [3, 9]; there are also policies for placement of objects that decrease the time of garbage collection and that also favors the data locality [9, 10, 8].

We can also find proposals for cache prefetching in Java [6]. This work considers small arrays and linked lists of objects, and proposes to modify the compiler to analyze the source code and insert prefetch operations into the application code. They simplify traditional techniques of static prefetch at compile time, in order to incorporate them to the JIT compiler and obtain a dynamic prefetch, but they leave as future evaluation the suitability of this approach.

Out of the Java scope, in [4] work there is a proposal for virtual memory prefetch at runtime that consists on implementing cooperation between the compiler and the OS, using a runtime layer. The compiler analyzes the code of the application, inserting the prefetch instructions that the static analysis considers adequate. Before executing each prefetch operation, the runtime layer checks if is really necessary to prefetch the required page, and, this way, eliminates unnecessary prefetch operations. Finally, the OS may also ignore prefetch operations if the current conditions of execution are not favorable.

Our proposal takes advantage of the JVM to avoid the modification of the source code and, therefore, preserve portability, and we decide at runtime if the execution conditions recommend to execute prefetch.

7 Conclusions and future work

In this paper we have proposed a new approach to tackle the performance of Java applications. The idea is to take advantage of the knowledge the JVM has on the running application to take decisions usually taken by the OS, which does not have accurate information on the application behavior. To prove our idea, we have implemented a prototype of a pages prefetcher at the JVM level. This prefetcher has accurate information of data structures used and the real access pattern (not fault pattern as the OS has). We have tested it with a matrix multiply obtaining a significant improvement. Finally, we have detected that cooperation between the OS and the JVM is desirable. We have seen that the performance of a prefetch mechanism implemented only in user level depends heavily on implementation details of the memory management tasks offered by the OS. Thus, the evaluation of the prefetcher on different versions of the OS has produced different performance results. In addition, many heuristics have to be used in the current prototype that would not be needed if the OS cooperates with the JVM.

As future work, we plan to study with more detail how to add to our prefetch prototype the cooperation with the OS. In this new design the JVM will take the prefetch decisions considering the information it owns about the application behavior and the information the OS has about the memory system state. This information will replace our current heuristics used to filter useless requests. Then, the JVM will ask the OS to load them in advance.

References

[1] Y. Becerra, T. Cortes, J. Garcia, and N. Navarro. Evaluating the importance of virtual memory for java. In *Proceedings of the IEEE 2003 International Symposium on Performance Evaluation of Systems and Applications*, Austin, TX, March 2003.

[2] Y. Becerra, J. Garcia, T. Cortes, and N. Navarro. Memory prefetching managed by the java virtual machine. Technical Report UPC-DAC-RR-2006-2, Computer Architecture Dept., UPC, Barcelona, Spain, 2006.

[3] H.-J. Boehm. Reducing garbage collector cache misses. Technical Report HPL-2000-99, Hewlett-Packard Laboratories, Palo Alto, CA, 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] B. Cahoon and K. S. McKinley. Simple and effective array prefetching in java. In *Proceedings of the ACM Java Grande Conference*, pages 86–95, Seattle, WA, November 2002.

[7] J. González and A. González. Speculative execution via address prediction and data prefetching. In *Proceedings of the ACM 1997 International Conference on Supercomputing*, pages 196–203, Vienna, Austria, July 1997.

[8] 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.

[9] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh. Exploiting profilic types for memory management and optimizations. In *Proceedings of the ACM Symposium on Principles of Programming Languages 2002*, Portland, OR, January 2002.

[10] Y. Shuf, M. Gupta, H. Franke, A. Apple, and J. P. Singh. Creating and preserving locality of java applications at allocation and garbage collection times. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2002*, Seattle, WA, November 2002.