John von Neumann Institute for Computing

Performance Evaluation of Barrier Techniques for Distributed Tracing Garbage Collectors

J.M. Velasco, D. Atienza, K. Olcoz, F. Catthoor

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 549-556, 2006.

Performance Evaluation of Barrier Techniques for Distributed Tracing Garbage Collectors

Jose M. Velasco[†], David Atienza[†⋆], Katzalin Olcoz[†], Francky Catthoor[*]
[†]DACYA/UCM, Avda. Complutense s/n, 28040 Madrid, Spain.
Email: mvelascc@fis.ucm.es, {datienza, katzalin}@dacya.ucm.es
[⋆]IMEC vzw, Kapeldreef 75, 3001 Heverlee, Belgium.
Email:{Francky.Catthoor, atienza}@imec.be

Abstract. Currently, software engineering is becoming even more complex due to distributed computing. In this new context, portability is one of the key issues and hence a cluster-aware Java Virtual Machine (JVM) that can transparently execute Java applications in a distributed fashion on nodes of a cluster, while providing the programmer with the single system image of a classical JVM, is really desirable. This way multi-threaded server applications can take advantage of cluster resources without increasing their programming complexity.

However, such kind of JVM is not easy to design. Moreover, one of the most challenging tasks in its design is the development of an efficient, scalable and automatic dynamic memory manager. Inside this manager, one important module is the automatic recycling mechanism or garbage collector. This collector is a module with very intensive processing demands that must concurrently run with user's application. It can consume a very significant portion of the total execution time spent inside JVM in uniprocessor systems, and its overhead increases in distributed garbage collection because of the update of changing references in different nodes. Hence, the garbage collector is a very critical part in distributed designs of JVMs, both for performance and energy.

In this paper our contribution to automatic distributed garbage collection is two-fold. First, we have analyzed the barrier mechanism design space for the study of tracing-based distributed garbage collectors. Second, we have evaluated the impact of the most significative barrier strategies as main bottlenecks in global performance. Our preliminary results show that the choice of the specific technique used in barrier mechanisms produces significant differences both in performance and inter-nodes messaging overhead.

1. Keywords

Software engineering, clusters, runtime support, distributed garbage collection.

2. Introduction

A cluster-aware Java Virtual Machine (JVM) can transparently execute java applications in a distributed fashion on the nodes of a cluster while providing the programmer with the single system image of a classical JVM. This way multi-threaded server applications can take advantage of cluster resources without increasing the programming complexity.

When a JVM is ported into a distributed environment, one of the most challenging tasks is the development of an efficient, scalable and fault-tolerance automatic dynamic memory manager. The automatic recycling of the memory blocks that are no longer used is one of the most attractive characteristics of Java for the programmer and the software engineer since engineers do not need to worry

[*]This work is partially supported by the Spanish Government Research Grant TIC2002/0750.

about designing a convenient dynamic memory management, i.e. missusing the available memory because incorrect intervals for memory allocation and deallocation are provided. This automatic process, very well-known as Garbage Collection (GC), makes much easier the development of complex parallel applications that include different modules and algorithms with different dynamic memory behaviors that need to be taken care of from the software engineering point of view. However, since the garbage collector is an additional module with intensive processing demands that runs concurrently with the application itself, it always attains for a critical portion of the total execution time spent inside the virtual machine in uniprocessor systems. As Plainfosse and Shapiro[7] point out, distributed GC is even harder because of the difficult job to keep updated the changing references between the address spaces of the different nodes.

In this paper we consider a distributed memory heap partitioned into disjoint spaces. Spaces communicate with each other by message passing and the allocation can be local or remote. Spaces records the entering references and use them as living roots for tracing purposes. In addition, in our approach the global marking phase is concurrently executed with the application.

Intuitively, the weakness of tracing algorithm relies in the cost of barriers and lack of scalability due to the amount of messages going through the processing nodes. In fact, our preliminary results show that the choice of the technique used in write-barriers produces significant differences both in performance and inter-nodes messaging overhead.

This rest of the paper is organized as follows. We first describe related work on both distributed garbage collection and uniprocessor barrier techniques. Link to this, we then overview the main topics in distributed tracing garbage collection. Next, we present our proposed barrier techniques design space and the configurations we have explored in this paper. Later, we describe the experimental setup used in our experiments and the results obtained. Finally, we present an overview of our main conclusions and outline possible future research lines.

3. Related Work

In this section, we discuss related work on both distributed garbage collection and uniprocessor barrier techniques. To our knowledge, no one has developed a experimental comparison among barrier techniques in a distributed context.

Plainfosse and Shapiro[7] published a complete survey of distributed garbage collection techniques. For good tutorial overview, Lins offers a chapter within the Jones's classical book about garbage collection [4].

In section 6.2 we make a presentation of dJVM from Zigman et Al. A distributed Java virtual machine on a cluster which presents a single system image to the programmer. Another similar aproach is the Java VM on a cluster from Aridor et Al[9]. This distributed virtual machine is built on top of a cluster enabled infrastructure.They have developed a new object model and thread implementation that are hide to the programmer.

In [6], Pirinen presents a methodical analysis of barrier techniques from a formal point of view and for incremental tracing. Our present work is an extension of his. In this paper, we have implemented the different options in the much stricter environment of distributed collection to make an empirical study and to look for optimal solutions.

Blackburn et Al[1], analyze the effect of inlining on write-barriers. They use *Remembered sets* with slots and objects and they measure the impact of inlining and partial inlining on execution time and compilation overload. They use Jikes RVM with the optimizing compiler.

4. Distributed Tracing Garbage Collection

In prior work, the cluster JVM from Aridor et Al([9]) and the distributed JVM (i.e. dJVM) from Zigman et Al([10]) use a conservative approach that does not reclaim objects with direct or indirect global references. In our research, we are developing a new framework for the analysis and optimization of tracing-based distributed garbage collectors by using the dJVM as background approach. Our new framework does not include any reference counting collection mechanisms [5], which are very popular in monoprocessor GC solutions, because of two reasons. First, the reference counting collector is not complete and needs a periodical tracing phase to reclaim cycles, which will create an unaffordable overhead in execution time since it needs to block all the processing nodes. Second, the update of short-lived references produces continuous messages to go back and forth between the different nodes of the distributed GC, which makes this algorithm not scalable within a cluster.

On the contrary, as we propose, tracing collectors seem to be the more convenient for creating such distributed garbage collectors. Conceptually, all consist in two different phases([5]), which are the following:

- First, the marking phase allows the garbage collector to identify living objects. This phase is global and implies scanning the whole distributed heap.
- Second, the reclaiming phase takes care of recycling the unmarked objects (i.e. garbage). The reclaiming phase is local to each node and can be implemented as a non-moving collector or as a moving collector.

During the mark phase, the collector traverses the graph of references between objects through memory recursively. As this phase makes progress, the collector segregates the objects into three sets:

- Black objects which have been marked as alive and will not be visited again during the tracing. We say that they are scanned.
- Grey objects have been noted reachable but must still be processed in order to follow the graph to their offspring. When a white object becomes grey, we call this *shading*.
- White objects have not been visited and are candidates to be recycled.

If this phase is interleaved with the program execution, the garbage collector needs to keep track of the changes in the reference graph produced by the application. To guarantee that no living object will be reclaimed, the collector must be aware of every new reference created from a black object to a white object. To this end, the JVM executes barriers to remember the locations where new pointers have been created or modificated. Therefore, the main problem associated with distributed tracing comes from the difficult synchronization between the distributed global mark phase with several local and independent recycling phases.

In the uniprocessor context, both for incremental and generational garbage collectors, it has been proven that write-barriers are the main bottleneck that degrades performance, and we believe that in distributed environments, the optimization of write-barriers is even more critical. In summary, barriers are used to intercept new references among objects and give the virtual machine a chance to be aware of this situation before it is concluded. *Read-barriers* intercept loads and *write-barriers* intercept stores. There are several techniques for maintaining the tricolour invariant. Each of them uses read-barriers or write-barriers or both.

In his paper, Pirinen define two invariants relative to the tricolour marking phase:

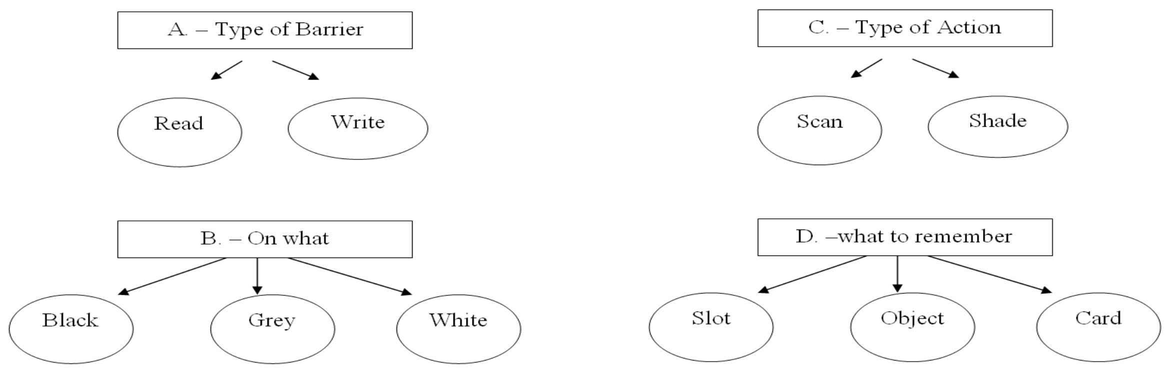

Figure 1. Design Space of Orthogonal Decisions for Barrier Techniques

- Strong invariant: There are no pointers from a black object to a white object.
- Weak invariant: All white objects pointed to by a black object are reachable from some grey object through a chain of white objects.

5. Design Space of Orthogonal Decisions for Barrier Techniques

As we have just explained, an extensive amount of possible barrier techniques (and implementations for them) exists. Therefore, all these options have to be enumerated to cover exhaustively the barriers design space. In our method, we have classified all the relevant decisions that can compose the design space of barrier mechanisms in different orthogonal decision trees (see Figure 1). Orthogonal means here that any decision in any tree can be combined with any decision in another tree, and the result should be a potentially valid combination, which does not necessarily mean that it meets all timing and cost constraints for a specific system. In addition, all possible solutions in the design space should be spanned by a combination of leaves in the orthogonal trees, just like any point in a geometrical space can be represented in a set of orthogonal axes. Moreover, the decisions in the different orthogonal trees can be ordered in such a way that traversing the trees can be done without decision iterations, as long as the appropriate constraints are propagated from one decision level to all subsequent levels. Basically, when one decision has been taken in every tree, one custom barrier techniques is defined (in our notation, atomic barrier mechanism) for a specific memory manager behaviour pattern. As a result, the four trees considered in our design space are the following ones:

- A. Type of Barrier: barriers are used to intercept the accesses among objects and give the virtual machine a chance to be aware of this situation before it is concluded. *Read-barriers* intercept loads and *write-barriers* intercept stores. There are several strategies for maintaining the tricolour invariant. Each of them uses read-barriers or write-barriers or a combination of both.
- B. On what type of objects the barrier is set. In this case the type of object is defined in terms of the tricolour marking scheme.
- C. When a barrier is hit what kind of action we must take: turn the object under the barrier black (*scan*) or turn the referent grey (*shade*). In this paper we have experimented taking only the *shade* option.

- D. As a result of the barrier, how we remember the relationship graph change:
  - Slot. We remember the object field that contains the pointer. The set of addresses of these fields are usually knom as *Remembered Set*. In this paper we have only explored this option since it is the one that seems to achieve more performance in the final distributed GCs.
  - Object. We remember the source object itself. The collector need to scan the whole object in order to find references.that the choice of the specific technique used in write-barriers produces significant differences in both performance and inter-nodes messaging overhead.
  - Card marking. This mechanism uses a table to remember fixed size of regions of memory (*cards*) as pointer sources.During collection the virtual machine needs to scan these memory regions looking for pointers.

Although the decision categories and trees presented in Figure 1 are orthogonal, certain leaves in some trees strongly affect the coherent decisions in other trees. Thus, they include interdependencies to take into account when a barrier techinque is designed. This fact, jointly with the mentioned choices for this paper that we have taken in C and D trees, have produced four barrier configurations:

- C1. A read-barrier on grey objects.
- C2. A write-barrier on black objects.
- C3. A write-barrier on both grey and white objects.
- C4. A write-barrier on grey objects and a read-barrier on white objects.

The two first configurations maintain the *Strong* invariant relative to tricolour marking scheme, while the two next ones preserve the *Weak* invariant.

6. Experimental Setup and Results

In this section we first describe the whole simulation environment used to obtain detailed memory access profiling of the JVM (for both the application and the collector phase). It is based on cycle-accurate simulations of the original Java code of the applications under study. Then we summarize the representative set of GCs used in our experiments. Finally we introduce the sets of applications selected as case studies and indicate the main results obtained with our experiments.

6.1. Basic Jikes RVM

Jikes RVM is a high performance Java virtual machine designed for research. It is written in Java and the components of the virtual machine are Java objects [3], which are designed as a modular system to enable the possibility of modifying extensively the source code to implement different GC strategies, optimizing techniques, etc. We have used version 2.3.0 along with the recently developed memory manager JMTk (Java Memory management Toolkit) [2]. Jikes RVM offers three compiler choices:

- Baseline, all methods are compiled by a quick non-optimizing compiler.
- Optimizing, all methods are compiled by an aggresive optimizing compiler.

- Adaptive, which initially compiles methods with the quick compiler, and then and after selecting hot zones, recompiles methods using the optimizng compiler. The hot method identification is based on application sampling and therefore it tends to produce slightly different choices for each compilation.

At runtime Jikes RVM compiles not only the application classes, but also the virtual machine classes. Since several classes are needed for bootstrapping the VM, Jikes can be configured with two levels of VM classes precompilation at build-time:

- A minimal configuration that only precompiles those classes essentials for booting the VM.

- A fast running configuration that precompiles as much as possible, including key libraries and the optimizing compiler.In this paper we have used this configuration.

Jikes code is scattered with a lot of assertion checking code that we have disabled for our experiments.

6.2. Distributed Jikes RVM

We have employed as our baseline framework to modify the Distributed Jikes RVM (dJVM)[10] developed at the Australian National University. It provides a single system image to Java applications and so it is transparent to the programmer. The dJVM employs a master-slave architecture, where one node is the master and the rest are slaves. The boot process starts at the master node. This node is also responsible for the setting up of communication channels among the differents slaves. The class loader runs in the master. In dJVM are objects available remotely and object which have only a node local instance. This is achieved by using a global and a local addressing schemes for objects. The global data is also stored in the master with a copy of its global identifier in each slave node. Each object has an associated universal identifier (UID) that uniquely identifies the object in the whole cluster. Each node has a range of UIDS of its own. Instances of primitives types, array types and most class types are always allocated locally. The exceptions are class types which implement the *Runnable* interface.

As the goal of this paper was the measurement of different barrier techniques, we have changed dJVM in order to force a bigger amount of remote object allocations. This way the performance is degraded but it gives us a better opportunity for analyzing the barriers influence.

6.3. Case Studies

We have applied the proposed experimental setup to dJVM running the most representive benchmarks in the suite SPECjvm98 [8]. These benchmarks could be launched as dynamic services and extensively use dynamic data allocation. The used set of applications is the following:

_201_compress: it compresses and then decompresses a large file.

_202_Jess: it is the Java version of an expert shell system using NASA CLIPS. It is compound fundamentally of structures of sentences if-then.

_205_Raytrace: raytraces a scene into a memory buffer. It allocates a lot of small data with different lifetimes. [4] _209_db: This benchmark reads a 1 MB size file and then it performs multiple database functions on memory.

_213_javac: it is the java compiler. It has the highest program complexity and its data is a mixture of short and quasi-inmortal objects. problem _222_mpegaudio: it is an MPEG audio decoder.

_227_Jack: it is a parser based on the Purdue Compiler Construction Tool Set (PCCTS). A parser determines the syntactic structure of a chain of symbols received from the exit of the lexical analyzer.

Table 1
Summary of results, normalized against C1 (read-barrier on grey objects)

	Execution Time				Messaging Overhead				Remembered Set Size			
	C1%	C2%	C3%	C4%	C1%	C2%	C3%	C4%	C1%	C2%	C3%	C4%
compress	100	94.4	85	95.5	100	92.2	77	83	100	90	50	76
Jess	100	92.4	73	79.9	100	94.4	66	71.5	100	91.6	84	77.5
Raytrace	100	91.6	84	77.5	100	96.3	55	75.5	100	92.4	73	79.9
db	100	98.4	87	75.5	100	94.4	73	72	100	94.4	88	72.5
javac	100	95.2	69.5	78.3	100	92.4	58	75.5	100	94.6	75	83
mpegaudio	100	94.4	88	72.5	100	93.4	55	75.5	100	94.4	55	75
Jack	100	94.6	75	83	100	94.4	68	79	100	90.8	64	75.8
mtrt	100	90.8	64	75.5	100	94.4	67	77	100	94.4	55	75.5

_228_mtrt: it is the multi-threaded version of _205_raytrace. It works in a graphical scene of a dinosaur. It has two threads, which make the render of the scene removed from a file of 340 KB.

The suite SPECjvm98 offers three input sets(referred as s1, s10, s100), with different data sizes. In this study we have used the biggest input data size, represented as s100, as it produces a bigger amount of cross-references among objects.

6.4. Experimental Results

In our experiments we have utilized as hardware platform an eight nodes cluster with Fast-Ethernet communication hardware. The networking protocol is TCP/IP. Each node is a Pentium IV, 866MHz with 1024Mb and Linux Red Hat 7.3. In Table 1 we summarize our experimental results. As we can see, the dichotomy between Strong and Weak invariant is resolved in favour of the latter. As a result, the C3 and C4 configurations obtain always better results both in execution time and in inter-node messaging overhead. Furthermore, for those configurations based on full or partial read-barriers, our results indicate a significant decrease on performance due to the unnecessary messages sent between the nodes, in comparison to write-barriers only schemes. Therefore, the configuration C3, with a write-barrier on both grey and white objects achives the better results. Following this previous reasoning of limiting the amount of messages per processed data, the benefits of C3 are shown even more clearly on those benchmark with more allocated data as _205_Raytrace or _228_mtrt.

7. Conclusions and Future Work

The barrier technique is a key factor relative to performance in uniprocessor garbage collectors. Related to this well-known problem, in a distributed context the increase of inter-node messaging problem is added. In this paper we have first presented the design pace of orthogonal decisions for barrier techniques. Second, we have evaluated four representative barrier mechanism configurations. Two of them maintain the *Strong* tricolour marking scheme invariant, and two that preserve only the *Weak* invariant to achieve faster execution. Our preliminary results show that the choice of the specific technique used in barrier mechanism produces significant differences in both performance and inter-nodes messaging overhead.

As future work we intend to extend our experimental results to different configurations, some of them maybe not even considered before in distributed garbage collection due to their limited design space, in order to cover the C and D trees of our complete barrier techniques' design space.

Acknowledgements

This work is partially supported by the Spanish Government Research Grant TIC2002/0750.

References

[1] Stephen M. Blackburn and Kathryn S. McKinley. In or out? putting write barriers in their place. In *Proceedings of SIGPLAN 2002 International Symposium on Memory Managment, ISMM'02, Berlin, June, 2002*, ACM SIGPLAN Notices. ACM Press, June 2002.

[2] IBM. The jikes' research virtual machine user's guide 2.2.0., 2003. `http://oss.software.ibm.com/developerworks/oss/jikesrvm/`.

[3] The source for java technology, 2003. `http://java.sun.com`.

[4] Richard Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 4th edition, July 2000.

[5] Richard Jones and Rafael D. Lins. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[6] Pikka Pirinen. Barrier techniques for incremental tracing. In *Proceedings of International symposium on Memory Management*, Vancouver, Canada, September 1998.

[7] David Plainfoss and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proceedings of International Workshop on Memory Management*, September 1995.

[8] SPEC. Specjvm98 documentation, March 1999. `http://www.specbench.org/osg/jvm98/`.

[9] Michael Factor Yariv Aridor and Avi Teperman. A distributed implementation of a virtual machine for java. *Concurrency and Computation: practice and experience*, 13:221–244, 2001.

[10] John Zigman and Ramesh Sankaranarayanara. djvm - a distributed jvm on a cluster. Technical report, 2002.