Reducing Memory Sharing Overheads in Distributed JVMs

Marcelo Lobosco[1], Orlando Loques[2] and Claudio L. de Amorim[1]

[1] Laboratório de Computação Paralela, PESC, COPPE, UFRJ
Bloco I-2000, Centro de Tecnologia, Cidade Universitária, Rio de Janeiro, Brazil
{lobosco, amorim }@cos.ufrj.br
[2] Instituto de Computação, Universidade Federal Fluminense
Rua Passo da Pátria, 156, Bloco E, 3º Andar, Boa Viagem, Niterói, Brazil
loques@ic.uff.br

Abstract. Distributed JVM systems enable concurrent Java applications to transparently run on clusters of commodity computers by supporting Java's shared-memory model over multiple JVMs distributed across the cluster's computer nodes. In this work, we developed and evaluated selective dynamic *diffing* and lazy home allocation, two new techniques that enable distributed JVMs to efficiently support memory sharing across the cluster. Specifically, the two techniques whether in isolation or in combination can reduce the overheads due to message traffic, extra memory space, and high latency of remote memory accesses that such distributed JVM systems require to enforce memory coherence. To evaluate the performance-related benefits of dynamic *diffing* and lazy home allocation, we implemented both techniques in CoJVM (Cooperative JVM), a distributed JVM system we developed in previous work. We carried out a performance comparison between basic CoJVM and CoJVM versions that used our proposed techniques for five representative concurrent Java applications (MM, LU, Radix, FFT, and SOR). Our experimental results showed that dynamic *diffing* and lazy home allocation reduced significantly memory sharing overheads, which in turn resulted in considerable gains in CoJVM system's performance, ranging from 9% up to 20% in four out of five applications with speedups varying from 6.5 up to 8.1. ***Keywords*: JVM, distributed shared memory, Java, cluster computing, concurrent Java applications, high-performance computing.**

1 – Introduction

In previous work [1], we introduced a new Java [2] environment for high-performance computing, namely the **Co**operative **J**ava **V**irtual **M**achine (**CoJVM**). We reported performance of CoJVM ranging from 5.4 to 7.7 on 8-node cluster for several parallel Java benchmarks. The motivation behind the development of CoJVM was the fact that the use of Java's concurrency model for developing parallel applications was limited to costly shared-memory computers although clusters of low-cost computers offered an even more cost-effective computing platform for high-performance computing. Therefore, CoJVM represented a first step towards providing a distributed shared-memory Java platform that could execute efficiently and transparently standard Java abstractions for concurrent programming.

Despite the good speedups CoJVM achieved we noticed that there was considerable room for further performance improvements. Specifically, we discovered a large imbalance in page distribution among the processing nodes for most applications we tested. It is well-known that uneven page distribution impacts severely the locality of reference, which in our case increased substantially the coherence protocol overheads, including the amount of control and data messages, page access faults, and barrier times, which hurt CoJVM performance significantly.

In this work, we further address CoJVM performance issues with two new simple but effective techniques, namely selective dynamic *diffing* and lazy home allocation, which take advantage of the information CoJVM extracts at run-time about the application behavior. Selective dynamic *diffing* technique uses a bit vector to track shared memory positions that are modified at run-time and uses that information to create *diffs* in a most effective way. Lazy home allocation postpones the association between pages and homes until the time at which the computation phase really takes place and pages will be definitely used. Therefore, lazy home allocation will generally prevent that any single node to becoming the home of a number of pages well above the average, as it often happens under the popular first-touch home assignment policy.

Five representative kernels, MM, SOR, LU, FFT, and Radix, were used as benchmarks to evaluate the impact of our two proposed techniques on CoJVM performance. Our experimental results showed that four out five benchmarks achieved superior speedups when they were executed in optimized CoJVM versions. The only exception was MM, which performed 1% better in original CoJVM. Overall these results demonstrate the effectiveness of the proposed techniques on CoJVM performance.

This paper presents three main contributions: a) we propose and evaluate two new techniques for distributed JVMs: selective dynamic *diffing* and lazy home allocation; b) we demonstrate the effectiveness of the bit vector mechanism and associate technique we created to reduce both the amount of message traffic and extra memory space that distributed JVMs required to support a distributed shared-memory model; and c) we present detailed performance analysis of the two techniques for five parallel Java benchmarks.

The remainder of this paper is organized as follows. Section 2 overviews distributed shared memory systems and the Java virtual machine. Section 3 describes the Cooperative Java Virtual Machine. Section 4 introduces the proposed optimization techniques. In section 5 we evaluate the impact of the optimization techniques on CoJVM. Section 6 presents related works, and in section 7 we draw our conclusions and outline future works.

2 – Background

2.1 - Distributed Shared-Memory Systems in Software

Software DSM systems provide the shared memory abstraction on a cluster of physically distributed computers. This illusion is often achieved through the use of the virtual memory protection mechanism [3]. However, using the virtual memory mechanism has two main shortcomings: a) false sharing and fragmentation of pages are likely to occur due to the use of a large virtual page as the unit of coherence, which

leads to unnecessary communication traffic; and b) high OS costs associated to treating page faults and crossing protection boundaries. Several relaxed memory models, such as LRC [4], have been proposed to alleviate false sharing. In LRC, shared pages are write-protected so that when a processor attempts to write to a shared page an interrupt will occur and a clean copy of the page, called the twin, is built and then the page is released to write operations. In this way, the changes to the page, called *diffs*, can be obtained at any time by comparing the current copy with its twin. LRC requires the programmer to use of two explicit synchronization primitives: acquire and release. Coherence-related messages are delayed until an acquire is performed by a processor. When an acquire operation is executed the acquirer receives from the last acquirer all the write-notices, which correspond to changes made to the pages that the acquirer has not seen according to the happen-before-1 partial order [6]. HLRC [5] introduced the concept of home node, in which each node is responsible for maintaining an up-to-date copy of its owned pages; then, the acquirer can request copies of modified pages from their home nodes. Pages are associated with home nodes through a first touch policy: a page is associated with the node that first referenced it. At release points, *diffs* are computed and sent to the page's home node, which reduces memory requirements in home-based DSM protocols and contributes to improve the scalability of the HLRC protocol.

2.2 – Java

The Java Virtual Machine (JVM) implements a platform independent virtual machine according to the Java Virtual Machine Specification [6]. Currently, the JVM is available in many different host (hardware + operating system) platforms.

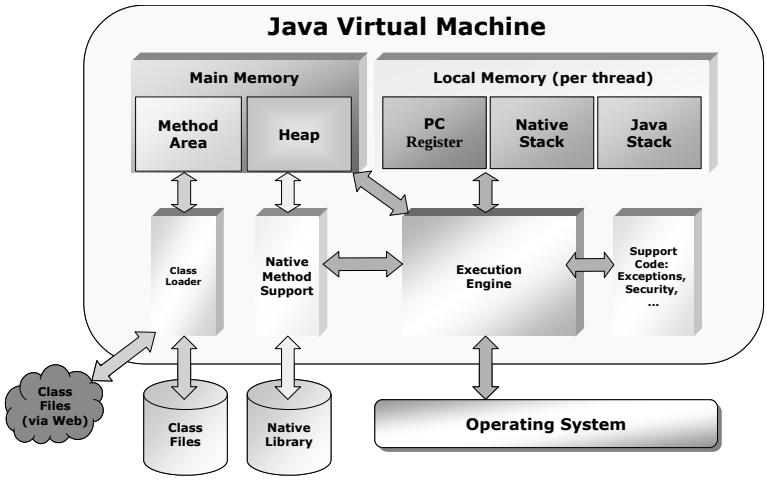

Fig. 1. Sketch of the JVM internal architecture.

Figure 1 sketches the JVM internal architecture. It includes the main subsystems and memory areas described by the Java Virtual Machine Specification. The class loader subsystem implements a mechanism for dynamically read, link, and initialize

the data types defined by the language (classes and interfaces), which can be loaded either from the file system or any machine connected to the network. The native method support subsystem is responsible for loading and executing native methods. The execution engine is the heart of the machine in the sense that it is responsible for executing instructions contained in the methods of the Java classes that are loaded by the class loader subsystem.

Each JVM instance has both a method area and a heap. The method area stores internal representations of classes and methods that are loaded by the class loader subsystem. The memory manager controls the heap by means of the garbage collection process which releases memory areas of objects and arrays the application store in the heap but which it uses no longer. Both method and heap compose the main memory which is shared by the JVM threads. In addition to those two main memory areas, the JVM also associate other private memory areas with each thread it creates: a) The PC register, which stores the address of the next instruction the thread will execute; b) the Java's stack that stores the state of methods the thread invoked, which includes local variables, parameters, return value, and intermediate results; and c) the native stack, which stores the state of native methods the thread invoked. These three areas compose the thread's local memory.

The JVM specifies the interaction model between threads and the main memory by defining an abstract memory system, a set of memory operations, and a set of rules for these operations. Each thread operates strictly on its local memory, so that variables have to be copied first from main memory to the thread's local memory before any computation can be carried out. Similarly, local results become accessible to other threads only after they are copied back to main memory. Variables are referred to as master or working copy depending on whether they are located in main or local memory, respectively. The copying between main and local memory, and vice-versa, adds a specific overhead to thread operation. The replication of variables in local memories introduces a potential memory coherence hazard since different threads can observe different values for the same variable. The JVM offers two synchronization primitives, called *monitorenter* and *monitorexit*, to enforce memory consistency. In brief, the model requires that upon a *monitorexit* operation, the running thread updates the master copies with corresponding working copy values that the thread has modified. After executing a *monitorenter* operation, a thread should either initialize its work copies or assign the master values to them. The only exceptions are variables declared as *volatile*, to which JVM imposes the sequential consistency model. The memory management model is transparent to the programmer and is implemented by the compiler, which automatically generates the code that transfers data values between main memory and the thread local memory.

3 – CoJVM

In previous works [1,7] we introduced the Cooperative Java Virtual Machine (CoJVM), a DSM implementation of the standard Java Virtual Machine (JVM) designed to efficiently execute parallel Java programs in clusters. Most importantly, CoJVM required no change to the syntax and the semantics of the Java language, allowing a programmer to write a concurrent program in the same way as he/she would do for a single JVM.

Java offers to the programmer highly convenient means to develop concurrent programs, since the language itself provides a native parallel programming model that includes support for multithreading. As we described previously, the Java Virtual Machine Specification defines a common memory area, the heap, which is shared among all threads that the program creates. To treat race conditions during concurrent accesses to the shared memory, Java offers a set of synchronization primitives: *synchronized*, *wait*, *notify* and *notifyAll*, which are based on an adaptation of the classic monitor model.

CoJVM implements the illusion of a single multiprocessing system as an interface between Java's multithreaded programming model and the cluster computing infrastructure (figure 2). The heap area is allocated in the DSM space, allowing threads running in distinct cluster nodes to share data. The synchronization primitives are implemented by CoJVM in a distributed fashion: a global monitor is created at the first time a monitor enter operation is applied to an object. Each global monitor keeps a counter and a waiting queue, which contains the threads that are blocked waiting for a determined object to be released. If the counter is equal to zero, the thread immediately blocks the object and increments the counter, otherwise the thread is put in the waiting queue. Whenever an object is released, the thread decreases the counter. Once the counter becomes zero, the first blocked thread is moved from the waiting queue to the ready queue.

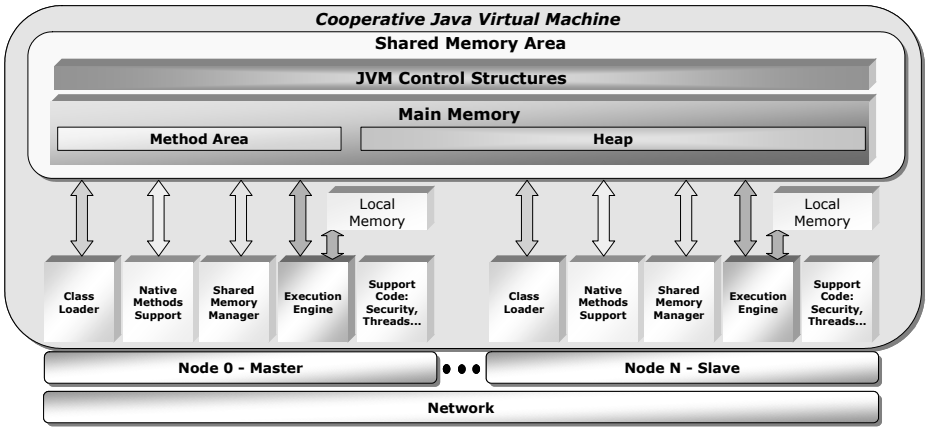

Fig. 2. Sketch of the CoJVM internal architecture.

A thread being executed will be blocked either if it attempts to execute a monitor enter on a blocked object or if it invokes the wait method of an object. A blocked thread is inserted in the ready queue either when the monitor it requests is released or another thread invokes the *notify* (or *notifyAll*) primitive on the object that the blocked thread was waiting for.

CoJVM also implements a mechanism for remote signaling between threads, allowing any JVM to redirect signals (including *wait*, *notify* and *notifyAll*) to another JVM. Remote signaling and global monitors are used to implement transparently the support required to provide concurrency and synchronization to the application threads.

The programmer with the use of Java's synchronization primitives can build a barrier or other synchronization constructs, or invoke a native routine. Nevertheless, since the barrier primitive is usually used by parallel applications, we decided to provide support for it in CoJVM. Besides simplifying the development of applications, that decision allowed us to offer an optimized implementation of the barrier abstraction. However, its use is optional: a Java application that implements its own barrier code can run without modifications in our environment, as well.

The current CoJVM version was aimed at scientific applications. For this reason, just one application thread is supposed to execute in each processing node. Thus the number of threads created by an application must be the same as the number of nodes available for running it. A configuration file is used to inform CoJVM the nodes that will be available for computation. CoJVM is started in one machine, called the master machine, in exactly the same way a Java application is started. Then, the master machine downloads the configuration file and creates one slave machine in each node that is listed in the configuration file. After being initialized in a certain node, a slave machine will keep waiting for a request the master machine will send to it. Whenever an application runs the code to create a thread, CoJVM automatically intercepts that instruction and asks a remote node to perform it. In a similar way, when the application executes a code that starts a new thread, CoJVM redirects that request to the node where the thread was created. When the application finishes, the master sends a message to every slave to finish its execution.

Our past performance analysis using several benchmarks showed that CoJVM achieved good speedups, ranging from 5.4 to 7.7 on 8 nodes. However, we verified that we could further modify CoJVM to improve application speedups. In particular, we noticed that imbalance on shared data distribution by CoJVM could impact negatively barrier times, the number of page access faults, and the *diff* creation task - and consequently the extra amount of control and data messages that CoJVM transfers unnecessarily.

4 – Optimization Techniques

This section presents selective dynamic *diffing* and lazy home allocation, two new techniques we devised to reduce overheads of the coherence protocol we used to implement memory sharing in CoJVM. Selective dynamic *diffing* monitors accesses to the shared memory so that it is able to dynamically create *diffs* selectively, i.e., the *diffs* only include those memory words that changed their values within a certain synchronization interval. Lazy home allocation delays the assignment of pages to home nodes until the application effectively starts its computation phase. Next, we describe the implementation of both techniques in CoJVM.

4.1 – Selective Dynamic *Diffing*

At the *bytecode* level, Java distinguishes shared memory accesses from local ones. In particular, the getfield and putfield *bytecodes* are used to read and write data, respectively, that are exclusively located in shared memory. A simple version of our dynamic *diffing* technique (DDT) uses information on memory access to monitor all

shared memory accesses over the execution time of an application, so that DDT enables CoJVM to avoid the use of the standard *diff* creation mechanism that most software DSM systems adopt. To do so, DDT represents the shared memory region as positions in a special bit vector and ensures that every time DDT detects a write access to the shared memory region, it will set the corresponding bit in the bit vector. DDT will generate *diffs* simply by inspecting the bit vector for marked bits while checking the contents of their associated shared-memory positions. In this way, DDT can reduce the extra memory space that typical software DSM protocols require for twinning. However, DDT cannot eliminate all the twining operations since CoJVM's internal shared data structures are kept coherent through traditional twinning and *diffing* techniques, otherwise CoJVM's write accesses to internal shared data structures should also be instrumented, which clearly would hurt CoJVM performance.

Nevertheless, the simple dynamic *diffing* technique will perform worse than the traditional *diff* creation technique as used by software DSM systems, in the following two cases: a) the elapsed time to mark *n* positions in the bit vector for one page is greater than the elapsed time to generate the *diff* for the same page; or b) an application rewrites the same value to a memory position. In this case, the traditional *diff* creation mechanism will not create a *diff* for that memory position since its value was unchanged. In contrast, the simple DDT will generate the *diff* for that memory position and thus will increase the size of the resulting *diff* for the associate page (Figure 3).

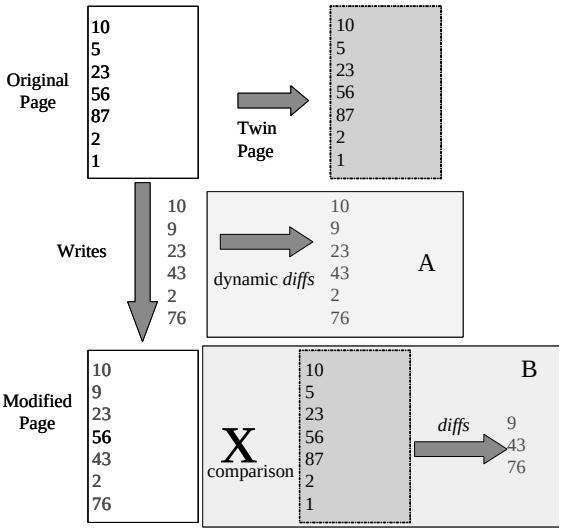

Fig. 3. Traditional *diffing* versus simple dynamic *diffing*. Box A (in yellow) shows that simple dynamic *diffing* operates while writes to the shared memory are being performed. Box B (in blue) shows that *diffs* are traditionally generated by comparison of the original page (the page's twin) with the modified page.

The selective dynamic *diffing* technique extended the simple DDT to take into account the new valued being written to memory position, in that, if a new value is equal to the actual value then the bit vector is not set and also the value is not written to memory, as Figure 4 shows.

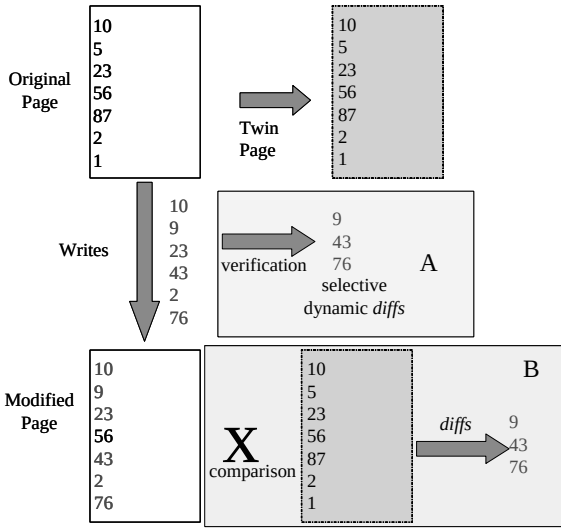

Fig. 4. Selective dynamic *diffing* assures that the *diffs* correspond to modified memory values.

Selective dynamic *diffing* can also reduce the amount of data the coherence protocol transfers. The reason is that CoJVM uses the HLRC, a well-known software DSM protocol to keep the heap consistent. However, some information stored in the heap are private to the local threads and thus are not really shared. By further distinguishing those private thread areas CoJVM avoids the HLRC protocol to unnecessarily transfer the associate data across the network.

4.2 – Lazy home allocation

HLRC adopts a first-touch policy, where a node that first touches a data unit becomes its home node. However, we notice that it is often difficult or even impossible to the programmer to enforce at the beginning of the execution that each node touches the memory pages it will use during computation. The CoJVM implementation is a concrete example. For instance, the Java Virtual Machine Specification defines that all program variables must be initialized with the values the program specifies before they are first used. Usually the initialization is carried out by the node where the computation starts (the master node). This means that the master node will become the home node for the pages it initializes. As we demonstrated in previous work [1], this is likely to lead to a great imbalance in the distribution of home nodes, which ultimately will impact negatively on the amount of data transfers across the network due to increasing substantially extra node requests for remote pages.

We propose the lazy home allocation technique to tackle the imbalance problem caused by HLRC's first-touch policy. In lazy home allocation the association between pages and homes is postponed until the time at which the computation phase of an application really takes place and pages will be definitely used. A careless analysis may suggest that pages be pre-initialized with the initial values (zeros) as defined by the language specification, so that the initialization phase will be avoided, and the

page where data reside would not be associated with the node. The problem is that the initialization process of an object is not restricted to set its initial field values only. Other control data structures the virtual machine uses and which are associated with an object must also be initialized, so pages associated with the residence node of the object will be touched, as well. Lazy home allocation avoids that situation, since the association will be made only when the execution of an application effectively starts. In the case of a Java parallel application, we chose the right moment at which the application invokes the first run method of a thread it created.

5 – Experimental Methodology

5.1 – Hardware Platform

We present experimental results we obtained with a 8-node Linux (2.2.14-5.0) cluster of 650 MHz Pentium III PCs, each of which with 512 MB RAM and 256 KB L2 cache. Each node was connected to a Giganet cLAN 5300 switch and used the VIA communication protocol [8]. We measured Giganet switch's point-to-point bandwidth at 101 MB/s to send 32 KB messages, 7.9 μs latency for sending 1-byte message, and 1.25 Gbps for aggregate bandwidth throughput.

5.2 - Selected concurrent Java Benchmarks

We collected performance results of CoJVM systems for five parallel Java kernel benchmarks: Matrix Multiply (MM) Java kernel, which we developed; SOR, which we ported from Treadmarks's C benchmark suite [9]; LU, FFT, and Radix, which were ported from SPLASH-2 C benchmark suite [10]. The selected concurrent Java kernels offer distinct communication/computation ratios that make them fairly representative of regular and irregular applications that are found in typical scientific applications.

Table 1. Program size, serial execution time in secs, and number of memory 4KB pages that the Java benchmarks required at runtime.

Concurrent Java benchmark	Program Size	Serial Time - CoJVM	No. Pages
MM	1000x1000	491.94	3,230
LU	2048x2048	2,259.10	8,583
Radix	8 Million Keys	25.80	16,723
FFT	4 M Cmplx Dbls	225.94	49,524
SOR	2500x2500	306.37	5,305

MM represents an application that achieves linear speedup in clusters due to its embarrassingly parallel nature, which in turn allowed us to measure communication and synchronization overheads that CoJVM adds to maintain coherent the internal distributed JVM states.

LU, FFT and SOR are concurrent Java kernels to factorize a dense matrix, calculate the Fast Fourier Transform, and solving partial differential equations using the

red-black successive over-relaxation method, respectively. LU, FFT, and SOR represent regular applications with medium or coarse grain access to shared memory but with different sharing patterns that cause fragmentation. In contrast, Radix represents an irregular application that combines fine grain access to shared memory with high synchronization rates. Usually, Radix's performance scales poorly in software DSM systems.

The sequential execution time for each of the five benchmarks is shown in Table 1. Note that the execution times excluded the initialization time to create remote threads or initialize array elements. We submitted the benchmarks to CoJVM five times, and reported the average execution times we obtained for each benchmarks in table 1. The standard deviation was less than 0.1% for each benchmark.

All the kernels but Radix presented sequential execution times greater than 3 minutes. For Radix, the processing node's memory size (512 MB) limited Radix's largest input size to 8-Million keys and execution time to 25.80 seconds, as shown in table 1. Overall, the input data sizes we used can be considered reasonably adequate for our performance analysis of CoJVM systems.

Table 2. CoJVM versions.

CoJVM Version	Dynamic Diffing	Selective Dynamic Diffing	Lazy Home Allocation
Basic	No	No	No
Simple	Yes	No	No
Selective	Yes	Yes	No
Lazy	No	No	Yes
Simple+Lazy	Yes	No	Yes
Selective+Lazy	Yes	Yes	Yes

To quantify the contribution of each run-time technique in isolation and in combination, we submitted the benchmarks to execute in six different versions of the CoJVM system: a) *Basic* version that uses none of run-time techniques; b) *Simple* dynamic *diffing* version that creates and marks the bit vector with all the write accesses the application makes to shared memory without verifying whether memory values have changed or not; c) *Selective* dynamic *diffing* version that creates the bit vector, but only marks write accesses to shared memory that actually changed the memory values; d) *Lazy* version that allocates homes to nodes in a lazy fashion; e) *Simple+Lazy* version that combines the *Simple* with *Lazy* versions; and f) *Selective+Lazy* version that combines the *Selective* with *Lazy* versions. Table 2 summarizes the six CoJVM versions we evaluated.

To understand the experimental results better, we broke down execution time into six distinct components: computation, page, lock, barrier, overhead, and handler time. **Computation** indicates the time spent in useful computation. **Page** indicates the amount of time spent in fetching remote pages from home nodes on page misses. **Lock** is the time spent in acquiring a lock from its current owner. **Barrier** is the time spent at a barrier, waiting for messages from other nodes. **Overhead** time is the time spent performing protocol actions. **Handler** time is the time spent inside the remote handler, a separate thread that is used to service requests from remote nodes. In addition, we measured the amount of messages and data traffic due to memory coherence actions that were generated by both application and run-time CoJVM version.

5.3 – Performance

In Table 3, we present speedups for each of the CoJVM versions. For each benchmark, the table shows speedup of the *Basic* CoJVM version, the best speedup we obtained, which CoJVM version that produced it, and resulting performance gain in comparison with the *Basic* speedup.

Table 3 shows that our run-time techniques were very effective in improving application performance for all the applications, which resulted in speedups between 6.5 and 8.1, except for the embarrassingly parallel MM kernel. Note that the performance gains were even more expressive if we consider that the *Basic* version speedups were already respectable, ranging from 5.4 to 6.9 for those four benchmarks.

Table 3. Speedup figures: *Basic* version, best speedup we measured, best CoJVM version, and resulting performance gain.

Benchmark	*Basic* Speedup	Best Speedup	Best Version	Performance Gain
MM	7,7	7,6	*Lazy*	-1%
LU	6,9	8,1	*Selective+Lazy*	17%
Radix	6,1	6,7	*Selective+Lazy*	9%
FFT	5,4	6,5	*Lazy*	20%
SOR	6,7	7,8	*Lazy*	16%

As shown in figure 5, MM benefited little from using our techniques. The reason is that it spent over 97% of the time performing local computations (Fig. 6). In this situation, the addition of any extra code has direct impact on computation time, as shown in figure 6. Indeed, despite of the benefits accrued to MM from using lazy home allocation, such as reducing substantially the number of control messages and data traffic (Fig. 7), the *Lazy* version's performance became 1% lower than that of the *Basic* version (Fig. 5).

Figure 8 shows LU speedups for the CoJVM versions. As can be seen in figure 8, both *Lazy and Selective+Lazy* versions reduced 17% LU's execution time, whereas the *Selective* version decreased LU's execution time by 6%. Lazy home allocation and selective dynamic *diffing* were responsible for the dramatic reductions in page and barrier times as shown in figure 9. These time reductions had distinct causes depending on the particular CoJVM version it used. The *Lazy* and *Selective+Lazy* versions obtained effective page distribution among the 8 nodes (Fig. 10), which eliminated most of remote page access and *diff* transfer overheads (Fig. 9). As a result, applications spent less time in barrier waiting for *diff*s and bins (data structures that store identifiers of pages that were modified in a certain synchronization interval) transfers. For the *Selective* version, *diff*s were generated only for memory regions that were both shared and modified within a certain time interval, thus reducing network traffic (Fig. 11) and decreasing barrier waiting time.

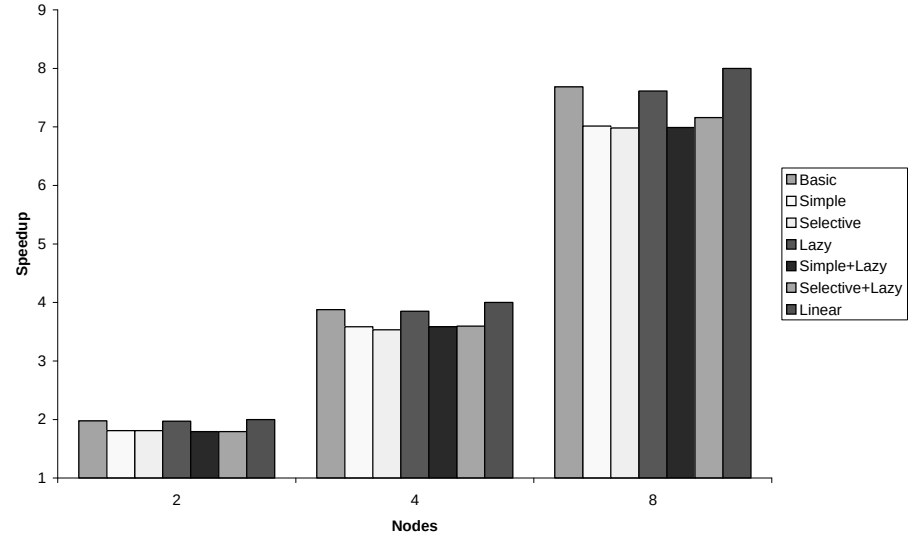

Fig. 5. MM speedups on 2, 4 and 8 nodes.

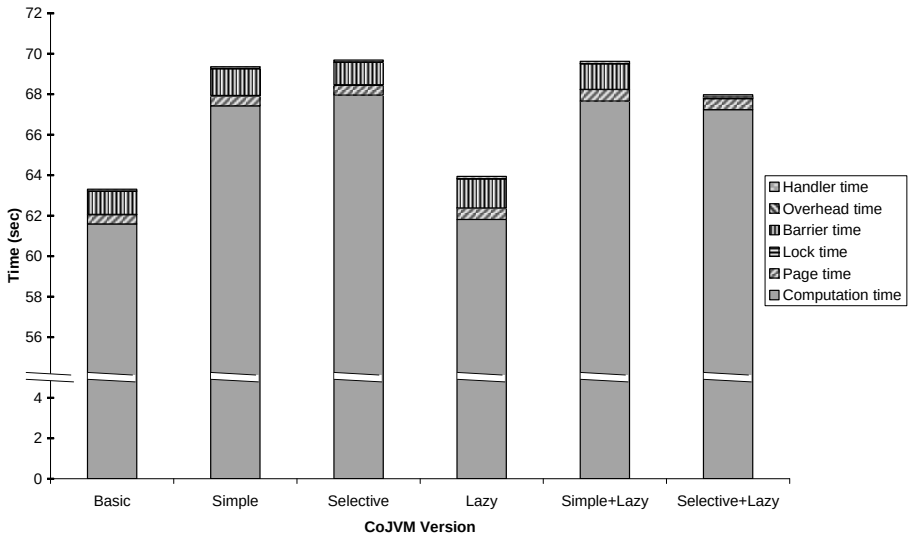

Fig. 6. Execution time breakdown for MM on 8 nodes.

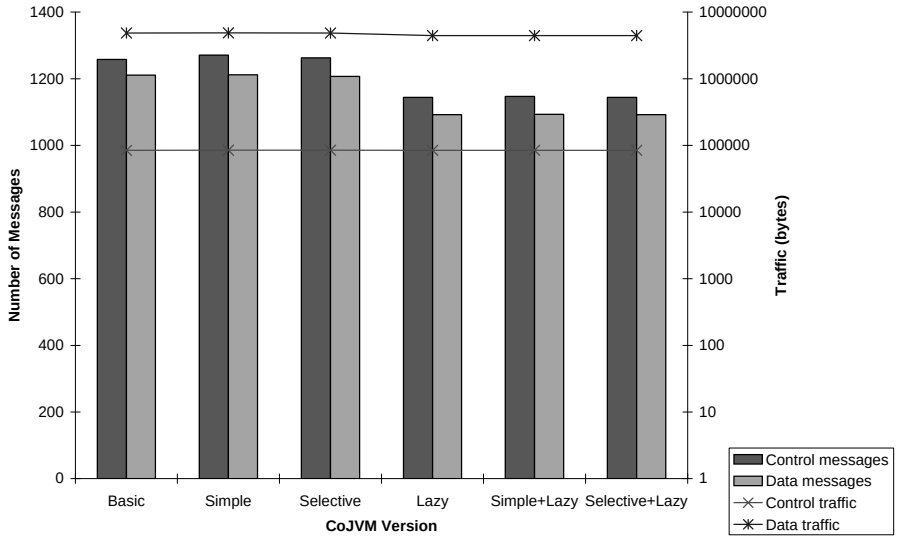

Fig. 7. Amount of messages, data and control traffic for MM on 8 nodes (right Y-axe is logarithmic).

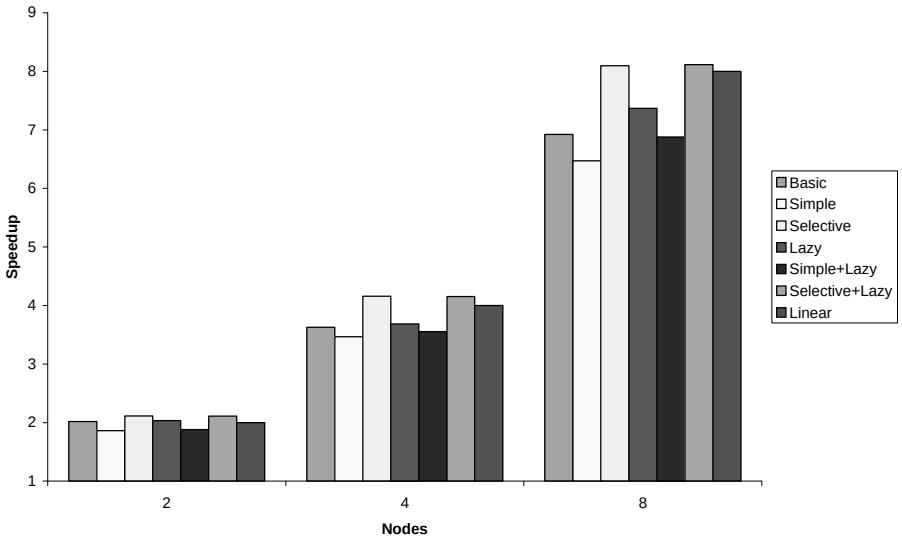

Fig. 8. LU speedups on 2, 4 and 8 nodes.

Figure 12 presents performance results for Radix under each CoJVM version. Specifically, Radix speedup improvements were 3%, 7%, and 9% for *Selective*, *Lazy*, and *Selective+Lazy*, respectively. Again, both run-time techniques contributed to reduce

page and barrier times (Fig. 13), the amount of messages and led to a small reduction in network traffic (Fig. 14).

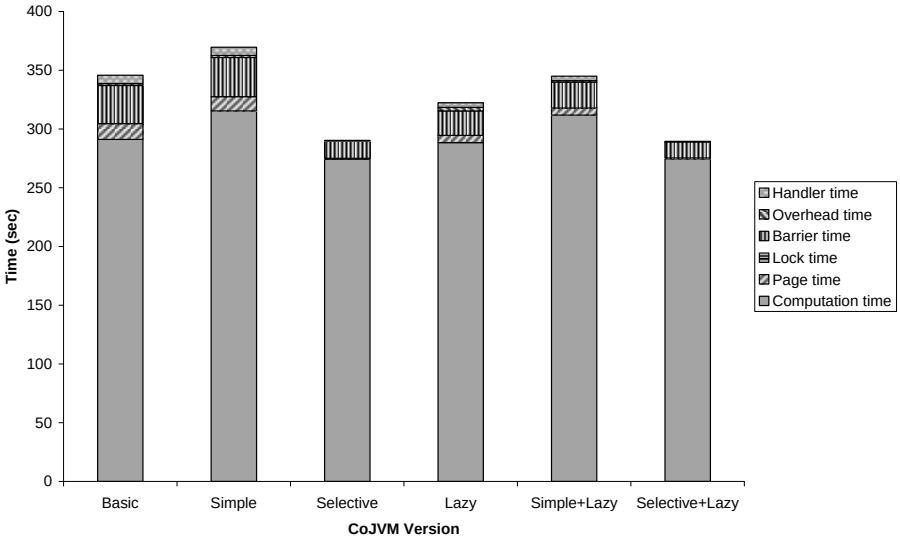

Fig. 9. Execution time breakdown for LU on 8 nodes.

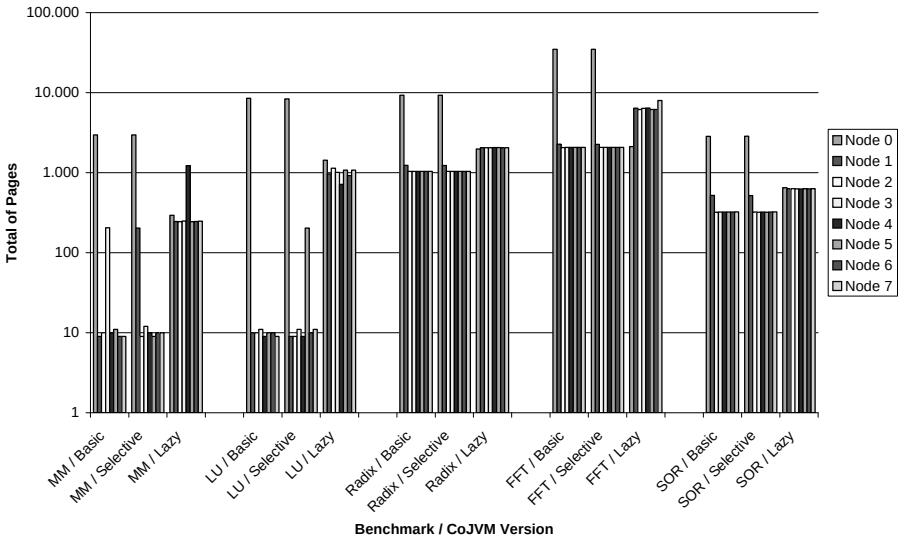

Fig. 10. Page distribution for applications in 8 nodes using three versions: *Basic*, *Selective*, and *Lazy*. The Y-axe is logarithmic.

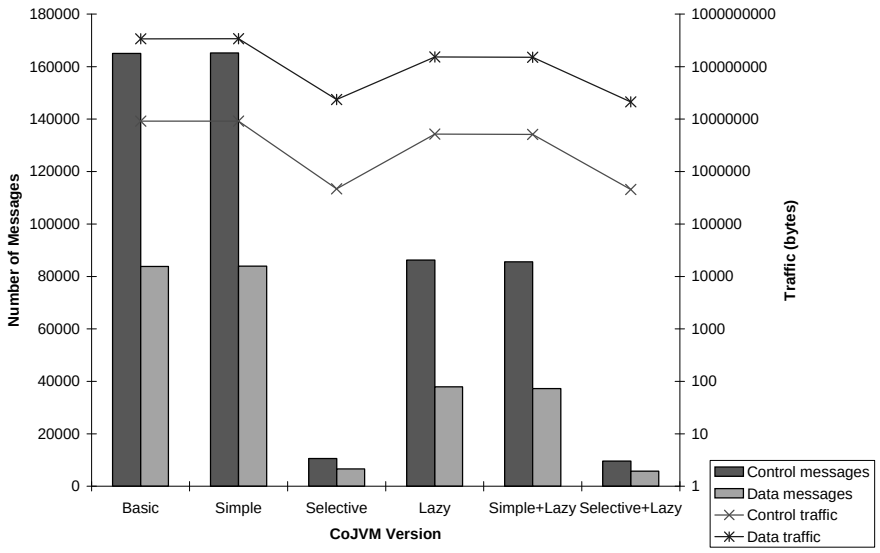

Fig. 11. Amount of messages, data and control traffic for LU on 8 nodes (right Y-axe is logarithmic).

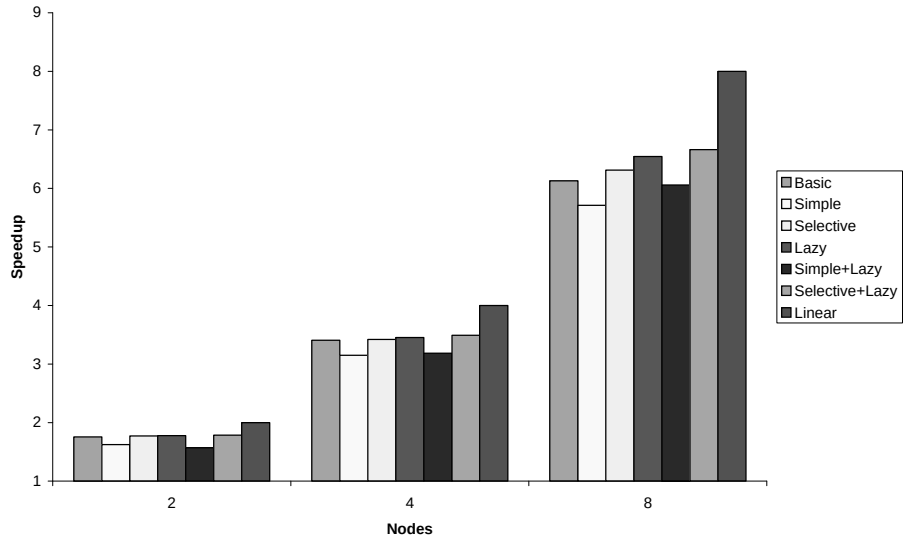

Fig. 12. Radix speedups on 2, 4 and 8 nodes.

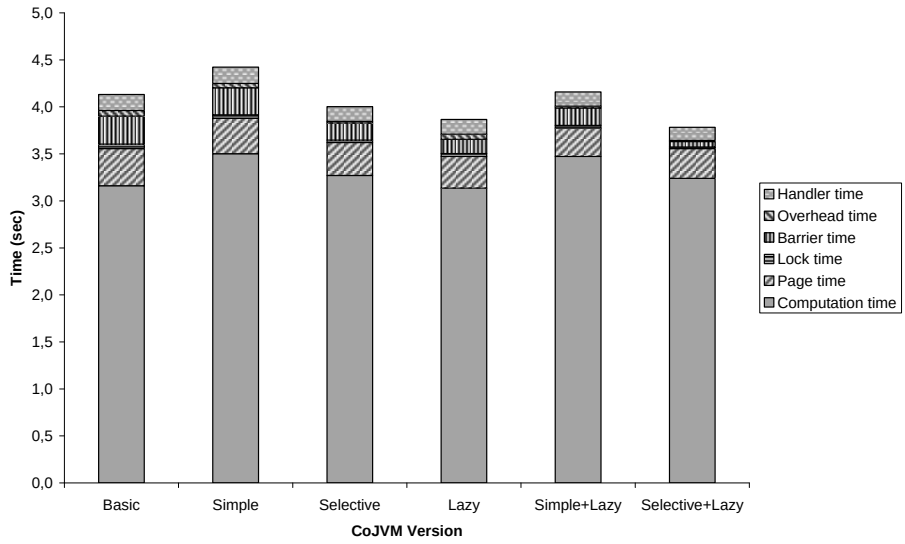

Fig. 13. Execution time breakdown for Radix on 8 nodes.

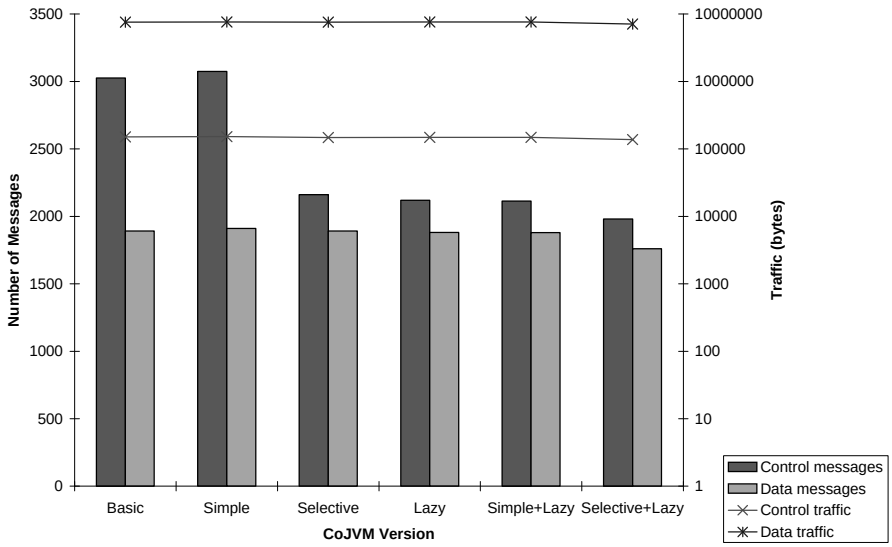

Fig. 14. Amount of messages, data and control traffic for Radix on 8 nodes (right Y-axe is logarithmic).

We observed that the *Selective* version eliminated up to 95% of *diff* creation overheads in Radix, which surprisingly resulted in no benefit to *Selective* version performance. This was explained by the fact that even reducing overheads of *diff* creation the *Selective* version generated as much network traffic as the *Basic* version, which ulti-

mately offset the gains due to *diff* elimination in the *Selective* version. Specifically, the *Selective* version requested 1,852 remote pages and generated 7.25 MB of data traffic, while the *Basic* version produced 1,856 remote page requests and 7.23 MB of data traffic.

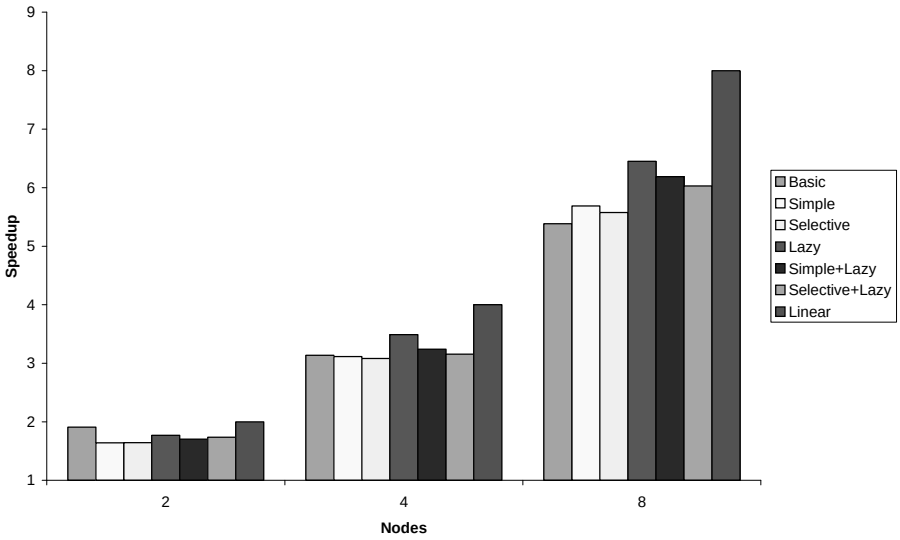

Fig. 15. FFT speedups on 2, 4 and 8 nodes.

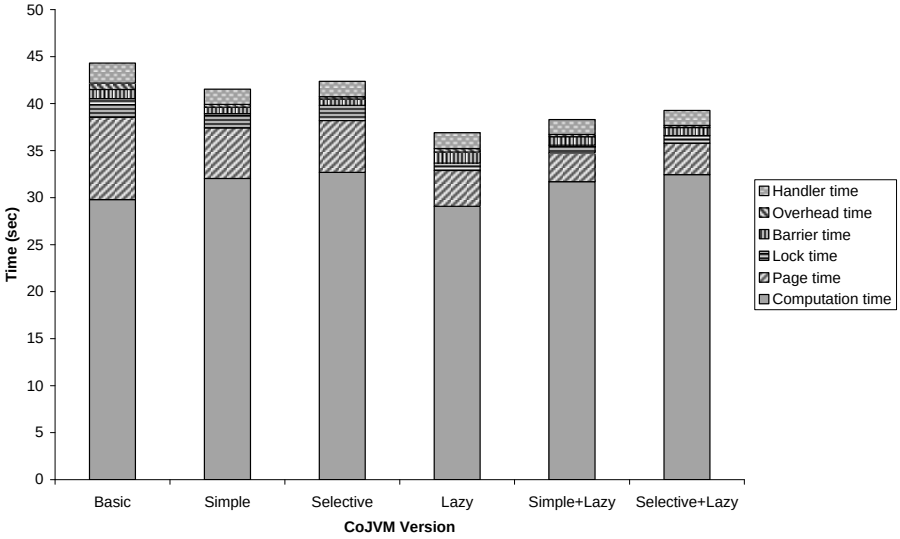

Fig. 16. Execution time breakdown for FFT on 8 nodes.

Single and combined run-time techniques were most beneficial to FFT as shown in Figure 15. All new CoJVM versions performed better than the *Basic* version. The *Lazy* version was the best one, as it performed as much 20% faster than the *Basic* version. This result is explained by effective page distribution among nodes (Fig. 10) and precise identification of data structures that were effectively shared by the application, which contributed to reduce considerably both remote page overheads (Fig. 16), but without a equivalent decrease in the message and the communication traffic, which reduced slightly (Fig. 17).

Table 4. Total number of twin pages that each application pre-allocated (including those used by JVM), when running on 2 nodes, and memory savings (% and in MB). 1 K = 1,024 pages.

Benchmark	*Basic*	*Selective*	Memory Savings
MM	7 K	3 K	43% - 16 MB
LU	34 K	9 K	26% - 100 MB
Radix	25 K	5 K	20% - 80 MB
FFT	101 K	17 K	17% - 336 MB
SOR	9 K	2 K	22% - 28 MB

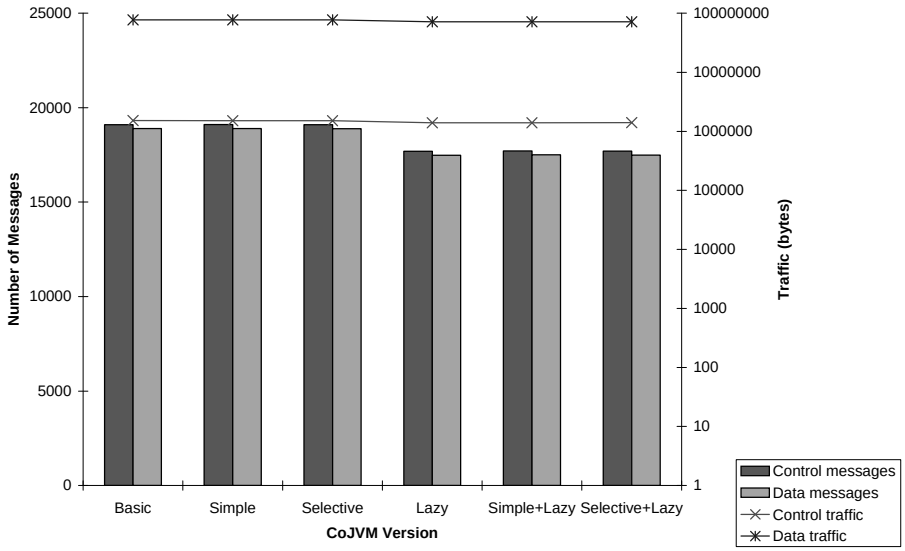

Fig. 17. Amount of messages, data and control traffic for FFT on 8 nodes (right Y-axe is logarithmic).

FFT was the show case for using the dynamic *diffing* technique. FFT was the application that demanded the largest memory size (50,000 pages of 4KB) to execute in the *Basic* CoJVM. However, the use of dynamic *diffing* in the new CoJVM versions reduced greatly FFT's memory requirements as shown in Table 4. Most importantly, the main effect of large memory savings in FFT was to eliminate disc swap operations, which explains the superior performance of all CoJVM versions that use the dynamic *diffing* technique. Actually, FFT was the only application where the *Simple*

version performed better than the *Basic* version. Recall that memory savings occurred due to elimination of twin pages creation.

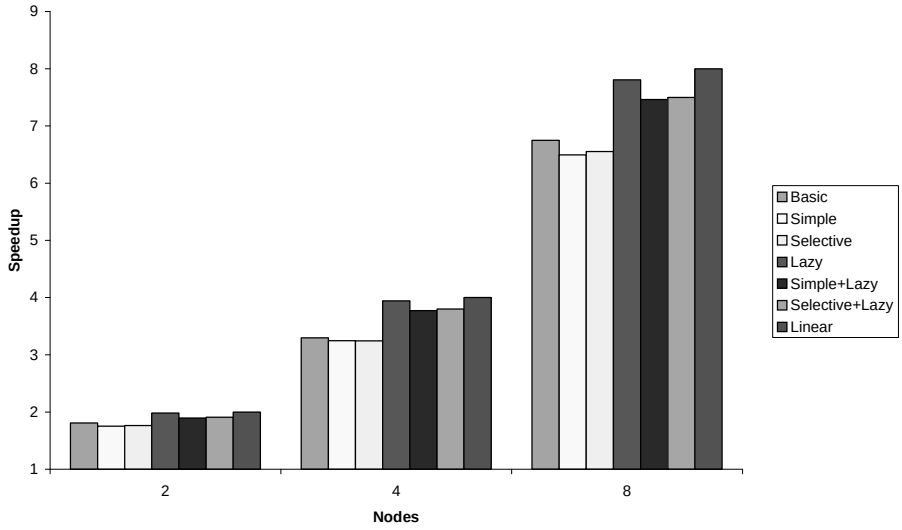

Fig. 18. SOR speedups on 2, 4 and 8 nodes.

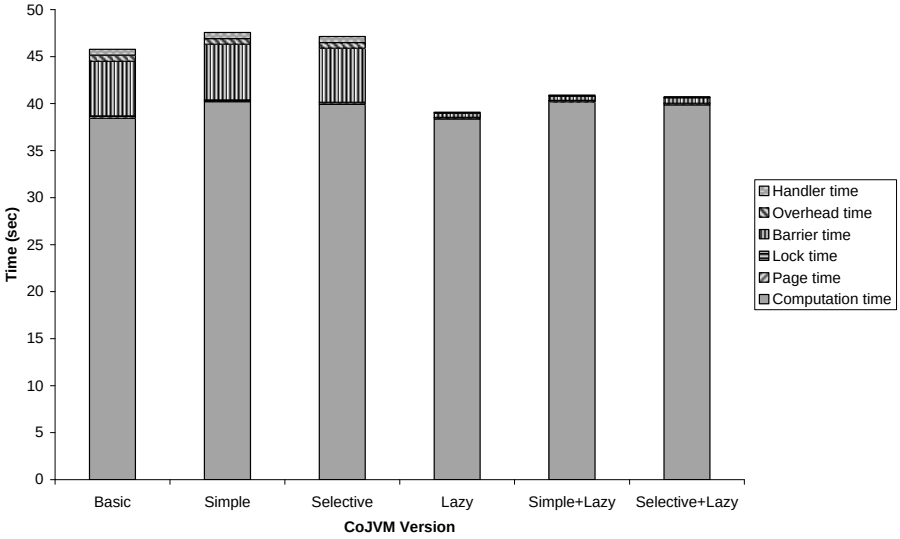

Fig. 19. Execution time breakdown for SOR on 8 nodes.

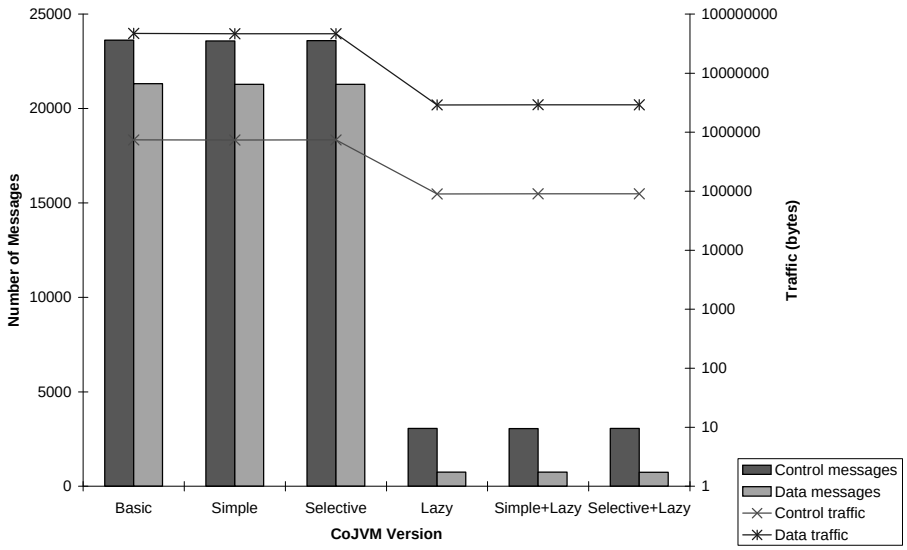

Fig. 20. Amount of messages, data and control traffic for LU on 8 nodes (right Y-axe is logarithmic).

In figure 18, we present speedups that each CoJVM version produced for the SOR benchmark. All *Lazy*-based versions outperformed the *Basic* version, due to better page distribution among nodes. Table 5 shows that the Lazy-based versions obtained an expressive decrease in the amount of *diffs* they created. This reduction explains their lower overheads in barrier time (Fig. 19) and less communication traffic among nodes (Fig. 20).

Table 5. Page access overheads for CoJVM versions on 8-node cluster.

Benchmark	Overhead	*Basic*	*Simple*	*Selective*	*Lazy*	*Lazy+ Simple*	*Lazy + Selective*
MM	Page Fault	1,233	1,232	1,229	1,263	1,263	1,263
	Page Miss	1,081	1,082	1,082	1,087	1,087	1,087
	Diff created	136	133	128	6	7	6
LU	Page Fault	172,237	172,370	7,234	114,564	113,307	6,895
	Page Miss	81,995	82,110	4,853	36,850	36,216	4,674
	Diff created	79,366	79,447	2,086	45,774	45,740	1,301
Radix	Page Fault	3,832	3,854	2,930	3,033	3,031	2,011
	Page Miss	1,852	1,855	1,846	1,857	1,856	1,734
	Diff created	948	966	50	32	31	28
FFT	Page Fault	21,917	21,917	23,512	18,762	18,765	20.142
	Page Miss	15,266	15,268	15,267	13,542	13,520	13,530
	Diff created	3,640	3,640	3,633	3,939	3,985	3,963
SOR	Page Fault	21,755	21,722	21,724	1,441	1,448	1,446
	Page Miss	727	706	708	696	706	705
	Diff created	20,727	20,715	20,715	182	169	169

It is important to notice that the total amount of computation time was reduced in both SOR and LU when submitted to the *Selective* version. The reason was that every time a memory location had to be rewritten with the same value, selective dynamic *diffing* did not set the bit vector and also eliminated unnecessary access to memory.

6 – Related Works

Distributed Java systems based on clusters face major performance problems due to the overheads of costly remote memory access through a high-latency network. A detailed survey focused on Java for high performance computing can be found in [11].

cJVM [12] supports the idea of single system image (SSI) using the proxy design pattern [13], which contrasts to our approach that adopts a software DSM protocol. In cJVM a new object is always created in the node where the request for the object was executed first. Every object has one master copy that is located in the node where the object was created; objects from other nodes will use a proxy to access objects that were created remotely. If an object is heavily accessed, the node where the master copy is located is likely to become a bottleneck. The optimization techniques adopted by cJVM[14] to tackle the problem of heavily accessed objects are: i) object migration and ii) method shipping. Method shipping allows local execution of a method, without accessing the master copy. However, it can not be used for native methods, synchronized methods, and methods that access data in the heap.

JESSICA[15] adopts a home-based, object-based, invalidation-based, multiple-writer memory consistency protocol for a distributed JVM. In JESSICA, the node that started the application, which is called the console node, performs all the synchronization operations, which impose severe performance degradation: for the SOR kernel, synchronization is responsible for 68% of the execution time, and the application achieved a speedup of only 3.4 on 8-node cluster[15]. In CoJVM the lock ownership is distributed equally among all the computing nodes, which contributed to CoJVM better performance. JESSICA developers implemented several performance optimizations [16], including a home migration method to reallocate the home of an object; migration of synchronized methods to allow remote execution of a synchronized method at its home node; and an object pushing optimization in an attempt to improve access locality using the object connectivity information available at run-time.

Hyperion [17] uses the PM2 distributed shared-memory library to implement the DSM model into Java. In contrast to CoJVM, Hyperion, translates Java *bytecodes* into C code, which is used to generate native code. Also, Hyperion does not extract any information from the source code nor the run-time system that could help to optimize the execution of an application, as CoJVM does.

Jackal [18] implements a software DSM abstraction on a cluster by using a special run-time support. In contrast to CoJVM, Jackal uses a compiler to generate native code and to make optimizations, such as data *prefetching* and access check removing, in a similar way to Shasta [19]. In Jackal, every time a thread reaches a synchronization point it invalidates its own data to ensure memory coherence for subsequent accesses. However, this means that it can invalidate data that will not be touched by any other node and thus adds unnecessary overhead to the memory coherence mechanism.

In contrast, the CoJVM coherence protocol only invalidates the local copies of data that will be used in the synchronized method.

DOSA [20] is a fine-grained, object-based distributed shared-memory system. Although it is not a Java environment, its key idea could be applied to Java as well. Specifically, DOSA supported both fine-grained and coarse-grained access patterns provided that pointers and data could be distinguished at run-time. To this end, DOSA introduced an indirection table to locate the address of an object, at the cost of adding an extra memory access, and also used three distinct virtual memory mapping mechanisms, which affected negatively virtual memory usage.

Although current CoJVM version does not reduce the granularity of access, this could be achieved through the use of the bit vector structure to replace the time-expensive virtual-memory protection mechanism for detecting memory writes, as initially planned [7]. However, we noticed that although the time of 40 nanoseconds to mark a single bit in the bit vector was negligible, the huge number of accesses to shared memory may offset the potential gains of avoiding the VM protection mechanism unless we resort to compiler assistance such as in SHASTA.

A dynamic *diffing* technique for distributed shared memory systems has been previously proposed by Pinto *et alli* [21]. Their approach used a non-intrusive hardware support to snoop the memory bus and record all shared data writes in a bit vector structure, although without verifying whether writes modify or not the memory contents. Our work differs in two aspects: a) CoJVM does not use hardware support, but the knowledge it extracted from the JVM by intercepting writes at run-time, and b) CoJVM used the bit vector in a selective way, since their bits are only marked if the corresponding memory contents change at least once. However, that kind of hardware support could be adopted to further reduce the costs of the huge amount of shared memory operations in CoJVM.

Whately *et alli* [22] developed a home-based adaptive DSM protocol (HAP) that categorized memory sharing patterns in order to select a suitable strategy that could optimize the system performance. The strategies they used were home migration, dynamic adaptation between single- and multiple-writer protocols and dynamic adaptation between invalidation and updated based protocols. Differently from HAP, CoJVM does not migrate pages; it just postpones the association of pages with home nodes. The strategies HAP proposed were orthogonal to the techniques we presented in this work and thus they could be integrated with CoJVM.

7 – Conclusions and Future Work

In this work, we described two new optimization techniques namely selective dynamic *diffing* and lazy home allocation that used the information the Cooperative Java Virtual Machine extracts at run-time about the application behavior.

Five representative kernels, MM, SOR, LU, FFT, and Radix, were used as benchmarks to evaluate the impact of our proposed techniques on CoJVM performance. Our experimental results show that four out five benchmarks achieved superior speedups ranging from 9% to 20% when they were executed in optimized CoJVM versions, in comparison with original CoJVM version without using our techniques. The only exception was MM, which performed 1% better under original CoJVM than under the optimized versions. Overall, our techniques contributed to reduce overheads

related to memory usage, number of messages, the amount of data transferred, and even computation time of CoJVM.

Although the techniques were tested in an environment that provided run-time support to collect information about application behavior, we believe that our techniques could be also used in software DSM system that do not provide such run-time support. In this case, a similar mechanism for code instrumentation could be developed to identify both write accesses to shared memory and starting point of actual computation phase of an application.

Current CoJVM implementation is based on Sun's JDK 1.2. We plan to migrate our environment to the newest version of the Sun's released JVM, and to investigate how the techniques we proposed would perform on DSM environments that adopt languages that do not provide run-time information support.

References

1 Lobosco, M, et alli. A New Distributed JVM for Cluster Computing. 9th International Euro-Par Conference, pp. 1207-1215, Aug. 2003.
2 Arnold, K; Gosling, J. The Java Programming Language. AddisonWesley, 1996.
3 Li, K, Hudak, P. Memory Coherence in Shared Virtual Memory Systems. ACM Transactions on Computer Systems, 7(4):321-359, Nov 1989.
4 Keleher, P, Cox, A, Zwaenepoel, W. Lazy Release Consistency for Software Distributed Shared Memory. Int. Symp. on Computer Architecture, pp. 13-21, May 1992.
5 Zhou, Y, et alli. Performance Evaluation of Two Homebased Lazy Release Consistency Protocols for Shared Virtual Memory Systems. OSDI, Oct 1996.
6 Lindholm, T, Yellin, F. The Java Virtual Machine Specification. Addison-Wesley, 1999.
7 Lobosco, M, Amorim, C, Loques, O. A Java Environment for High-Performance Computing. 13th Symp. on Computer Architecture and High-Performance Computing, pp. 180-186, Sep 2001.
8 VIA. *VIA Specification, Version 1.0*. http://www.viarch.org. Accessed on Jan, 29.
9 Keleher, P. et. alli, Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems. *Proceedings of the Winter 1994 USENIX Conference*, pp. 115-131, January 1994
10 Woo, S, et alli. The SPLASH-2 Programs: Characterization and Methodological Considerations. Int. Symp. on Computer Architecture, pp. 24-36, Jun 1995.
11 Lobosco, M, Amorim, C, Loques, O. Java for High-Performance Network-Based Computing: a Survey. Conc. and Computation: Practice and Experience: 2002(14), pp 1-31.
12 Aridor, Y, Factor, M, Teperman, A. cJVM: a Single System Image of a JVM on a Cluster. Int. Conf. on Parallel Processing, Sep 1999.
13 Gamma, E, et alli. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
14 Aridor, Y. et alli. A High Performance Cluster JVM Presenting a Pure Single System Image, JavaGrande 2000, pp. 168-177.
15 Ma, M, Wang, C, Lau, F. JESSICA: Java-Enabled Single-System-Image Computing Architecture. Journal of Parallel and Distributed Computing (60), pp. 1194-1222, 2000.
16 Fang, W, et alli. Efficient Global Object Space Support for Distributed JVM on Cluster. Int. Conf. on Parallel Processing, Aug 2002.
17 Hatcher, P. et alli, Cluster computing with Java. IEEE Computing in Science and Engineering, vol. 7, n. 2, pp 34-39, March/April 2005.
18 Veldema, R., et alli. Runtime optimizations for a Java DSM implementation. Conc. and Computation: Practice and Experience, Special Issue: ACM 2001 Java Grande-ISCOPE (JGI2001) Conference,.vol. 15, issue 3-5, pp 299-316, February 2003.

19 Scales, D., Gharachorloo, K., Thekkath, C. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. ACM Conference on Architectural Support for Programming Languages and Operating Systems, pp. 174-185, October 1996.
20 Hu, Y. et alli. Runtime Support for Distributed Sharing in Typed Languages. Lecture Notes in Computer Science: Selected Papers from the 5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers; vol. 1915, pp 192-206, 2000.
21 Pinto, R., Bianchini, R., Amorim, C. Comparing Latency-Tolerance Techniques for Software DSM Systems. IEEE Trans. on Parallel and Distributed Systems, vol. 14, n. 11, pp. 1180-1190, November 2003.
22 Whately, L. et alli. Adaptive Techniques for Home-based Software DSM. 13th Symp. on Computer Architecture and High-Performance Computing, pp. 164-171, Sep 2001.