Proceedings of the 16th IASTED International Conference
PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS
November 9-11, 2004, Cambridge, MA, USA

Jupiter/SVM: A JVM-based Single System Image for Clusters of Workstations

Carlos D. Cavanna∗, Tarek S. Abdelrahman†, Angelos Bilas‡, and Peter Jamieson†

†Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada M5S 3G4

∗Compiler Development Group
IBM Canada Ltd.
Markham, Ontario
Canada L6G 1C7

‡Institute of Computer Science (ICS)
Foundation of Research and Technology – Hellas (FORTH)
P.O. Box 1385
Heraklion, GR 71110, Greece

Abstract

We address the problem of providing a single system image (SSI) on clusters of workstations, based on the Java Virtual Machine (JVM). Our approach is unique in that the needed functionality is separated in two layers: a shared virtual memory (SVM) system, CableS, that is optimized for system area networks and provides a standard Pthreads API, and a multithreaded JVM, Jupiter, that was originally developed for symmetric multiprocessors (SMPs). We identify the JVM extensions that are required to deal with CableS's more relaxed memory consistency model, to optimize memory allocation by using private memory where possible, and to deal with various dynamic resource limitations imposed by CableS. We present a preliminary evaluation of the new JVM using the Java Grande benchmark suite on a 16-processor cluster of PCs interconnected with a Myrinet network, which (to the best of our knowledge) is the largest configuration reported to the literature. We find that: (i) the overhead introduced by SVM-specific extensions is less than 7% on average and (ii) Jupiter/SVM scales well to achieve an average speedup of 14 on 16 processors—a significantly better speedup than for previous reported work. Our main contribution is the conclusion that JVM-based SSIs for clusters do not have to be based on specially designed JVMs but may use JVMs that have been developed for popular SMP platforms.

Keywords: Clusters of Workstation, Java Virtual Machine (JVM), Single System Image (SSI), Virtual Shared Memory (SVM), Scalable Performance.

1. Introduction

Virtual machines that provide higher-level abstractions of system resources are becoming important as a tool for separating applications from system characteristics. This is particularly the case for clusters of workstations that are interconnected with low-latency, high-bandwidth system area networks. These clusters are gaining acceptance as a platform for supporting applications that require access to scalable resources. Yet, the abstractions provided by clusters to applications tend to be restricted and low-level, hindering the use of clusters in new applications areas.

We address this problem by providing a single system image (SSI) on clusters of workstation, based on the Java Virtual Machine (JVM). Java has steadily gained wide acceptance as a programming language of choice for application development, mainly because of its platform-independence and because of the rich set of classes and libraries that it provides. Today, there exist several large server applications that are centered around and/or written in Java, including IBM's WebSphere [1], a scalable high-performance transaction engine for e-business, JigSaw [2], an advanced web server platform, and Jetty [3], a Java HTTP server and servlet container. It is natural to explore the use of clusters to deliver the resources and performance needed by such large applications.

In this paper, we describe our approach to providing a JVM-based SSI on clusters. The approach is unique in that the necessary SSI functionality is provided in two layers. The bottom layer is the CableS Shared Virtual Memory (SVM) system [4]. It provides a release-consistency-based shared address space abstraction and a standard Pthreads API for applications. This layer essentially provides the illusion of an SMP system that supports Pthreads and a relaxed memory consistency model. The top layer is Jupiter, an SMP-based multithreaded JVM [5]. We identify the extensions that are required to support SVM clusters. These include support for dealing with the release consistency model provided by CableS, optimizing memory allocation by using private memory where possible, and dealing with various dynamic resource limitations imposed by SVM systems.

Our approach has several benefits. First, it enables on modern clusters the use of JVMs that have been developed for SMPs. Thus, JVM-based SSIs for clusters do not have to be based on custom JVMs. It identifies and supports system modularity through a clear division of system functionality in two distinct layers. Finally, it demonstrates that a standard Pthreads API [6] is adequate to separate the two layers, facilitating their independent evolution. Given the complexity of such systems and in particular the fact that JVMs require significant effort to develop and optimize, our work provides a path for leveraging this effort on an important architectural platform. In fact, the JVM we use, after our moderate extensions, supports both SMPs and clusters without increasing code complexity.

We perform a preliminary evaluation of Jupiter/SVM, using the Java Grande benchmark suite on a 16-processor cluster of PCs interconnected with a Myrinet network. Page-

‡Also with the Department of Computer Science, University of Crete, P.O. Box 2208, Heraklion, GR 71409, Greece.

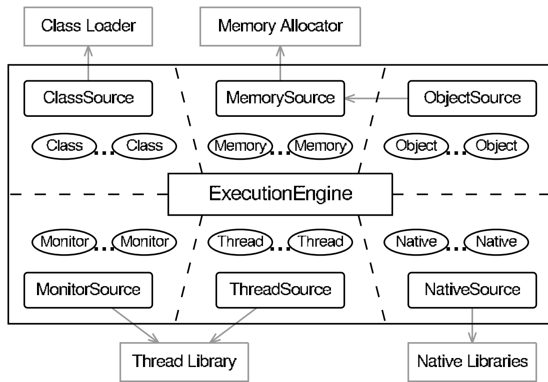

Figure 1. The overall conceptual structure of Jupiter.

based SVM systems can introduce significant overhead because of (i) false sharing, (ii) the potential mismatch between the memory model of Java and SVM, and (iii) modularity, which prevents Java run-time information from being propagated to, and exploited by, the SVM system. However, our preliminary evaluation indicates that Jupiter/SVM scales well, achieving an average speedup of 14 on 16 processors. The underlying SVM system provides support for local caching and replication and alleviates the need to implement complex changes to the JVM. Our experimental evaluation indicates that the overhead introduced by SVM-specific extensions is less than 7% when run on a single SMP without the SVM system, indicating that the same JVM can be used both on SMPs and clusters with minimal performance impact.

The remainder of this paper is organized as follows. Section 2. presents an overview of Jupiter's architecture and our shared virtual memory cluster. Section 3. describes the changes to Jupiter that were necessary to deal with the release consistency model, private memory allocation, and limitations of resources. Section 4. presents the results of our experimental evaluation of Jupiter/SVM. Section 5. describes related work. Finally, Section 6. provides concluding remarks and directions for future work.

2. Background

2.1 The Jupiter JVM

Jupiter [5, 7] is a modular and extensible JVM infrastructure that targets Symmetric Multiprocessors (SMPs) with hardware support for sequential consistency. It is a working JVM that provides the basic facilities required to execute multi-threaded Java programs. It has an interpreter and gives Java programs access to the Java standard class libraries via a customized version of the GNU Classpath library [8]. It is also capable of invoking native code through the Java Native Interface [9] and it provides memory allocation and collection using the Boehm garbage collector [10]. It currently has no bytecode verifier, no JIT compiler, though the design allows for such extensions. The performance of Jupiter's interpreter is midway between that of Kaffe and that of Sun's JDK [5].

The overall conceptual structure of Jupiter is depicted in Figure 1. In the center is the `ExecutionEngine`, the control center of the JVM. It decodes bytecode instructions and determines how to manipulate resources to implement those instructions. The resources, shown as ovals, include Java classes, fields, methods, attributes, objects, monitors, threads, stacks and stack frames (not all of which are shown in the diagram). The responsibility for managing each resource is delegated by the `ExecutionEngine` to a particular *Source* class, each shown as a rectangle within the pie slice that surrounds the resource it manages.

2.2 The CableS SVM system

Our SVM system consists of two software layers that collaborate to provide applications with a single system image with respect to memory, thread management, and synchronization. These layers are shown in Figure 2.

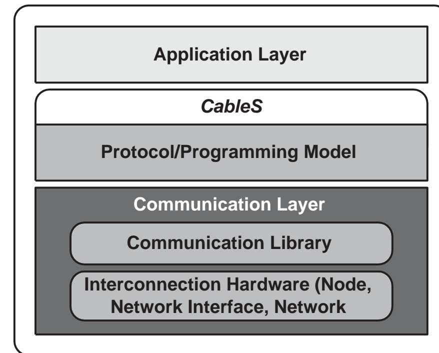

Figure 2. The CableS SVM system Layers.

The lowest layer is the communications layer, and it is called Virtual Memory Mapped Communication (VMMC) [11]. It consists of a custom control program for the Myrinet Network Interface Cards (NIC), an OS driver, and a user library. VMMC allows a user to directly access the NIC in a protected manner without having to trap into the kernel and also to write memory through the network without interrupting the remote processor.

The next layer up is CableS (Cluster enABLEd threadS) [4], a page-level shared virtual memory (SVM) system. CableS relies on GeNIMA [12], which provides a single address space abstraction on top of VMMC and implements a Lazy Release Consistency (LRC) memory model [13, 14].

In LRC (as in other relaxed memory models), accesses to shared data must be protected with lock acquire and lock release operations. For the most part, this is also true under the Sequential Consistency (SC) model [15] commonly implemented in hardware on SMPs. However, under SC it is possible in a small number of cases, such as flag-based synchronization, to not use locks. In the LRC model *all* accesses to shared data must be protected with locks. This not only ensures race-free execution [16], but also guarantees that local copies of shared memory are properly updated.

GeNIMA uses a software, home-based multiple writer scheme to reduce the impact of false sharing. Each page is assigned a home node that collects page updates. When a processor first writes to a page during a new interval it saves a copy of the page, called a *twin*, before writing to it. At

a lock release operation the releasing processor compares the current (dirty) copy of the page with the (clean) twin to detect modifications and records these modifications in a structure called a *diff*. On an acquire operation, the requesting processor invalidates all pages by consulting the information about updated pages received in conjunction with the acquired lock. The next access to an invalidated page causes a page fault, and the page fault handler fetches the correct page version from the home.

CableS provides resource management for memory, threads, and synchronization through a POSIX threads (Pthreads) [6] standard API. More specifically, CableS extends GeNIMA in three respects. First, CableS allows applications to dynamically create and destroy threads during execution. Second, it provides support for dynamically allocating shared memory and for memory placement (first-touch in the current implementation). Finally, CableS supports modern synchronization primitives including the Pthread conditional-wait primitive, mutex/lock primitives, and barriers.

3. JVM extension for SVM clusters

A JVM that supports an SSI on clusters must provide three broad sets of functionalities in accordance with the Java Memory Model (JMM): (i) thread management, including thread creation, migration, and termination; (ii) thread synchronization, including locks and condition variables; and (iii) a shared object space that is accessible to all threads. We provide these functionalities through existing standard interfaces and, where possible, existing system layers. Thus, we divide the required functionalities in two layers: CableS, that provides a single system image with respect to memory and thread management as well as synchronization support, and Jupiter/SVM, that we modify to deal with the relaxed memory model and the limitations imposed by CableS.

This approach eliminates the need for extensive modifications to the SMP-based JVM. In addition, the underlying SVM system provides support for local caching and replication of data at the page level. This alleviates the need to implement mechanisms for object sharing and consistency within the already complex JVM. Moreover, the SVM system provides support for shared memory allocation and thread management, which alleviates the need for heap address translation and explicit thread management. Finally, the SVM system manages important resources, such as locks, which allows for efficient thread synchronization to be performed in a de-centralized fashion. All this allows us to use a JVM that is internally very similar to an SMP-based JVM. In fact, after our extensions Jupiter/SVM supports both SMPs as well as clusters of workstations.

Nonetheless, our SMP-based Jupiter relies on sequential consistency and must be extended to support the more relaxed consistency model of our cluster. Furthermore, additional modifications and optimizations are necessary for building a practical system. Overall, the related issues can be classified in three categories: (1) *memory consistency*, which arises due to the use of the cluster's lazy release memory consistency model; (2) *private memory allocation* for those Jupiter structures that are used exclusively by a single thread; and (3) *limitation of resources*, which arise because of various limitations imposed by SVM systems.We examine each of these issues in details.

3.1 Memory consistency

The lazy release consistency (LRC) model [13] of the cluster dictates that all accesses to shared data must be protected by lock acquire and lock release operations. Since Jupiter is already multithreaded, the majority of accesses to shared data are already protected by such operations. However, Jupiter was originally developed for SMPs that mostly support the sequential consistency (SC) model [15]. This allows Jupiter to avoid the use of explicit synchronization operations in the following cases: System thread creation, Java thread creation, object creation and initialization, and volatile field accesses.

3.1.1 Argument passing at system thread creation

Creating a system thread in Jupiter involves invoking the `pthread_create` call. This call allows for a single value (a pointer) to be passed to the child thread. This is sufficient since the pointer may be used to pass an argument structure that is large enough to hold multiple arguments passed from the parent to the child. Thus, this structure is shared among the parent and child threads. A similar technique is used in many other thread APIs as well.

Under SC this form of sharing does not require any additional synchronization. All writes to objects by the parent thread proceed all reads to the same object by child threads. This precedence is guaranteed by the fact that child threads are created by the parent thread after all writes have been performed.

However, in CableS as well as other SVM systems that support LRC, the child thread will see the parent updates only if the shared data structure is protected by explicit synchronization operations. For this reason, the parent and child threads use a statically-agreed upon lock to protect accesses to the arguments data structure. Note that this lock cannot be part of the structure that is passed to the child because this would create a cyclic dependency between getting the lock and using the lock.

In our implementation, we employ a single system-wide global lock that is used by all threads to protect accesses to parameter structures. A parent thread acquires and releases this lock when writing the arguments to the structure. Each child thread uses the same operations when reading the arguments. The use of a single global lock for thread creation implies that at any time, only one thread can prepare and read arguments. Multiple threads attempting to do so at the same time must serialize, even though their preparations of arguments are completely independent of one another. Although, one can use a pool of initial locks for this purpose, we find that a single lock is adequate. The additional overhead is negligible mainly due to the fact that thread creation already incurs high overhead in SVM systems [4].

3.1.2 Object initialization at Java thread creation

A Java thread can potentially access any class or object that its parent has initialized; when a Java `Thread` object is created, its parent may initialize fields in the child object with

references to any of the fields, objects, or classes of the parent. Thus, the child thread object may contain references to data that was created and/or modified by its parent before the child is created, and this data is shared between the parent and the child. As before, when using sequential consistency, such data is automatically visible to the child thread upon its creation because of the sequential consistency model. However, under CableS and LRC we must ensure that updates to such objects are explicitly `released` by the parent and `acquired` by the child thread.

It is important to note that this issue arises only at thread creation time. The parent and child thread may proceed to concurrently access shared fields. This happens either through synchronized methods or through unprotected access to object fields. In the former case, the corresponding monitor ensures that values of shared data are updated for each thread. In the latter case, there is no guarantee of data updates, but this is consistent with the JVM specification [17].

To guarantee that object modifications are visible by the child thread, the parent thread uses a pair of lock/unlock operations with an empty critical section right before creating the child thread. The child thread uses a similar pair before starting execution. These pairs of operations are performed on the same lock variable. Another approach would be to use `acquire/release` primitives available in the CableS system, however since this issue arises only at thread creation time, which is less performance critical, we use the more portable calls to lock synchronization operations.

3.1.3 Java thread termination

The Java Memory Model [17, 18] states that, when a Java thread terminates, it has to flush all modified variables to main memory. Under sequential consistency, there is no need to take direct action when a thread terminates; memory is updated automatically. However, under CableS and LRC, while it is possible to have CableS automatically force a memory update upon the termination of a system thread, this is not a solution from which all SVM applications could benefit, and thus, was not implemented. For this reason, it is necessary to introduce a mechanism for forcing the memory system to flush the memory contents to the other system nodes. Thus, we extend Jupiter to use an explicit call, `svm_flush_memory`, provided by CableS. This call allows memory updating on demand and is invoked before the system thread terminates and after all Java code has completed its execution in the terminating thread.

3.1.4 Volatile fields

The Java Language Specification [19] states that *volatile* fields must reconcile their contents in memory every time they are accessed. Furthermore, the operations on volatile fields must be performed in exactly the same order the threads request them. In the multithreaded Jupiter, it was not necessary to take any special actions on accesses to single-word volatile fields because of sequential consistency. However, all cluster accesses to such fields have to be protected by explicit lock synchronization operations. When another thread requires access to a volatile field, it is guaranteed that it will see all previous updates.

3.2 Private memory allocation

The use of memory in shared memory clusters requires special attention for two reasons. First, access to remote memory incurs high overhead, regardless of how optimized the base communication system is. For this reason, shared memory protocols try to avoid accessing remote memory, by extensive caching and replication of shared pages. Second, the 32-bit address space of the systems we use (and are also used in most clusters today), imposes a limit of 4 GBytes total address space. Consequently, and unlike SMP systems, there is a need to perform careful allocation of the internal JVM data structures and assign to shared memory only what is absolutely necessary. The rest of the data structures that are essentially private to each thread, can reside in private thread memory throughout execution. Furthermore, depending on the implementation of the underlying SVM system, private memory allocation may result in performance benefits as well. The Jupiter modules and data structures that can benefit from such private memory allocation are:

- *The thread stack*. In Jupiter, a thread has an independent opcode interpreter, with its own stack, which stores the state of Java non-native method invocations. Thus, the stack always remains private to a thread, and can be allocated in private memory. Note that the stack itself may contain references to shared objects, but in itself is private.

- *The native stack and cache*. When the operating system creates a new native thread, it provides a stack that the thread uses for native method invocation. This stack is also private to the thread and can be allocated in private memory. Jupiter allows the call to native methods to use this same stack. Furthermore, all native methods that are accessed from a Java thread are resolved and kept in a cache of method bodies, indexed by method name. Since calls to native methods are local, the cache is also local to the thread and can be allocated in private memory.

- *Thread handler*. In Jupiter, threads are managed in the same way as other resources; they have a corresponding Source class, `ThreadSource`. In the current design, a single instance of `ThreadSource` is created for each thread in the system, which remains private to each of them, and may be allocated in private memory. Also, in Jupiter, each thread only needs to be able to identify its children and wait for their completion. A supporting data structure used for this purpose is private to each thread, and therefore, can be stored in private memory as well.

- *The bytecode interpreter*. Each thread contains a separate bytecode interpreter (`ExecutionEngine`). This object has a single instance for each thread, which is completely independent of others and thus, makes it possible to allocate it in private memory.

- *The class parser*. No information regarding the parsing of a class is shared among threads. Thus, the data structures used by the parser can be allocated in private memory. It is important to note that while these structures are private, Jupiter's internal representation of the

class must be stored in shared memory, since this representation is shared.

We extend Jupiter's memory allocation infrastructure to allow the allocation of private and shared memory. `MemorySource` is the current module responsible for memory allocation. We replace it with a version that provides two types of memory allocation objects: (i) `malloc`-based for private memory and CableS `svm-malloc`-based for globally shared memory. The `malloc`-based objects are then used in each of the above cases to allocate the corresponding objects in private memory.

3.3 Limitation of resources

CableS and most SVM systems impose various limitations on the use of system resources. There are two reasons for these limitations: First, there are certain restrictions imposed by the underlying platforms, which are not important for smaller systems, but become important in modern clusters. Second, SVM systems use resources internally for their operation. The most important limitations are:

First, the available virtual address space. The x86-based processors used extensively today provide a 32-bit virtual address space. Moreover, Linux limits this address space to 2 GBytes by reserving portions of the address space for internal use[1]. Thus, the total shared address space cannot exceed this maximum. Furthermore, CableS [4] requires that shared pages be mapped twice in the virtual address space for managing transparently update propagation of modified data. Thus, the final address space available to the JVM is about 1 GByte. Similar limitations are imposed by other efficient SVM systems as well. Although, this issue will be resolved with 64-bit processor architectures and 64-bit extensions to existing processor architectures, today it remains an important limitation. Our technique of optimizing allocation of internal JVM structures by using private memory makes it possible to mitigate the impact of this limitation on user programs.

Second, in many cases, to reduce synchronization overheads modern clusters provide support for lock operations in the communication subsystem. For instance, VMMC provides simple support for locks that is used by GeNIMA and CableS. This results in a limit in the number of locks that user programs can use, mainly based on the amount of memory that modern network interfaces incorporate. Currently, this limit in our system is a few thousand locks and condition variables. Although this is sufficient for most cases, it may result in false sharing of locks.

A more important limitation of modern SVM systems is that they perform garbage collection of internal data structures at global synchronization points (barriers) only. Each synchronization point in an SVM system defines a new interval for which various metadata is maintained. For instance, the pages that have been updated during an interval are recorded internally. These data structures are garbage collected at barriers, which is sufficient for scientific type workloads that use global synchronization extensively. However, Pthreads only provide point-to-point synchronization primitives, which imposes a significant limitation. Currently, node memory is adequate for a few hundred thousand intervals between barrier invocations.

[1]This can be extended to 3 GBytes with appropriate Linux kernel reconfiguration.

	2-way cluster node	4-way SMP
CPUs	2 x PentiumII 400MHz	4 x Xeon 1.4GHz HT disabled
L1 cache	16KB Icache 16KB Dcache	12KB Trace cache 8KB Dcache
L2 cache	512 KB	512 KB
Front bus speed	100MHz	400MHz
Memory size	512 MB	2 GBytes
Linux kernel	2.2.16-3smp	2.4.18.-4smp
Ethernet	100 MBit	100 MBit
Myrinet	M2M-PCI64A-2	N/A
Link speed	2 x 1.28GBits/s	
NIC processor	Lanai7 66MHz	
Board memory	2MB	

Table 1. Configuration of our platforms.

We deal with this issue by (i) optimizing internal lock use and (ii) by introducing a native `barrier` primitive that does not rely on high-level, point-to-point synchronization provided by Pthreads but rather uses directly a more efficient barrier primitives provided by CableS. It is important to note that both of these optimizations are necessary for performance reasons as well. However, it is true that they impose limitations on JVM implementations that do not necessarily target performance. Finally, we should note that SVM protocol extensions to perform garbage collection at point-to-point synchronization would be an important addition to support certain classes of applications and thus, constitute an interesting direction for future work.

4. Experimental evaluation

In this section, we present the results of our evaluation of the performance of Jupiter/SVM. We first describe the platforms, benchmarks, and methodology we use for the evaluation, then we present 3 sets of results. The first set demonstrates the scalability of Jupiter/SVM up to 16 processors. The second set indicates that the overhead of the extensions made to Jupiter is not significant. Finally, the third set quantifies the overhead of using software-based shared memory.

4.1 Platforms and methodology

We use two platforms in our evaluation: a 16-processor cluster and a 4-processor SMP. The cluster consists of 8 dual processor PCs (each referred to as a node), for a total of 16 processors. The configurations of the two platforms are summarized in Table 1. The software components described earlier in Section 2.2 are used to provide shared virtual memory and thread support; the Linux kernel is modified to accommodate these components. The Jupiter and Jupiter/SVM JVMs are each compiled on both platforms with GNU gcc 2.95.3 20010315 (release) at `-O3` and linked to ClassPath Version `0.03`. All source code modules of each JVM are combined into a single compilation unit to facilitate function inlining [20].

We use the Java Grande multithreaded applications benchmark suite [21, 22]. We are also able to run all the SPECjvm98 benchmarks. However, they are single-threaded applications, with the exception of `raytrace` and `mtrt`, which just calls `raytrace`. Since a program with a similar function is part of Java Grande (`RayTracer`), we opt not to report on the performance of `raytrace`.

The Java Grande benchmarks provide two standard input data sets, small-sized (*A*) and mid-sized (*B*). Some include a large-sized input set (*C*), but we restrict our evaluation to sizes *A* and *B*. The number of threads used for each application is always the same to the number of processors on the test platform. For the most part, the benchmarks execute unmodified on Jupiter/SVM. However, in some cases it was necessary to make minor modifications:

- `LUFact`, `SOR`, `MolDyn` and `RayTracer` use application-level barriers that are written in Java and are inefficient. We replace calls to these barriers with calls to the more efficient native cluster barriers.

- `Crypt`, `SOR`, `SparseMM` and `MonteCarlo` use the standard ClassPath random number generator to generate random values before parallel execution. This generator uses a `synchronized` method, and thus, locks, which are a limited resource on our system. To reduce the use of locks, we replace the random number generator with a compatible version with no `synchronized` methods.

- Since CableS does not yet support sharing of internal operating system handles the I/O operations in `MonteCarlo` are replaced with a class that stores the necessary values.

Furthermore, we do not report the performance of `crypt` because it creates many threads that are short-lived. These threads do not run in parallel in a coordinated fashion and given the high overhead of thread creation on our cluster platform, threads finish executing before all threads are created. Thus, they are allocated to a small number of processors, making the speedup an inappropriate metric for system scalability. Although other metrics, such as system throughput could be used instead, this is beyond the scope of our work. `MonteCarlo` and `MolDyn` exceed the dynamic resource limitations imposed by CableS in synchronization intervals for input size *B*. Thus, we do not report on their performance for this data size on the cluster.

We measure all execution times using the `time` utility. These times are total execution times and they reflect the times experienced by the user. We also measure the benchmark time for each benchmark, as reported by the benchmarks. This is defined as the time spent in the Java program and includes initialization[2], Java thread creation, parallel section execution, and Java thread termination. Thus, the difference between the execution time and the benchmark time reflects the cost of CableS and JVM initialization. We report the average execution time and benchmark time of each benchmark, which are calculated for each benchmark as the arithmetic mean of 5 independent runs of the benchmark. All times are reported in seconds.

[2] `Series`, `LUFact`, `SOR`, and `SparseMM` exclude data initialization from benchmark time.

4.2 SVM-related instrumentation overhead

We measure the overhead of the extensions made to Jupiter to enable its execution on SVM clusters that support a more relaxed consistency model by comparing the execution times of the benchmarks using Jupiter/SVM running on the 4-processor SMP (i.e., without the CableS SVM) to their execution times using Jupiter in the same platform. Note, that this version of Jupiter/SVM is not linked with the SVM system, however, it includes all modifications discussed in Section 3. We find that the overhead is less than 17% across all applications, with a much lower geometric mean of less than 7%.

Next, we measure the impact of the software-based shared memory system by comparing execution and benchmark times of the benchmarks on one cluster node (i.e., only up to 2 processors) using Jupiter and Jupiter/SVM. This indicates the overhead introduced by the SVM system for initialization and bookkeeping purposes.

Figure 3 shows the normalized execution and the benchmark time of the benchmarks using Jupiter and Jupiter/SVM. The results are normalized to the execution time of Jupiter/SVM running without SVM. The figure indicates that the average penalty of using the SVM as opposed to hardware-based shared memory is 39% for one thread, and 44% for two threads, jumping to more than 70% for some benchmarks. These overheads stem from both the expensive initialization of the SVM, thread creation, which is slow on the cluster, and runtime overhead caused by the use of the SVM during program execution.

4.3 System scalability

We measure scalability using application *speedup*. In our experiments, the single-thread execution includes SVM execution; our intent is to show how the system scales with more processors. We also present a cumulative (total) application speedup, which is defined as the ratio of the sum of the execution (benchmark) times of all the benchmarks at one thread/processor to the sum of the execution (benchmark) times of all the benchmarks at P threads/processors. Thus, the cumulative speedup gives more relative importance to benchmarks within the Java Grande suite that execute longer. The execution and benchmark times of the Java Grande benchmarks using Jupiter/SVM on a one cluster processor is shown in Table 2. The speedups of the individual benchmarks as well as the total speedup for the benchmark suite are depicted in Figure 4.

Figure 4 shows that most of the benchmarks exhibit sublinear speedup for input size *A*. However, for input size *B*, the larger problem sizes result in better computation to communication ratios for most of the benchmarks, and thus, lead to better speedups. The total speedup of the benchmarks for input size *B* is close to 14 at 16 processors, indicating that for sufficiently large data sets, for which it is natural to use a cluster, the system scales well.

5. Related work

There has been a number of projects aiming to build JVMs that provide an SSI on clusters. These cluster-JVMs may be

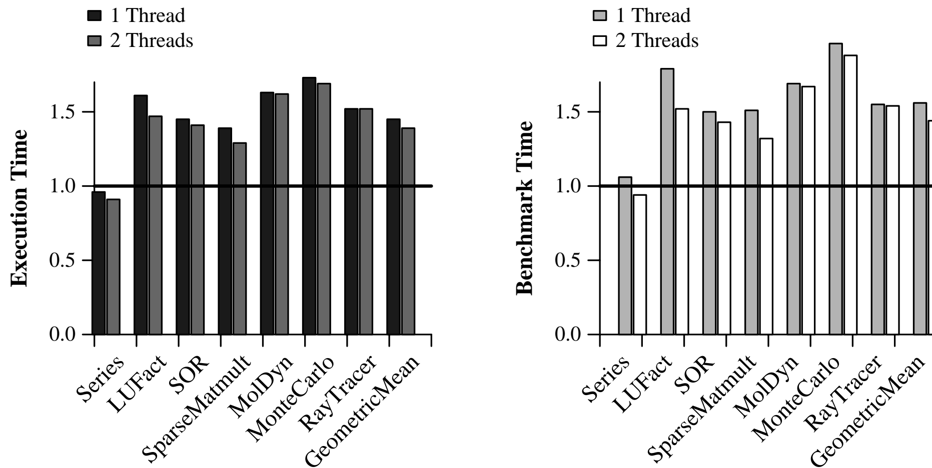

Figure 3. Normalized execution and benchmark time overheads introduced by the SVM system for input size *A* on a single cluster node. Results are normalized to the execution time of Jupiter/SVM running without SVM.

Benchmark	Input size *A*		Input size *B*	
	Exec.	Bench.	Exec.	Bench.
Series	147.75	137.93	1388.30	1378.45
LUFact	111.40	97.01	795.59	768.16
SOR	494.85	469.09	1094.15	1050.84
SparseMM	182.02	158.87	360.93	324.65
MolDyn	1583.76	1573.61	N/A	N/A
MonteCarlo	460.67	448.77	N/A	N/A
RayTracer	2319.42	2308.33	25705.96	25692.78
Total	**5299.87**	**5193.61**	**29344.93**	**29214.88**

Table 2. The average execution and benchmark times (in seconds) at one thread/processor.

broadly classified into three types: *monolithic*, *SVM-based*, or *hybrid* [23].

A monolithic cluster-JVM provides support for thread management and a shared object space internally within the JVM itself. While such an approach may lead to better performance by tailoring various protocols to the Java Memory Model (JMM), it involves extensive re-architecture of the JVM. Examples of monolithic cluster JVMs include cJVM [24] and the Jackal system [25].

A SVM-based cluster-JVM executes on the top of a SVM system which provides thread management support and the illusion of a shared object space to the JVM. This approach simplifies a cluster-JVM design, but is commonly believed to result in inferior performance, because of: (i) the potential mismatch between the JMM and the memory model supported by the SVM, and (ii) the difficulty of optimizing performance due to the fact that the cluster JVM and the SVM are in two separate system layers. Examples of SVM-based cluster-JVMs include Java/DSM [26], Kaffemik [27], and our Jupiter/SVM.

Finally, a hybrid cluster-JVM combines aspects of the first two types of cluster-JVMs. For example, a hybrid JVM may employ a SVM system to provide a shared object space but is internally modified to provide thread and synchronization support. Hybrid cluster-JVMs attempt to reduce the extent of architectural changes to the JVM by relying on standard system layers, but also tailor some of its protocols to the JMM. Examples of hybrid cluster-JVMs include Hyperion [28], JESSICA [29], and JESSICA2 [23].

Our approach in Jupiter/SVM is that of a SVM-based cluster-JVM, but it improves upon similar systems. For example, it incorporates data caching and replication mechanisms that are lacking in Kaffemik. It also alleviates the need for heap address translation which is necessary in Java/DSM. Furthermore, we are able to transparently run a large set of benchmarks and provide concrete demonstration of the viability of our approach (e.g., it is unclear whether Java/DSM was ever developed to completion).

More importantly, we show that with our SVM-based approach it is possible to obtain scalable performance of the cluster JVM on a 16-processor system, which is, to the best of our knowledge, one of the largest cluster configurations reported in the literature. In spite of its current limitations, Jupiter/SVM delivers better performance than what is reported for existing cluster-JVMs, and on a larger set of applications. For example, the speedups of Jupiter/SVM for `RayTracer` and `SOR` on 8 processors exceed those of JESSICA (indeed, they experience a slowdown). Similarly, the speedup of Jupiter/SVM, at 3 processors, for `RayTracer` is significantly better than that of Kaffemik for the same program.

6. Concluding remarks

In this paper, we describe the design, implementation, and evaluation of a JVM-based SSI on clusters. We provide the necessary SSI functionality in two layers: a bottom layer, CableS, that provides a release-consistency-based shared address space abstraction and a standard Pthreads API, and a top layer, Jupiter/SVM, which is an SMP-based JVM extended to deal with memory consistency, private memory allocation, and resource limitation issues imposed by the lower layer.

We evaluate the performance of our system on a 16-processor cluster of PCs that are interconnected by a Myrinet network, as well as on a 4-processor SMP. We

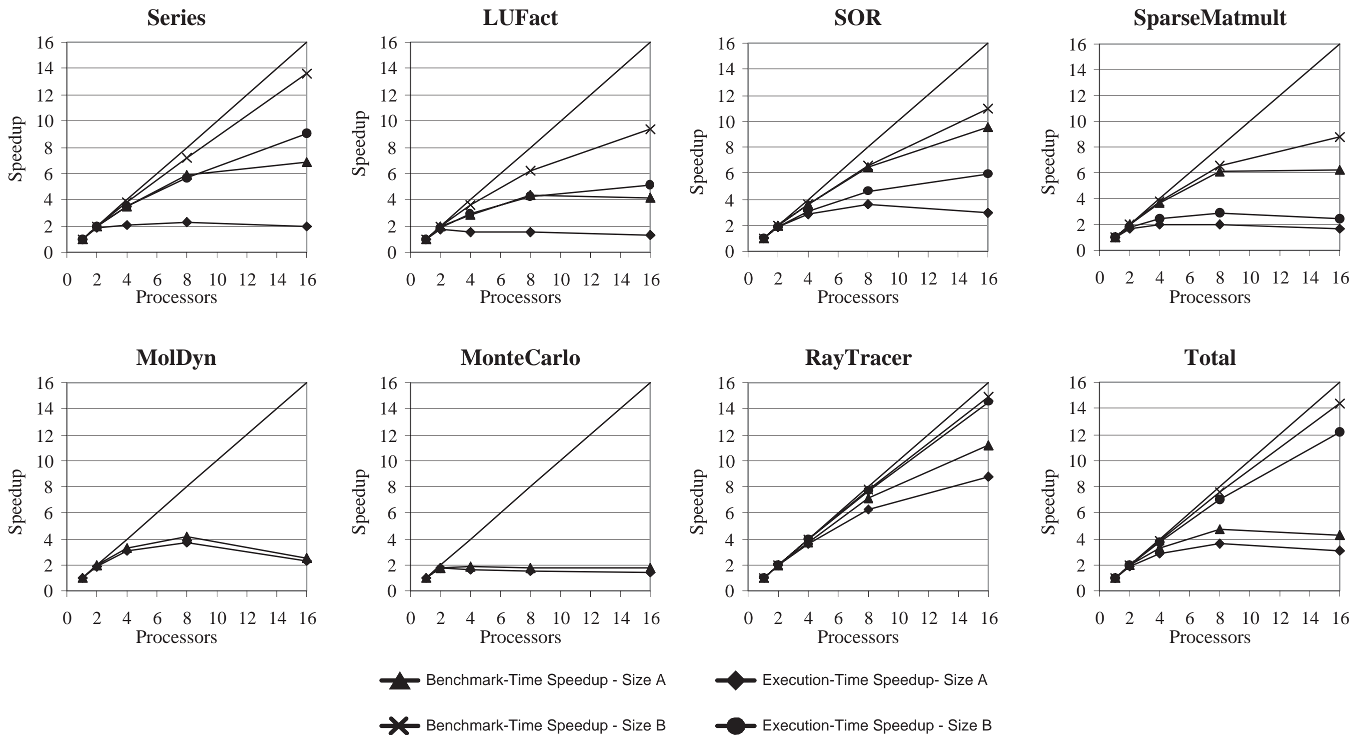

Figure 4. Benchmark speedups. For each application we show speedups based both on execution as well as benchmarks time.

demonstrate that: (i) the overhead introduced by SVM-specific extensions is less than 7% on average and (ii) Jupiter/SVM scales well to achieve an average speedup of 14 on 16 processors. Thus, our approach demonstrates that JVMs designed for SMPs are also able to run on clusters. We identify the required support and prototype a working JVM that supports both SMPs and clusters. Given that JVMs require significant effort to develop and optimize, our work allows leveraging this effort on clusters, an important architectural platform.

Finally, future work will focus on two venues. First, we will address the functionality limitations of our current implementation, which include SSI I/O, garbage collection, and a JIT compiler. Second, more work is required to better understand the overheads associated with providing JVM-based SSIs on clusters and the use of locality enhancement techniques for scaling to larger systems than what we examine.

Acknowledgments

This work was supported by Natural Sciences and Engineering Research Council of Canada (NSERC), by Communications and Information Technology Ontario (CITO), and by the General Secretariat for Research and Technology, Greece (GSRT). The work was carried out when Carlos D. Cavanna was with the University of Toronto.

References

[1] IBM., *IBM Websphere*, 2003. `http://www.ibm.com/websphere`.

[2] World Wide Web Consortium., *Jigsaw: W3C's Server*, 2003. `http://www.w3.org/jigsaw/`.

[3] Mort Bay Consulting., *Jetty: Web Server & Servlet Container*, 2003. `http://jetty.mortbay.com`.

[4] P. A. Jamieson and A. Bilas, "CableS: Thread control and memory management extensions for shared virtual memory clusters," in *Proc. of HPCA*, pp. 263–274, 2002.

[5] P. Doyle and T. S. Abdelrahman, "Jupiter: A modular and extensible JVM infrastructure," in *Proc. of JVM*, pp. 65–78, 2002.

[6] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming. A POSIX Standard for Better Multiprocessing*. O'Reilly & Associates, 1996.

[7] "The Jupiter project," 2002. `http://www.eecg.toronto.edu/jupiter`.

[8] "GNU Classpath," 2003. `http://www.gnu.org/software/classpath/classpath.html`.

[9] "Java Native Interface," 2002. `http://java.sun.com/j2se/1.3/docs/guide/jni/index.html`.

[10] H. Boehm, "A garbage collector for C and C++," 2002. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`.

[11] A. Bilas, C. Liao, and J. Singh, "Using network interface support to avoid asynchronous protocol processing in shared virtual memory systems," in *Proc. of ISCA*, 1999.

[12] A. Bilas, C. Liao, and J. Singh, "Accelerating shared virtual memory using commodity NI support to avoid asynchronous message handling," in *Proc. of ISCA*, 1999.

[13] P. Keleher, A. Cox, and W. Zwaenepoel, "Lazy consistency for software distributed shared memory," in *Proc. of ISCA*, pp. 13–21, 1992.

[14] Y. Zhou, L. Iftode, and K. Li, "Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems," in *Proc. of OSDI*, 1996.

[15] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocessor programs," *IEEE Trans. on Computers*, vol. 28, no. 9, pp. 690–691, 1979.

[16] K. Gharachorloo et al., "Programming for different memory consistency models," *Journal of Parallel and Distributed Computing*, vol. 15, no. 4, 1992.

[17] T. Lindholm and F. Yellin, *The Java^{TM} Virtual Machine Specification, 2nd edition*. Addison-Wesley Publishers Co, 1999.

[18] D. Lea, *Concurrent Programming in Java. Design Principles and Patterns, 2nd edition*, pp. 90–97. Sun Microsystems, 1999.

[19] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java^{TM} Language Specification, 2nd edition*. Addison-Wesley Publishers Co, 2000.

[20] P. Doyle, "Jupiter: a modular and extensible Java Virtual Machine framework," Master's thesis, University of Toronto, 2002.

[21] "Edinburgh Parallel Computing Centre (EPCC)," 2003. `http://www.epcc.ed.ac.uk/`.

[22] "Java Grande Forum," 2003. `http://www.javagrande.org/`.

[23] W. Zhu, C. Wang, and F. C. M. Lau, "Jessica2: A distributed Java virtual machine with transparent thread migration support," in *Proc. of Cluster Computing*, 2002.

[24] Y. Aridor et al., "Transparently obtaining scalability for Java applications on a cluster," *J. of Parallel and Distributed Computing*, vol. 60, no. 10, pp. 1159–1193, 2000.

[25] R. Veldema, R. Bhoedjang, and H. Bal, "Jackal, a compiler based implementation of java for clusters of workstations," in *Proc. of PPoPP*, 2001.

[26] W. Yu and A. L. Cox, "Java/DSM: A platform for heterogeneous computing," *Concurrency - Practice and Experience*, vol. 9, no. 11, pp. 1213–1224, 1997.

[27] J. Andersson, S. Weber, E. Cecchet, C. Jensen, and V. Cahill, "Kaffemik - a distributed JVM on a single address space architecture," 2001.

[28] G. Antoniu et al., "Compiling multithreaded Java bytecode for distributed execution," *LNCS*, vol. 1900, pp. 1039–1052, 2001.

[29] M. J. M. Ma, C. Wang, and F. C. M. Lau, "JESSICA: Java-Enabled Single-System-Image Computing Architecture," *J. of Parallel and Distributed Computing*, vol. 60, no. 10, pp. 1194–1222, 2000.