Distributed Clustering and Scheduling of Object-Oriented Virtual Machines

João Lemos

Instituto Superior Técnico, Avenida Rovisco Pais, 1 - 1049-001 Lisboa, Portugal,
joao.lemos@ist.utl.pt,
WWW home page: http://cms.gsd.inesc-id.pt/Members/jlemos

Abstract. This report presents an overview of several approaches to provide a *Single System Image* view of a cluster, particularly concerning the view of a single address space. The main focus of our work is to understand the current approaches for clustering a regular multithreaded and non-cluster-aware Java application, as well as the current techniques and metrics for scheduling threads in a heterogeneous cluster. We will start by addressing the Distributed Shared Memory (DSM) approach, together with the consistency models studied in the academic world to improve performance and reduce the communication. Software Transactional Memory (STM) will also be addressed as a possible alternative to lock-based approach for providing data synchronization between nodes. In section 3.2 we will take a closer look to current Java approaches for clustering and their main features (e.g. global GC, support for JIT, native code and load-balancing mechanisms). Algorithms for scheduling threads and thread migration will also be considered in 3.3, complemented by a small reference to live migration of virtual machines. Finally, we propose a new extension to provide Terracotta with a global scheduler in an attempt to make it a viable platform for general-purpose multithreaded applications without having the programmer to worry about load-balancing.

1 Introduction

The designation "Virtual Machine" has been around since the 60s, and it was originally used to describe a software implementation that executes programs like the "real" hardware. In those days, hardware-level virtual machines were popular [1], and several VMMs, like IBM's CP-40, were developed at that time. This allowed IBM to run several single-user operating system instead of a multi-user operating system, such as Unix. However, the VMM solution implied higher overheads and difficult design decisions, such as what should be handled by the VMM and what should be handled by the guest OS (e.g. swapping memory to disk). In a system where both the VMM and the OS had mechanisms for page swapping we could end up with conflicts or a suboptimal decision. As a result, multi-user OSs ended up being widely adopted and the concept of "Virtual Machines" was abandoned until the late 1990s, with Sun's Java Virtual Machine

becoming widely used. Nowadays, the term "Virtual Machine" designates a full taxonomy of different virtualizations, which some authors such as Smith et all [2] try to classify. Despite the variations, we can define a virtual machine as a target architecture for a developer or compilation system, that can have or not correspondence to an existing physical hardware.

In recent years, computer clusters made entirely of simple desktop computers are becoming the standard for high-performance computing, as the scalability and cost-efficiency of such solution surpasses most high-end-mainframes. If the workstations in a cluster can work collectively and provide the illusion of being a single workstation with more resources, then we would have what is referred in the literature as a *Single System Image* [3]. Much research has been done in the area of SSIs, providing sophisticated systems that achieve a single view of resources such as process space (OpenSSI) or filesystem (NFS). One of the initially most promising techniques that has been widely used is Distributed Shared Memory (DSM). By extending the traditional virtual memory architecture we can provide a distributed global address space that allows a cluster composed of different machines to be used as a shared memory system. The first software-based DSMs systems such as TreadMarks [4] simply organized the memory into pages of fixed sized that were split across the machines in the cluster, with a *Release Consistency* algorithm to provide proper synchronization. Unfortunately, programming in accordance with the consistency algorithm proved to be a difficult task and the performance was far from excellence. Modern DSM systems such as Terracotta [5] follow an object-based approach and the memory is organized as an abstract space for storing objects of different sizes, offering a transparent location of objects to the applications. Those and many other systems are described in more detail in the Related Work, including the algorithms used for guaranteeing consistency and minimizing communication among nodes.

Considering these facts, and the known popularity of programming languages designed for running with a High Level Language VM such as Java or C#, it is worth studying the possibility of extending a VM with clustering support in a SSI fashion. There are three major approaches in achieving this goal:

- **Extend a programming language at source or bytecode level:** allows a simple and straight-forward implementation fully compatible with current VMs but it does not provide full transparency to the programmer and existing applications need to be modified or recompiled for using a specific library. In either case, the application source might not be available.
- **Design a cluster-aware VM:** gives full transparency to the programmer but it requires the applications to use a specific cluster-aware VM instead of any standard VM, sacrificing the portability of the system.
- **Design a cluster infrastructure capable of running several standard VMs:** gives the best compromise between portability and transparency but it is the hardest one to develop and many implementations are incomplete and do not provide a full SSI.

One of the essential mechanisms necessary for providing SSI systems is the scheduling of threads for load balancing across the cluster. To the best of authors

knowledge, some work has been done in improving the scheduling of threads for page-based DSM systems in order to avoid *Page-Thrashing* and improve the locality of memory accesses but no modern DSM system can provide the full transparency desired for running already existent applications. Terracotta, for example, has no global thread scheduler and the programmer of a multithreaded application needs to be concerned about manually launching multiple instances of the applications, and manual load-balancing. This is the main motivation for developing this work.

A global scheduler can also have thread migration support, as the load in the cluster changes overtime and it becomes necessary to rebalance the load, as many authors defend that otherwise the communication overhead becomes a bottleneck in performance [6]. Also, in a heterogeneous cluster the processors differ in speed and the computational resources available also change during runtime, adding extra complexity for the global scheduler to deal with [7]. Despite these difficulties, recent research in Virtual Machine technology has allowed the concept of *capsule* to appear and entire systems can be migrated within a cluster for user commodity and also for load-balancing. Recent studies by Chen el all. [8] have showed that this approach can be just as efficient as thread migration.

The rest of this paper is organized as follows. Section 2 describes the main goals of this work. Section 3 discusses related work. Section 4 describes the architecture of the prototype to develop. Section 5 describes the evaluation method to measure solution adequacy and performance and section 6 concludes this paper.

2 Objectives

The goal of this project is to develop a prototype of a Java-based SSI system with a global thread scheduler that can provide efficient load-balancing across an entire cluster of computers. It is also worth studying the possibility of integrating such a system on top of a VMM supporting Virtual Machine Migration or extending the system with thread migration mechanisms for improving the load balancing.

3 Related work

In this section, we are going to focus on solutions developed in the academic world and in the industry for providing a SSI view of a cluster, particularly for providing a global address space. In section 3.1 we describe the Distributed Shared Memory (DSM) approach, as well as the consistency models that support it and the adaptations necessary to make a common application work with a specific consistency model. In section 3.2 we are going to examine systems that integrate a global address space with a software platform that can make a regular application written in Java to become cluster-aware and run seamlessly with minimal programmer intervention.

3.1 Distributed Shared Memory

Distributed Shared Memory Systems have been around for quite some time, and it was one of the first solutions adopted for clustering [9]. By extending the traditional virtual memory architecture, the distributed memory is hidden from the programmers and applications can be developed using the shared memory paradigm instead of other traditional and more error-prone, albeit more performant, parallel computing communication forms such as message-passing. Like in traditional shared memory systems, there is a possibility that two or more processors are working in the same data at the same time, and as soon as one of them updates a value the others are working in an out-of-date copy. To solve this problem, there are a significant number of possible data consistency models that were adopted by DSM implementations [9], which will be described in section 3.1.1. In section 3.1.2 we are going to describe several practical software DSM systems that were developed in the academic world. Finally, in section 3.1.3 we are going to focus on software transactional memory, an alternative and promising concurrency control mechanism analogous to database transactions.

3.1.1 Consistency models The very first consistency model was Sequential Consistency [10], which is the simplest and most restrictive consistency model. Roughly speaking, sequential consistency requires that all writes be immediately visible to all processors accessing each memory page. This synchronization in every memory access is expensive and in many cases it is stronger than necessary for a distributed application to run correctly. Therefore, a more relaxed model is needed to minimize the number of messages exchanged and the amount of data in each message, as a high amount of traffic in the network can have a serious impact on the performance of the system.

Release Consistency (RC) [11] was one of the first and most important relaxed consistency models developed for concurrent programming. This model has two synchronization operations: *acquire* and *release*. The former is used by any processor before attempting to make a write to a given object belonging to the global address space, while the latter is used after the writes are done. Therefore, in the RC model, the writes made by a certain processor p1 only need to be seen by all the other processors in the cluster after p1 releases the lock, so all writes from p1 could be queued and put in a single message which is sent to all the nodes.

Lazy Release Consistency (LRC) [12] is an algorithm similar to RC, but instead of globally propagating all changes at the time of a processor release, it postpones the propagation to the time of acquire, guaranteeing that the acquiring processor will receive all changes that "precede" the acquire operation. For example, consider a scenario where processor p1 acquires a lock over an object A, performing a few writes and then releasing it. If a processor p2 attempts to acquire a lock over A, both the lock and the writes will be propagated from p1 to p2 (and only to p2). Similarly, if another processor p3 tries to acquire the lock over A it will receive from p2 all the writes done by p1 and p2 before the p3 acquire of the lock. Therefore, for each acquire operation only one message

needs to be sent, and naturally, only the differences between each memory page need to be sent.

Entry Consistency (EC) [13] is another memory consistency model. It was first used in Midway, a programming system for distributed shared memory multicomputers. The entry consistency model takes advantage of the relationship between typical synchronization objects that define critical sections, like mutexes or barriers, and the data protected by those objects. Since a critical section defines a region where the data may have been written by another processor, and a synchronization object controls a processor's access to the data and code inside it, the view of the shared memory can become consistent only when the processor enters that same critical section. Performance measurements were promising, as the number of messages decreased a lot comparing to RC. However, comparing with LRC, the results were about the same and the need to have an explicit association between every object and a synchronization variable can be troublesome for the programmer.

Automatic Update Release Consistency (AURC) [14] is yet another release consistency model that uses automatic update to propagate and merge shared memory modifications. Automatic update is a communication mechanism implemented in the SHRIMP multicomputer that forwards local writes to remote memory transparently, which is accomplished by having the network snoop all write traffic on the memory bus and checking if the page written has an automatic memory mapping, that is, if the source process virtual address is mapped with a virtual address from a remote process. If such a mapping exists, all writes to the source page will be automatically propagated to the destination page. Consecutive written addresses are combined into a single packet, in order to reduce the network traffic. This allows for zero CPU overhead in synchronization, as the only thing a processor has to do is to store the write in the memory address as he usually would. As a result, performance is substantially increased comparing to the original LRC, but unfortunately AURC is dependent on specialized hardware support.

Scope Consistency (ScC) [15] was designed as an improvement to the EC model, offering most of the advantages without the explicit binding between variables and synchronization objects. ScC introduces a new concept called *consistency scope* to establish the relationship between data and synchronization events implicitly from the synchronization already present in programs to implement release consistency. A consistency scope consists of all critical sections protected by the same lock.

In conclusion, all consistency models can reduce communication and give some performance improvements, but they are very dependent on the applications synchronization mechanisms and may not work without some tinkering. In the next subsection, we are going to describe some systems that were developed in the academic world as proof of concepts to distributed shared memory and consistency models. The table below summarizes the consistency models main proprieties:

Table 1. Consistency models

Model	Time of propagation	Program modifications	Hardware dependent
SC	page write	None	No
RC	lock-release	acquire/release operations	No
LRC	lock-acquire	acquire/release operations	No
EC	critical section entering	object/lock association	No
AURC	page-write	None	Yes
ScC	consistency scope entering	None, if RC consistent	No

3.1.2 Software Distributed Shared Memory Systems Ivy [16] was one of the very first distributed shared memory system prototypes to be implemented and proven to be more simple than the traditional message-passing interface. Read-only pages could reside in more than one node but a page marked for writing could only reside in one node and the mapping-manager would map the writes to the remote node, guaranteeing simple sequential consistency at all times. The nodes were simple single processor machines, which means that no multithreading was considered. Unfortunately, the large size of consistency unit makes the system prone to the *false sharing* problem. The false sharing problem occurs when two or more unrelated objects are written concurrently on the same page, causing the page to "ping-pong" back and forth between the processors.

Munin [17] is a second-generation distributed shared memory system. Compared to Ivy, it was also used with simple single processor machines but it uses a release-consistent memory interface to reduce the overhead, as seen in the previous section. Also, it supports multiple consistency protocols by having the programmer annotate each shared variable to establish the protocol according to the expected access pattern, and then allowing it to change at runtime. Despite the improvements, a more transparent model to the programmer was still needed and Munin still uses a home-based protocol for handling memory pages (e.g. a memory page belongs to a node and needs to be entirely fetched on a page fault).

TreadMarks [4] is another second-generation distributed shared memory system. It uses Lazy Release Consistency to reduce the number of messages used in comparison to Munin and it also supports multiple-writers by creating a twin copy of the virtual memory page and when the modifications need to be sent to another processor the differences between the page and the twin copy are put in a separate data structure to be sent and the twin is discarded. This way, the overall bandwidth is reduced comparing to the Munin home-based approach.

Brazos [18] is a third-generation distributed shared memory system, supporting multiple multi-core processor nodes. Brazos uses a software-only implementation of Scope Consistency and a distributed page management system similar to the one in TreadMarks. Comparing to previous generation DSMs, Brazos is multithreaded and can overlap the computation with the communication latencies associated with many DSM systems. Also, it uses multicast instead of multiple point-to-point messages, reducing the communication necessary and improving the performance.

In conclusion, despite the improvements that were made to adapt the DSM concept to new hardware, all these prototypes imply a different programming approach that is impractical, as most programmers do not want to have that many worries to guarantee that the multithreaded application that is perfectly fine on one computer works correctly with a given consistency model. This problem gets even worse if the systems support multiple consistency models, and so a more transparent system is needed if we want it to be used for general-purpose applications. The table below summarizes the Software DSMs studied:

Table 2. Software DSMs

System	Consistency Model	Multithreading support
Ivy	SC	No
Munin	multiple	No
TreadMarks	LRC	No
Brazos	ScC	Yes

3.1.3 Software Transactional Memory So far, all systems and consistency models considered are based on a pessimistic lock-based approach with the definition of critical sections to protect data. A new approach called Transactional Memory [19] was developed to try to circumvent the three main issues with lock-based solutions:

- **Priority inversion:** can occur if a low-priority thread gets hold of a lock before a higher priority thread. Because there is a mutual exclusion paradigm, the higher priority thread will have to wait until the lock if free.
- **Convoying:** can occur if a thread holding a lock is preempted by the scheduler by some kind of interrupt (e.g. a page fault) resulting in other threads inability to progress.
- **Deadlock:** can occur if two threads try to get hold of the same data sets and both wait for the other to release it, being both unable to progress.

Besides these three main issues, the lock mechanism is conservative by nature and if there *might* be a conflict in a certain region, only one thread will be allowed in that section, even if at runtime the probability of conflict is not high. Therefore, a new concept of transaction was introduced in memory operations, very similar to the transactions in relational databases. Instead of having locks, all threads are allowed to execute a critical region at the same time and after finishing the operations a conflict detection algorithm is run. If there are no conflicts, the writes are made permanent into memory, otherwise the atomic operation is rolled back and retried at a later time.

There are two main approaches in implementing STMs: *transaction log* and *locks*. The former is implemented by having a transaction log local to each thread. All writes are done in the transaction log and at the end they are written to the memory after checking that there is no conflict, which means that the rollback operation is trivial but the commit operation implies much larger overhead. The latter can be further divided into two approaches: *commit-time locking* and *encounter-time locking*. The former is implemented by locking all memory locations during commit and marking access time with a global logical clock that is checked in every read/write and if the memory was accessed after the beginning of the transaction the transaction is aborted. Again, this makes the rollback simple but the commit, which should be the most common operation, expensive. The latter just gives exclusive access of the memory positions to a thread and all other threads that try to access the same memory positions simply abort, putting the largest overhead in the rollback operation, which in the STM paradigm should be the less common operation.

Unfortunately, STM also has some disadvantages that makes it still unpractical for very large systems, as the overheads from conflict detection and commit cannot be avoided. Also, some operations cannot be undone by nature (e.g I/O). Some authors have proposed new instructions for the IA-32 ISA to improve performance [20], but no standard processor available in the market adopted them yet. Sun has recently attempted to develop a multi-core processor capable of supporting hardware transactional memory [21], which unfortunately was canceled in November 2009. We believe STM has a lot of potential in the future once there is hardware support for transactions that amortizes the inherent overheads, as STM programming is far less error prone.

3.2 Distributed Virtual Machines

In section 3.1, we saw how the DSM abstraction can provide an SSI view of a cluster and the main concerns in implementing it. The next logical step is to study how we can combine this virtual memory abstraction with a platform that can make a normal application written in a high-level language cluster-aware without modifying the source code. Due to the popularity of Java programming language, not only for commercial applications but also for research due to its open-source nature, most of the systems presented in this section focus clustering of Java applications but similar approaches could be done for other high-level languages, such as C#, etc.

The current techniques used for supporting distributed execution in a cluster can be divided in three major categories. The first set can be classified as *Compiler-based DSMs* and it consists of a combination of a traditional compiler and a DSM system (see section 3.1). By compiling a normal application we can insert special instructions or bytecodes that add clustering support without modifying the source itself. The second set can be classified as *Cluster-aware Virtual Machines* and it includes implementations of Virtual Machines that provide clustering capabilities at middleware level. For instance, cJVM is a cluster-aware Virtual Machine with a global object space. The last set can be classified as *Systems using standard VMs*. In this approach, the applications will run on standard VMs that run on top of a DSM system. Some systems that rely on standard VMs also have static compilers similar to the *Compiler-based DSM* approach, with the major difference being that they transform a Java bytecode application into a parallel Java bytecode application instead of native code. Other systems, like Terracotta, perform bytecode enhancement at load-time.

3.2.1 Compiler-based DSMs The need to combine both performance and cluster-aware capabilities have led some authors to develop compilers that put special checks or instructions in the program at compile-time, adding cluster-aware capabilities without modifying the source code. The application can then be run as any native application would, in a virtual or real machine.

Jackal [22] incorporates a DSM system with a local and global GC that provides full transparency relative to the location of threads and objects. Jackal compiler generates an access check for every use of an object field or array element and the source is directly compiled to Intel x86 assembly instructions, giving the maximum performance of execution possible without a JIT. Jackal has no support for thread migration or load balancing.

Hyperion [23] also has a runtime that gives the illusion of a single memory space and it supports the remote creation of threads, which provides a better load-balancing. To keep the objects synchronized, a "master" copy is kept and updated in every write, resulting in a performance bottleneck.

In this approach, classes with native methods cannot be distributed as the already compiled code is not portable. Also, the compilation to native code indicates that these systems will only work in a homogeneous cluster, which is a severe limitation to our *Single System Image* ideal.

3.2.2 Cluster-aware Virtual Machines Many Cluster-aware Virtual Machines were developed in an attempt to provide a *Single System Image* view of a cluster, especially in Java as it is a very widely used platform for developing object-oriented applications. Java/DSM [24] was one of the very first platforms for heterogeneous computing to be able to handle both the hardware differences and the distributed nature of the system, as the alternative of developing a distributed application with RMI required extra effort from the programmer. Despite the better abstraction, Java/DSM did not explore Java semantics for performing optimizations and the load-balancing was limited since it had no

thread migration mechanisms. Also, every node needed to have a copy of every shared object, which meant that all the extra memory added by having more nodes in the cluster was just wasted.

cJVM [25] distributes the application's threads and objects over the cluster without modifying the source code or the bytecodes. Java object access an memory semantics are exploited, allowing optimization mechanisms such as caching of individual fields and thread migration. In the original implementation, there is no JIT support and it only works with an interpreter loop. To keep the objects synchronized, a "master" copy is kept and updated in every write, resulting in a performance bottleneck compared to the original Sun JVM.

Kaffemik [26] followed an approach where every object is allocated in the same virtual memory address in every machine. The biggest advantage is that the address can be used as a unique reference that is valid in every instance of every Kaffemik node. The virtual machines then work together, each containing a part of the global heap. Unfortunately, Kaffemik had no means of caching or replication which meant that an array access for example could result in several remote memory accesses, reducing performance.

JESSICA2 [27] also provides a global object space (GOS) that gives the illusion of a single heap. Each JVM heap space is divided into two sections, one that contributes to the global heap space and stores master copies of objects and another for object caching for improving performance. An interesting optimization also referred in the article is the possibility that a cached copy of an object can become the "master" copy if accessed many times, which allows locality improvements at runtime by migration of ownership of the objects. To support thread migration and be able to restore the Java thread stack in a different memory space, the stack is captured at bytecode boundary an translated into a platform-independent text format to be restored by the target JVM. JESSICA2 also supports JIT compilation, which is a major improvement relative to the previous systems.

In conclusion, the major advantage of this approach is not having to modify the applications, as all clustering is done at the VM level. Despite the very promising systems described above, all of them have a major disadvantage as they sacrifice one of the most important features of Java: cross-platform compatibility. Also, the already existing JVM facilities such as local garbage collection and JIT compiler are difficult to integrate in this type of systems. Therefore, these special cluster-aware VMs either invest a considerable amount of effort reimplementing such features or they do not implement them at all. It would be interesting if the clustering capabilities could be used with a combination of different virtual machines and in the ideal scenario we would use the standard and better supported Sun's Java Virtual Machine. This approach will be described in the next chapter.

3.2.3 Systems using standard VMs JavaParty [28] was one of the very first platforms to support the aggregation of several standard Java Virtual Machines and allow the execution of a multithreaded program in a clustered environment.

JavaParty [28] extends the Java language with a new "remote" keyword to indicate that a certain class and its instances should be visible anywhere in the distributed environment, avoiding explicit socket or RMI communication. This implementation does not fulfill the ideal SSI since the programmer has to explicitly point the classes to be clustered and needs to distinguish which invocations are remote and which ones are local because the argument passing conventions are different.

JavaSymphony [29] works under a new concept of *Virtual Architectures* that impose a virtual hierarchy on a distributed system, allowing the programmer to explicitly control locality of data and load balancing. Again, this is far from our ideal solution of having a SSI system as all objects need to be created, mapped and freed explicitly, which defeats the important advantage of built-in garbage collection in the JVM. The entire process can be quite cumbersome and since JavaSymphony does not provide assistance for these steps, the semi-automatic distribution is likely to be error-prone.

Addistant [30] works by transforming the bytecode of the Java application at load time and the developers only have to specify the host where instances of each class are allocated. All the instances of the same class are then allocated in the same node, giving poor load balancing flexibility. Moreover, the population of the cluster (number of nodes) is static and must be known in advance. System classes with native code cannot be migrated as there is no bytecode to instrument at load time. Also, application classes that use system classes with native code generate dependencies that make the former non-migratable.

J-Orchestra [31] also uses bytecode transformation to replace local method calls for remote method calls and the object references are replaced by proxy references. J-Orchestra can partition a Java program in such a way that any application object can be placed on any machine. Additionally, any object can be migrated to a different node at run-time to improve load-balancing and take advantage of a better locality. J-Orchestra also offers some run-time optimizations such as the lazy creation of distributed objects that do not suffer the overhead of registering until they need to be used. Despite these improvements, the tools provided by J-Orchestra to determine class dependencies and to ensure the correct partition requires non-trivial intervention from the user, still not achieving the SSI ideal. In addition, the bytecode instrumentation technique has the same limitation as Addistant (objects that have or depend on native code cannot be migrated).

JavaSplit [32] is yet another runtime for executing Java applications that uses bytecode instrumentation for adding clustering support. JavaSplit supports the multithreaded paradigm directly, without introducing unconventional programming constructs. All the bootstrap classes are rewritten with JavaSplit and the final result is a distributed Java application that uses nothing besides its local standard Java Virtual Machine (JVM). Each newly created thread is placed on one of the worker nodes using a load-balancing function and thread migration is not supported.

Terracotta [5] is a recent JVM-level clustering product, used in a high percentage of the companies belonging to Forbes Global 2000. Terracotta supports full transparency in a way similar to JavaSplit except that it works within an aspect-oriented programming (AOP) framework. To take advantage of the Terracotta clustering model, an instance of the Java Application needs to be launched in every node and the central Terracotta Server also needs to be setup. The programmer has to configure his Java application to decide which fields in each class remain local and which ones are going to belong to the Distributed Shared Objects space (DSO), as well as all locking and synchronization concerns. The Terracotta (TC) libraries are loaded by each JVM running the application and are responsible for handling the bytecode instrumentation at load-time for implementing the behavior specified by the programmer. The Terracotta Server implements the Virtual Memory Manager (VMM), which is responsible for holding the global heap and propagating the differences to the JVM clients (objects are cached on disc before the server runs out of memory). Also, the Terracotta Server can itself be clustered for improved scalability. Figure 1 illustrates the architecture described. The dashed squares represent a cluster node, either corresponding to a real or virtual machine.

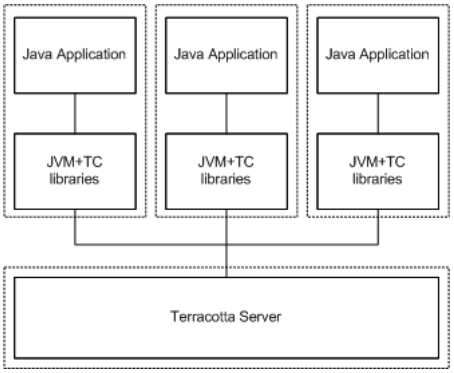

Fig. 1. Terracotta architecture

The main features of Terracotta make it an appropriate platform for clustering application servers like Apache Tomcat or JBoss, but it lacks transparency for running multithreaded applications non-cluster aware. In section 4 we are going to propose a Terracotta extension that attempts to make the scheduling of threads in a cluster transparent.

To summarize, the table on the next page illustrates the main features of all systems studied in this section. In section 3.3 we are going to study the existing scheduling algorithms and thread migration techniques in order to have a good theoretical background to choose the best approach for extending Terracotta.

Table 3. Distributed Virtual Machines

System	Application changes	Interoperability	Single heap topology	Object caching	Global GC	JIT	System classes clustering support	Load-balancing mechanisms	Thread migration
Jackal	source compilation	same ISA	Fixed compile-time object distribution	No	Yes	N.A	No	None	No
Hyperion	source compilation	same ISA	Fixed compile-time object distribution	No	Yes	N.A	No	Initial thread placement	No
Java/DSM	None	same VM	Shared objects in every node	No	No	No	No	None	No
cJVM	None	same VM	Master/proxy	No	No	No	No	Initial thread placement	Yes
Kaffemik	None	same VM	Master/proxy	No	No	No	No	Initial thread placement	Yes
JESSICA2	None	same VM	Master/proxy + flexible home	Yes	Yes	Yes	Yes	Object ownership migration	Yes
JavaParty	source/ new keyword	any standard JVM	Clustered classes in every node	No	Yes	Yes	No	None	No
JavaSymphony	None	any standard JVM	Explicit object mapping	Yes	No	Yes	No	Explicit object mapping	No
Addistant	bytecode level	any standard JVM	Master/proxy class-based	No	Partial, no cyclic garbage support	Yes	No	None	No
J-Orchestra	bytecode level	any standard JVM	Master/proxy	No	Partial, no cyclic garbage support	Yes	No	Object migration	No
JavaSplit	bytecode level	any standard JVM	Master/proxy + LRC	Yes	Yes	Yes	No	Initial thread placement	No
Terracotta	bytecode level	any standard JVM	Central Terracotta server+proxys	Yes	Yes	Yes	Yes	None	No

3.3 Clustering and thread scheduling

One of the problems considering clustering in distributed systems and software DSMs in particular is the scheduling of threads for maintaining a balanced system with a fair share load that minimizes communication and can make good enough decisions that give an acceptable performance in the long run. In this section, we will discuss a few algorithms and techniques to attempt to reach an ideal scenario where no nodes will be heavily loaded while other nodes are idle or only lightly loaded.

Load-distribution algorithms [33] can be classified in the following categories: static, dynamic and adaptive. Static algorithms are the most straight-forward approach, a new task is simply assigned to a node known a priori via a round-robin policy. An heterogeneous cluster might have an adapted weighted policy, assigning more tasks to the most powerful nodes and less tasks to the less powerful nodes, but no information about the current state of the system is used. Both these approaches have been tested in web clustering architectures [34]. Therefore, static algorithms can potentially make poor assignment decisions. For example, a new thread might be initiated in a node A, which is heavily loaded, while the local node B was idle, simply because node A was next in our fixed scheduling algorithm.

Dynamic algorithms attempt to improve the performance of their static counterparts by exploiting system-state information in runtime before making the decision. Because they must collect, store, and analyse state information they have more overheads and are harder to implement, but this extra overhead is usually compensated. In our idle node example, a dynamic algorithm could check that the node B where the new thread was initiated was idle and decide not to create the task at node A. The algorithm could even consider the state of the receiving node, possibly only assigning the new thread if the node was idle.

Adaptive algorithms are a special case of dynamic algorithms. Besides considering the system load, the system state itself can change the scheduling policies. For example, if a given policy performs better under a heavy-loaded system and another one performs better in a lightly-loaded system, an adaptive algorithm can use the former after a certain threshold of CPU load and the latter when the system reaches a lighter state, adapting itself to different workloads or application suites.

Both dynamic and adaptive algorithms raise an important issue: what is a "heavily-loaded node" and how can we define a metric that will allow us to determine if node A will be a good option for executing the next thread? Some authors like Kuntz [35] have defined the best metric as being the CPU queue length, and no significant performance was gained by using or combining other metrics such as the system call rate and the CPU utilization. In addition to the queue length metric, many authors proposed that a dynamic load-balancing system should have a priori knowledge of the resources needed by of the task in order to choose the best node. Since the node is chosen before the task is executed, the resource usage of a task must be predicted, either based on the past behavior of the task or by providing the load-balancing system with a user

estimation. Both approaches are error-prone and can have a very negative impact if used with a wrong estimation.

Choi et all. [36] have proposed a novel metric to minimize the impact of inaccurate predictions. It is known that overlapping CPU bound and I/O bound jobs results in better resource utilization, so the number of tasks considered in the CPU queue length should consider its nature. For example, a node with three CPU bound and two I/O bound should be considered as having five tasks in its queue but only three *effective tasks*, since the CPU bound overlap with the I/O bound. However, there is still a need to classify a task as "CPU bound" or "I/O bound", which is not trivial to do, and the performance improvements proved to be marginal compared to the historical-based approach.

Considering this, we can now take a deeper look to some scheduling algorithms and define what major considerations should they have to be time and space efficient. Two major requirements have been identified: good locality and low space [37]. The former means that threads that access the same memory pages should be scheduled to the same processor, as long as it is not overloaded, minimizing the overhead of page fetching, while the latter indicates that the memory requirements for the scheduling algorithm should be kept small to scale with the number of threads or processors, as in a cluster both numbers tend to grow overtime.

Work stealing schedulers [38] is a dynamic scheduling solution that provides a good compromise between the above requirements. Each processor keeps its own queue and when it runs out of threads it steals and runs a thread from another processor queue. This way, threads relatively close to each other in the computation graph are often scheduled to the same processor, providing good locality. The space required is at most S_1P, where S_1 is the minimum serial space required. This space bound can still be improved, as we will see in the next paragraph.

Depth-first search schedulers [39] is another dynamic scheduling approach. It works by computing a task graph as the computation goes by. A thread is broken into a new task by detecting certain *breakpoints* that indicate a new series of actions that can be performed in parallel by another processor (e.g. a fork). The tasks are then scheduled to a set of *worker* processors that hold two queues, one for receiving tasks (Q_{in}) and the other to put tasks created (Q_{out}), while the remaining processors are responsible to take tasks from the Q_{out} queues and schedule it to the Q_{in} queue of another processor. It was proven than the asymptotic space bound for this algorithm is $S_1 + O(p.D)$ for nested parallel computations of depth D, which is an improvement over the previous work-stealing approach. However, as the created tasks have a relative high probability of being related with the previous computation, the locality is not as good.

DFDeques [37] is a dynamic scheduling approach that seeks the best of both worlds. Threads are assigned to multiple ready queues that are depth-first ordered, similarly to the depth-first search schedulers seen in the previous paragraph. The ready queues are treated as LIFO stacks similar to the work-stealing schedulers. When a processor runs out of threads to run, it can steal from a ready

queue chosen randomly from a set of high-priority queues. The asymptotic space bound is $S_1 + O(K.p.D)$, with K being a runtime parameter which specifies the amount of memory a processor may allocate between consecutive steals. As K is usually small, the space bound is about the same as in pure depth-first schedulers and at the same time we can take advantage of thread locality as threads close to each other in the computation graph will be scheduler to the same ready queue.

These algorithms have been widely studied and were used to introduce scheduling in many parallel programming libraries and applications. Satin [40] is a Java-based grid computing programming library that implements a work stealing approach by allowing a worker node to steal a method invocation from another node. Athapascan-1 [41] is a C++ library for multithreaded parallel programming that implements a data-flow graph where both computation and data grains are explicit, allowing a depth-first scheduler algorithm to take advantage of the existing structure to schedule new threads. When considering applying one of these algorithms to a DSM system for general-purpose computations there are a few extra considerations that should be taken. We have to deal with heterogeneous nodes with different clocks and resources that may or may not be available at a certain time. This implies that a system should be dynamic and support some kind of migration of tasks to rebalance the load [7]. Also, DSMs have a much higher communication requirements than message-passing and, unlike parallel programming, we cannot predict easily the kind of applications that will run and what could be the best parallelism possible. In the subsection 3.3.1 we are going to cover implementation issues of thread migration mechanisms and how much we can benefit from them and in subsection 3.3.2 we are going to cover another approach that takes advantage of system Virtual Machines to provide a more coarse-grained but easier to implement migration mechanism.

3.3.1 Thread Migration In the previous section we studied thread scheduling in distributed shared memory systems in the perspective of initial placement of tasks, and the metrics that can be used for measuring the least loaded node. Besides the initial placement, transparent thread migration has long been used as a load-balancing mechanism to optimize resource usage in distributed environments [42]. Such systems typically use the *raw thread context* (RTC) itself as the interface between two nodes. The RTC consists of the thread virtual memory space, thread execution stack and hardware memory registers. This is the typical platform-dependent format used to represent a thread context and cannot be migrated directly without a few extra considerations. For example, the pointers or references used for data objects can be meaningless in another machine or the thread might be executing a system call that does not have any meaning in another node (e.g I/O calls). Some authors [43] have solved this issues by reserving all stacks in the beginning of the execution and guaranteeing that all of them use the same virtual addresses, and at the same time kernel threads are used to handle all system calls. This way, all threads are portable, considering

an homogeneous system, but the number of threads in each node has a fixed limit.

In the particular case of software DSMs there are a few extra considerations that should be taken into account, as increased communication can exceed the benefits of better load-balancing. For example, the original pages that were cached by a thread before migration may not be needed anymore by the source node. In contrast, the destination node will definitely need those pages and the amount of communication needed to keep consistency among the processors implies a considerable overhead. Therefore, threads for migration need to be carefully chosen in such a way that the communication overhead does not exceed the benefit given by the better load-balancing.

One of the simplest solutions is to consider the number of shared pages between pairs of threads and assume that more shared pages implies a bigger communication cost if one of the threads migrates. However, not all data-sharing results in data consistency communication as two threads can simply read the same pages without any of them changing any data. Therefore, data-sharing policies should also consider the type of memory access.

In addition, an efficient thread selection policy needs to consider *global sharing* (i.e. the communication necessary with all processors). Although such a policy will result in a much more informed decision the cost of computing the thread migration cost for each thread increases linearly with the number of processors. Other solutions involve a *partial sharing* policy, which only considers communication cost between the source and the destination node, without regard to global relations, increasing the risk of making a wrong decision. Liang et al. [44] propose a novel thread selection policy called *reduction inter-node sharing cost* (RISC) for page-based DSMs that support release-consistency, which combines the type of memory access with a global sharing policy.

Concerning thread migration in Java systems, Java has a serialization mechanism that can capture an object state and restore it in other node running another virtual machine. However, the Thread class is not serializable and the standard JVM does not provide a mechanism to access a thread stack directly. Therefore, most existing Java solutions rely on the *bytecode-oriented thread context* (BTC) as interface. The BTC is organized in a sequence of blocks called *frames*, each one associated with a Java method being executed by that thread. Each frame contains the class name, the method signature, and the activation record of the method. The activation record consists of a bytecode program counter (PC), which points to the Java instruction currently being interpreted, a JVM operand stack pointer for the stack that holds the partial results of the method execution, and the local variables of the associated method, encoded in a JVM-independent format. Considering this, and the fact that we still have to deal with threads executing native code in an RTC fashion due to system classes and JITs that compile bytecode at runtime, the following basic approaches were found in the literature [45]:

– **Static byte code instrumentation:** thread migration support is added by pre-processing the already compiled bytecode source and adding statements

which backup the thread state in a special backup object. When an application requires a snapshot of a thread state, it just has to use the backup object produced by the code inserted by the pre-processor. The main advantage of this approach is that this way the thread migration can be implemented as a simple extension that manipulates bytecode, without the need to modify the JVM. Unfortunately, the fact that there are more bytecode instructions in the code introduces significant overhead and the thread state restoration requires a partial re-execution of the application. Some implementations of this approach for mobile agents such as AMO [46] also do not consider system classes or classes with native code and cannot migrate code that uses reflection.

- **Extending the JVM and its interpreter:** thread migration support is simply added as an extension to a normal JVM interpreter, as done in systems such as JESSICA [47]. This is accomplished by having a global thread space that spans the entire cluster and a mechanism that can separate the hardware-dependent contexts in native code and the hardware-independent contexts at bytecode level. This way, a thread can migrate with relatively good granularity between each bytecode instruction that is interpreted. However, modifying the JVM interpreter do deal with thread migration adds to the overhead of the already slow interpreter. This approach has been proven to have better performance than the previous static byte code instrumentation [48] but it is still much slower than the creation of a normal thread, which gives the impression that support for JITed code is needed.
- **Using the JVM Debugger Interface (JVMDI):** thread migration support is added by compiling Java applications with extra debugging information that allows access to the thread stack as well as the introduction of thread migration points. Modern debugger interfaces also support JIT compilers, as previous approaches only considered bytecode. However, JVMDI needs huge data structures and incurs large overhead to include the extra general debugging features and the limited optimizations that can be done in a debugging environment.

CEJVM [49] is a master-worker approach that relies on a master node that runs the Java application and delegates threads to worker nodes. It uses the JVMDI to implement thread migration transparently and compatible with any JVM that supports the debugger interface. Performance-wise, the master-worker paradigm only works well with a specific niche of applications and it would be desirable that all nodes be provided with thread migration capabilities in a point-to-point way.

Cho-Li et all [50] define a new approach that consists of integrating the RTC to BTC conversion and the implicit stack capturing and restoration directly inside the JIT. Stack capturing involves using the JIT to instrument native codes and transform them back into the platform-independent bytecode format. This way, the thread scheduler itself can perform on-stack scanning and to derive the BTC format instead of using a stand-alone process like in the JVMDI approach. For stack restoring, the authors introduce a mechanism called *Dynamic Register*

Patching that rebuilds the state of the hardware registers before returning the control to the thread instanced in the new node.

Another issue that we need to address is at which code points should migration be considered as a good option. The simplest approach is to allow migration in any bytecode boundary. However, with all the JVMs running sophisticated JITs there is a high probability that the execution is running native code at the time of migration and it may be very hard and inefficient to simulate the native instructions from the stopped point until the next bytecode boundary for migration. Cho-Li et all [50] define two basic points: the beginning of a Java method invocation and the beginning of a code block pointed by a back edge in the computational graph. The former indicates a new operation that can most likely be done in another node (very small methods that do not typically compensate will be inlined by the compiler and not considered for migration), while the latter represents the beginning of a loop, which is also a good option as it needs a more or less prolonged computation until it finishes. Intra-bytecode migration semantics would be very ambiguous and difficult to implement, so we are not considering it in this report. It is preferable, for example, to turn off the JIT compiler before migration and only enable it after scheduling on another node.

Finally, we have to deal with the type resolution of the operands. As operands in a thread context are pushed in and popped out of the stack at runtime, their types cannot be determined in advance. The simplest solution is to have a separate stack for operand types synchronized with the normal Java stack. This doubles the time in accessing the operand stack, which can be more or less significant depending on the number of possible migration points that we are considering. Also, this approach can be optimized as most types can be verified statically by the Java bytecode verifier [51]

3.3.2 Virtual Machine Migration The need to provide a cluster to support multiple operating systems, applications, and heterogeneous hardware has led to the development of Virtual Machine Monitors (VMM) or hypervisors that run right on top of the hardware and schedule one or more operating systems across the physical CPUs. The live migration mechanism is less granular than thread migration, as a system VM might have a large number of threads running simultaneously and the migration of an entire system VM to another node requires that the recipient node has indeed more resources to run the system VM. However, a recent performance study made by Chen et all. [8] using a page-based DSM system shows that the virtual machine migration approach can compete with thread migration and it has the advantage of providing a cleaner separation between hardware and software, as well as facilitating fault-tolerance and load-balancing.

IBM have developed the z/VM solution [52], an hypervisor software capable of supporting several thousands of Linux servers running on a single mainframe. z/VM supports full scheduling of user virtual machines according to each user needs by monitoring resource usage and giving a user class from 0 to 3. Higher class users get longer time-slices but lower classes tasks are given a higher priority

if the mainframe resources get constrained. Despite the good transparency and scheduling solution, the system only runs on the mainframe zSeries IBM servers, which are not available to the majority of programmers.

Xen [53] is another hypervisor that runs on standard x86 machines, developed in the University of Cambridge. Xen supports many popular operating systems such as Solaris, Linux and Windows. System administrators can migrate Xen Virtual Machines between physical hosts across a LAN without loss of availability

4 Architecture

In section 3.2.3 we studied several systems for clustering Java applications that use standard JVMs, including Terracotta. We recap here the three main issues that make this system impractical for non-cluster aware multithreaded applications:

- **Non-transparent configuration:** the programmer of the application needs to configure the classes to be instrumented for sharing fields among JVM instances (objects shared in this way are called *shared roots*). Methods that manipulate shared roots also need to be configured in order to ensure mutual exclusion between the nodes.
- **No scalability for multithreading applications:** Terracotta has no means of running a multithreaded application in a transparent way that takes direct advantage of the clustering interface. When a thread is created in Terracotta, either by extending the Thread class or implementing the Runnable interface, we are limited to have the same thread in each local node, as it is considered a JVM-specific non-portable resource by the Terracotta developers.
- **No load-balancer:** The Terracotta programmer needs to be concerned about how much data each JVM should have, as Terracotta Virtual Memory Manager (VMM) needs to be tuned for dealing with large object graphs.

Considering the limitations, we intend to design and implement a prototype that extends the Terracotta shared memory object space with a global scheduler. The following functionalities are desired:

- **Phase 1:** when a new thread is launched in Terracotta, it should be created in the least loaded node. The metric for deciding which node is the "least loaded" could be the CPU queue length, as seen in section 3.3.
- **Phase 2:** run multiple instances of Terracotta on top of multiple system VMs instances and attempt load-balancing through live migration.
- **Phase 3:** when too many threads are created in one node, there should be a mechanism to rebalance the load across the cluster by migrating some threads.

First, we need to define a "proxy" class for the Java Thread that can be clustered in Terracotta (ClusterThread). The Java application will be instrumented at bytecode level to use our custom class instead of the native Thread class. This

way, when a JVM node starts the execution of the ClassThread, it will inform a special coordinator component of the current CPU queue length asynchronously and continue its execution. The coordinator will then determine the less loaded node and notify it, in which it will create a new native Thread and execute the code. With this approach, we can solve the first main requirement of our work.

For phase 2, we will test Terracotta over instances of system VMs on an hypervisor software such as Xen and implement a monitor daemon in each node to keep track of the number of threads. After it reaches a certain threshold, the monitor daemon triggers the Xen VM migration mechanism to migrate the VM instance to another node. This can be done by communicating with the coordinator to check which node is less loaded or simply choose a neighbour node for migration. The main idea is to help balance the load in a simple way, thus preventing imbalance in load from getting unbounded, which is already a major improvement.

For phase 3, we will need to extend the monitor with the capability to serialize a thread's state, using the bytecode boundary approach described in 3.3.1. This phase is the most tricky part to implement and the possible performance gains compared to the system VM approach still remain an open issue.

Finally, figure 2 shows the final architecture of the Terracotta extension we propose, followed by a summary of all components. The dashed squares represent a cluster node, either corresponding to a real or virtual machine.

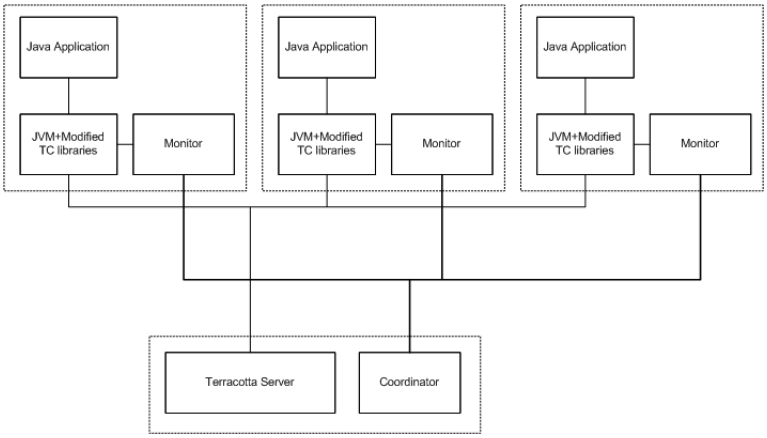

Fig. 2. Terracotta global scheduler architecture

- **Java Application:** Multithreaded Java application to be executed.
- **JVM+Modified TC libraries:** Standard Sun's Java Virtual Machine running Terracotta libraries for instrumenting classes at load-time, modified to use the special ClusterThread class.

- **Monitor:** Monitors the load in the local JVM and initializes migration mechanisms, if needed.
- **Terracotta Server:** Holds the global heap and propagates the differences to the JVM clients.
- **Coordinator:** Knows the current load of each node and sends messages to initiate threads, when needed.

5 Methodology and Evaluation

In this section, we are going to describe the metrics and benchmarks to use in order to measure the improvements we intend to achieve with the Terracotta Global Scheduler described in the previous section. Since we are interested in measuring the performance in the context of High Performance Computing, we need to measure the performance of execution of a computational intensive Java application (e.g. ray tracing calculations), and some low-level mechanisms such as the creation or joining of threads. Java Grande provides multithreading benchmarks that could be used in our prototype [54]. Java Grande multithreading benchmarks focus on the following aspects:

- **Low level operations:** thread creation and joining, barrier synchronization, etc. The main objective of these micro-benchmarks is to measure the number of low-level operations in a fixed period of time to have an idea of the additional overhead impact made by our prototype.
- **Kernels:** Fourier coefficients calculation, LU factorization, matrix multiplication, etc. These benchmarks focus on measuring the performance of scientific and numeric computing. For each benchmark, the time taken and a performance in operations per second (where the units are benchmark-specific) are reported.
- **Large Scale applications:** 3D ray-tracing applications and Monte Carlo simulations. Ray-tracing applications are a very interesting scenario due to the computational independence of each ray that makes it highly parallelizable. Monte Carlo simulations is also a good scenario as Monte Carlo methods are very important in todays physical and mathematical simulators and the random sampling techniques they use are completely parallelizable.

With this benchmark, we test our prototype in a large number of scenarios and we believe that the extra overhead of thread creation and synchronization will not be significant compared to the performance gain of parallelizing computation. Besides benchmarking specific scenarios, it is also desirable to test with real-world applications and verify that our prototype allows them to take advantage of the extra computational power and memory available in the cluster.

The DaCapo benchmark suite [55] uses several open source Java applications such as Antlr, HSQLDB, Jython, Xalan, etc. All this applications have non-trivial memory loads and provide very different scenarios between themselves. Antlr is a language recognition tool to generate parser and lexical analysers while Xalan is a parser for transforming XML into HTML documents. We expect, for

example, to be able to parse a larger number of grammar files or XML documents with our prototype, as well as a performance increase due to the ability to have more data in memory distributed across the cluster. HSQLDB is a relational database engine written in Java, on which we expect to increase the number of possible transactions.

6 Conclusion

In this report we presented several systems that focus on providing a SSI view of a Java cluster. We believe that approaches that use standard JVMs are the most likely to succeed due to the Sun JVM being present in the most common architectures and operating systems. The techniques for thread scheduling in distributed environments covered here can be integrated with a system that already provides a shared object space, giving common programmers the ability to run a regular multithreaded application in a cluster seamlessly, without worrying about load-balancing. The description of our prototype, based on Terracotta, is generic as it still in development but the final result has the potential to improve the cluster programming paradigm by exploring a new usage for Terracotta, beyond the clustering of application servers like Apache Tomcat or JBoss.

References

1. Mendel Rosenblum. The reincarnation of virtual machines. *Queue*, 2(5):34–40, 2004.
2. James E. Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005.
3. Rajkumar Buyya, Toni Cortes, and Hai Jin. Single system image. *Int. J. High Perform. Comput. Appl.*, 15(2):124–135, 2001.
4. Christiana Amza, Alan L. Cox, Sandhya Dwarkadas, Hya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29:18–28, 1996.
5. Terracotta. A technical introduction to terracotta. 2008.
6. Kritchalach Thitikamol and Pete Keleher. Thread migration and communication minimization in dsm systems. In *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, pages 487–497, 1999.
7. Kritchalach Thitikamol and Peter Keleher. Thread migration and load balancing in non-dedicated environments. *Parallel and Distributed Processing Symposium, International*, 0:583, 2000.
8. Po-Cheng Chen, Cheng-I Lin, Sheng-Wei Huang, Jyh-Biau Chang, Ce-Kuen Shieh, and Tyng-Yeu Liang. A performance study of virtual machine migration vs. thread migration for grid systems. *Advanced Information Networking and Applications Workshops, International Conference on*, 0:86–91, 2008.
9. J Protić, M Tomašević, and V Milutinović. *Distributed Shared Memory: Concepts and Systems*. John Wiley & Sons, 1998.
10. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programm. *IEEE Trans. Comput.*, 28(9):690–691, 1979.
11. Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *In Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.
12. Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *In Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, 1992.
13. B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The midway distributed shared memory system. In *Compcon Spring '93, Digest of Papers.*, pages 528–537, Feb 1993.
14. Liviu Iftode, Cezary Dubnicki, Edward W. Felten, and Kai Li. Improving release-consistent shared virtual memory using automatic update. In *In The 2nd IEEE Symposium on High-Performance Computer Architecture*, pages 14–25, 1996.

15. Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope consistency: A bridge between release consistency and entry consistency. In *In Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 277–287, 1996.
16. Kai Li. Ivy: a shared virtual memory system for parallel computing. pages 94–101, August 1988.
17. John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of munin. *SIGOPS Oper. Syst. Rev.*, 25(5):152–164, 1991.
18. Evan Speight and John K. Bennett. Brazos: A third generation dsm system. In *IN PROCEEDINGS OF THE 1ST USENIX WINDOWS NT SYMPOSIUM*, pages 95–106, 1997.
19. Maurice Herlihy, J. Eliot B. Moss, J. Eliot, and B. Moss. Transactional memory: Architectural support for lock-free data structures. In *in Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.
20. Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural support for software transactional memory. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.
21. Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. Rock: A high-performance sparc cmt processor. *IEEE Micro*, 29(2):6–16, 2009.
22. R. Veldema, R.A.F. Bhoedjang, and H.E. Bal. Distributed shared memory management for java. In *In Proc. sixth annual conference of the Advanced School for Computing and Imaging (ASCI 2000*, pages 256–264, 1999.
23. Gabriel Antoniu, Luc Boug, Philip Hatcher, Mark MacBeth, Keith Mcguigan, and Raymond Namyst. The hyperion system: Compiling multithreaded java bytecode for distributed execution, 2001.
24. WEIMIN YU and ALAN COX. Java/dsm: A platform for heterogeneous computing. 1997.
25. Yariv Aridor, Michael Factor, and Avi Teperman. cjvm: a single system image of a jvm on a cluster. In *In Proceedings of the International Conference on Parallel Processing*, pages 4–11, 1999.
26. J. Andersson, S. Weber, E. Cecchet, C. Jensen, V. Cahill, J. Andersson Ý, S. Weber Ý, E. Cecchet Þ, C. Jensen Ý, V. Cahill Ý, and Trinity College. Kaffemik - a distributed jvm on a single address space architecture, 2001.
27. Wenzhang Zhu, Cho-Li Wang, and Francis C. M. Lau. Jessica2: A distributed java virtual machine with transparent thread migration support. *Cluster Computing, IEEE International Conference on*, 0:381, 2002.
28. Matthias Zenger. Javaparty - transparent remote objects in java, 1997.
29. Thomas Fahringer. Javasymphony: A system for development of locality-oriented distributed and parallel java applications. In *In Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER 2000*. IEEE Computer Society, 2000.
30. Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano. A bytecode translator for distributed execution of "legacy" java software. pages 236–255. Springer-Verlag, 2001.
31. Eli Tilevich and Yannis Smaragdakis. J-orchestra: Automatic java application partitioning. pages 178–204. Springer-Verlag, 2002.
32. Michael Factor, Assaf Schuster, and Konstantin Shagin. Javasplit: A runtime for execution of monolithic java programs on heterogeneous collections of commodity workstations. *Cluster Computing, IEEE International Conference on*, 0:110, 2003.
33. Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal. Load distributing for locally distributed systems. *Computer*, 25(12):33–44, 1992.
34. Valeria Cardellini, Michele Colajanni, and Philip S. Yu. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, 3(3):28–39, 1999.
35. Fachbereich Informatik, Technische Hochschule Darmstadt, Thomas Kunz, and Thomas Kunz. The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering*, 17(17):725–730, 1991.
36. Min Choi, Jung-Lok Yu, Ho-Joong Kim, and Seung-Ryoul Maeng. Improving performance of a dynamic load balancing system by using number of effective tasks. *Cluster Computing, IEEE International Conference on*, 0:436, 2003.
37. Girija J. Narlikar. Scheduling threads for low space requirement and good locality. In *SPAA '99: Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, pages 83–95, New York, NY, USA, 1999. ACM.
38. Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.
39. Guy E. Blelloch, Phillip B. Gibbons, Girija J. Narlikar, and Yossi Matias. Space-efficient scheduling of parallelism with synchronization variables. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 12–23, New York, NY, USA, 1997. ACM.
40. Rob Van Nieuwpoort, Jason Maassen, Thilo Kielmann, and Henri E. Bal. Satin: Simple and efficient java-based grid programming. In *In AGridM 2003 Workshop on Adaptive Grid Middleware*, 2005.

41. Gerson G. H. Cavalheiro, François Galilée, and Jean louis Roch. Athapascan-1: Parallel programming with asynchronous tasks. In *Yale University*, page 98, 1998.
42. Bozhidar Dimitrov and Vernon Rego. Arachne: A portable threads system supporting migrant threads on heterogeneous network farms. *IEEE Transactions on Parallel and Distributed Systems*, 9:459–469, 1998.
43. Ayal Itzkovitz, Assaf Schuster, and Lea Shalev. Thread migration and its applications in distributed shared memory systems. *Journal of Systems and Software*, 42:71–87, 1997.
44. Tyng-Yeu Liang, Ce-Kuen Shieh, and Jun-Qi Li. Selecting threads for workload migration in software distributed shared memory systems. *Parallel Comput.*, 28(6):893–913, 2002.
45. Bouchenak Hagimont Sirac, S. Bouchenak, and D. Hagimont. Approaches to capturing java threads state. In *In Middleware 2000*, 2000.
46. Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa. Bytecode transformation for portable thread migration in java. In *ASA/MA 2000: Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents*, pages 16–28, London, UK, 2000. Springer-Verlag.
47. Matchy J. M. Ma, Cho-Li Wang, and Francis C. M. Lau. Jessica: Java-enabled single-system-image computing architecture. *J. Parallel Distrib. Comput.*, 60(10):1194–1222, 2000.
48. Sara Bouchenak and S. Bouchenak. Pickling threads state in the java system. In *Third European Research Seminar on Advances in Distributed Systems (ERSADS'99*, 1999.
49. M.U. Janjua, M. Yasin, F. Sher, K. Awan, and I. Hassan. Cejvm: "cluster enabled java virtual machine". *Cluster Computing, IEEE International Conference on*, 0:389, 2002.
50. Wenzhang Zhu Cho-Li, Cho li Wang, and Francis C. M. Lau. Lightweight transparent java thread migration for distributed jvm. In *In International Conference on Parallel Processing*, pages 465–472, 2003.
51. James Gosling and Henry McGilton. The java language environment: A white paper, May 1996. urlhttp://java.sun.com/.
52. Lydia Parziale, Eli M. Dow, Klaus Egeler, Jason J. Herne, Clive Jordan, Edi Lopes Alves, Eravimangalath P. Naveen, Manoj S Pattabhiraman, and Kyle Smith. *Introduction to the new mainframe: z/vm basics.* IBM Corp., Riverton, NJ, USA, 2007.
53. Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.
54. L. A. Smith and J. M. Bull. A multithreaded java grande benchmark suite. In *In Proceedings of the Third Workshop on Java for High Performance Computing*, pages 97–105, 2001.
55. Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, New York, NY, USA, 2006. ACM.