Distributed Clustering and Scheduling of VMs

João Lemos
joao.lemos@ist.utl.pt

ABSTRACT

In this work, we have developed Caft, a middleware that runs on top of the Terracotta system and has the capacity to run simple multi-threaded Java applications in a transparent way, scheduling threads across the several nodes in a Terracotta cluster and taking advantage of the extra computational and memory resources available. We use bytecode instrumentations to add clustering capabilities to the multi-threaded Java application, as well as extra synchronization if needed. The middleware supports several modes, in order to achieve a balance between transparency and flexibility. We tested the middleware with a Fibonacci computing application, an Open Source renderer (Sunflow) and an application that multiplies a matrix by a vector. We concluded that our middleware is scalable, as it allows a multi-threaded application to achieve lower execution times by adding more nodes to a Terracotta cluster.

Keywords

Java, Parallel and Distributed Computing, Single-System Image, Bytecode Instrumentation, Terracotta

1. INTRODUCTION

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* [10]. Much research has been done in the area of SSIs, such as Distributed Shared Memory (DSM) systems and Distributed Virtual Machines that can run applications written in a high-level language in a cluster, behaving as if it were on a single machine. There are three major approaches for implementing these kind of systems:

- **Extend a programming language at source or bytecode level:** allows a simple and straight-forward implementation. 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.
- **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, no modern DSM system can provide the full transparency desired for running already existent applications. The current most popular system that uses the concept of a shared object space is Terracotta. At present, Terracotta has no concept of global thread scheduler and the programmer of a multi-threaded application needs to be concerned about manually launching multiple instances of the applications, and manual load-balancing. Considering these limitations, we believe that if we had a middleware that could bridge both Terracotta and multi-threaded Java applications, handling the scheduling of threads and using the existent shared object space to keep data consistent, we could run already existing applications in a distributed environment with almost no extra effort and obtain scalability. This belief holds the main motivation for developing **Caft**, a Cluster Abstraction for Terracotta.

1.1 Development of Caft

The Caft middleware can be configured to either run as a **master** or as a **worker**. The former will load and run the desired multi-threaded Java application, while the latter will wait for requests from the master to run threads. The idea is to deploy one master and several workers and be able to obtain scalability by having more CPUs and memory available for running threads and parallelizing the application more than it would be possible with a single node. Also, considering the performance versus transparency trade-of, as well as the availability of source code, we developed Caft with three different modes: Identity, Full SSI and Serialization. Identity mode should be used if we have a multi-threaded Java application that is properly synchronized, or the programmer has access to the source code and can add synchronization with ease. Full SSI mode should be used if we have a multi-threaded Java application that is not properly synchronized, or the programmer has no access to the source code. In both modes, all fields belonging to a Java `Runnable` target that is passed to the Thread class will be shared. Serialization mode allows the programmer to specify the fields that need to be shared using Java annotations, allowing for a more fine grained configuration.

1.2 Document Roadmap

The rest of this paper is organized as follows. Section 2 describes the context of our work, including some SSI systems that share our topics of interest. Section 3 describes the architecture of the middleware developed, using Terracotta as an infrastructure for running multi-threaded applications. Section 4 describes the implementation of the middleware in further detail, focusing on the bytecode instrumentations that it performs on the Java application. Section 5 describes the evaluation method to measure solution ad-

equacy and performance. Section 6 summarizes all work done, and draws some conclusions.

2. 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 2.1 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 2.2 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. To finalize, in section 2.3 we are going to focus on scheduling algorithms and migration techniques to improve load-balancing.

2.1 Distributed Shared Memory

Distributed Shared Memory (DSM) Systems have been around for quite some time, and it was one of the first solutions adopted for clustering [28]. 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 [28]. We studied Sequential Consistency (SC)[23], Release Consistency (RC) [17], Lazy Release Consistency (LRC) [22], Entry Consistency (EC) [6], Automatic Update Release Consistency (AURC) [19] and Scope Consistency (ScC) [20]. All these consistency models can reduce communication and give some performance improvements, but they are very dependent on the applications synchronization mechanisms and the programmer must be aware of the underlying model for the application to work properly.

The software DSM systems studied include Ivy [24], Munin [11], TreadMarks [1] and Brazos [31]. All these prototypes imply a different programming approach that is impractical, as they rely on one or more of the consistency models defined. It is not desirable to have to understand a complex consistency model in order to guarantee that a multi-threaded application that is perfectly fine on one computer works correctly on a software DSM.

2.1.1 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 [18] was developed to try to circumvent the three main issues with lock-based solutions: Priority inversion, convoying and deadlocks. 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,while the latter just gives exclusive access of the memory positions to a thread. Unfortunately, as there is no standard hardware support for transactions, the overheads from conflict detection and commit cannot be avoided.

2.2 Distributed Virtual Machines

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 2.1). 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. 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.

2.2.1 Compiler-based DSMs

The systems studied that fit in this category are Jackal [35] and Hyperion [3]. Both have good performance as the application runs on native code. In this approach, classes with native methods cannot be distributed as the already compiled code is not portable. Also, these systems will only work in a homogeneous cluster.

2.2.2 Cluster-aware Virtual Machines

The systems studied that fit in this category are Java/DSM [36], cJVM [4], Kaffemik [2] and JESSICA2 [38]. The major advantage of this approach is not having to modify the applications, as all clustering is done at the VM level. Unfortunately, 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.

2.2.3 Systems using standard VMs

The systems studied that fit in this category are JavaParty [37], JavaSymphony [16], Addistant[32], J-Orchestra [34], JavaSplit [15] and Terracotta [33]. This approach has full support for existing features in standard VMs, such as Local GC and JITs. Also, by relying on bytecode transformations and configurations, there is little need to modify applications directly. However, these instrumentations require some work from the programmer, and it is difficult to integrate classes with native code in these type of systems.

2.3 Clustering and thread scheduling

Load-distribution algorithms [29] 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. Dynamic algorithms attempt to improve the performance of their static counterparts by exploiting system-state information in runtime before making the decision. Adaptive algorithms consider the system load, and the system state itself can change the scheduling policies.

Both dynamic and adaptive algorithms raise an important issue: what is a “heavily-loaded node”? Some authors like Kuntz [21] 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. Also, a good scheduling algorithm should fulfil two basic requirements: good locality and low space [26]. The former means that threads that access the same memory pages should be scheduled to the same processor, 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.

Work stealing schedulers [8] is a dynamic scheduling solution where 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.

Depth-first search schedulers [7] is another dynamic scheduling approach, computing a task graph 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. This algorithm aims for lower memory requirements, but it has worse locality than work stealing schedulers.

DFDeques [26] is a dynamic scheduling approach that seeks the best of both worlds. Threads are assigned to multiple ready queues that are depth-first ordered, and are treated as LIFO stacks similar to the work-stealing schedulers.

2.3.1 Thread Migration

Besides the initial placement, transparent thread migration has long been used as a load-balancing mechanism to optimize resource usage in distributed environments [14]. It should be noticed that the communication costs should not exceed the costs of migration. Concerning Java systems, the following approaches to perform thread migration were found in the literature [30]:

- **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.
- **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 [25]. 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.
- **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.

Another issue that we need to address is at which code points should migration be considered as a good option. Cho-Li et all [13] 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.

2.3.2 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. However, a recent performance study made by Chen et all. [12] suggests that the virtual machine migration approach can compete with thread migration. IBM have developed the z/VM solution [27], an hypervisor software capable of supporting several thousands of Linux servers running on a single mainframe. Xen [5] is another hypervisor that runs on standard x86 machines, supporting many popular operating systems such as Solaris, Linux and Windows.

3. ARCHITECTURE

This section describes the middleware **Caft** (Cluster Abstraction for Terracotta), developed during this work to allow Terracotta to run simple Java multi-threaded applications with minimum changes or concerns due to the different environment. We will start by familiarizing the reader with the mechanisms already offered by Terracotta that motivate it to be a very good choice for clustering application servers such as Tomcat or JBoss. After this introduction, we will describe the high-level architecture of the middleware, as well as all compromises assumed. To finalize, we are going to present the packages and classes that compose the middleware, along with a description of their functions and data structures used.

3.1 Terracotta

As Terracotta clusters JVMs transparently with no explicit API, control over what gets clustered and which operations in the application are sensitive to clustering is performed through the Terracotta configuration. The three main sections of the Terracotta configuration that must be specified by the developer are: roots, locks, and classes to instrument.

- **Root**: A root defines a field to be put in the global heap and shared across all JVMs, maintaining object identity. A root is what forms the top of a clustered object graph and allows Terracotta to distinguish which objects are shared and which are not.
- **Locks**: Access to shared roots need to be locked in Terracotta, in order to guarantee proper data consistency. It is the only allowed way to access shared objects in Terracotta. Terracotta supports auto and named locks. The former allows Terracotta to use already existing synchronization present in the methods that access shared objects while the latter allow the definition of a global lock across the cluster.
- **Instrumented Classes:** Classes that access shared roots, or are shared themselves in the global heap, need to be instrumented at bytecode level to guarantee that Terracotta applies modifications and adds proper locking.

With the current version of Terracotta, threads created never leave the home node and adapting an existing application implies that the programmer needs to add synchronization where needed, which in case of a large application can be troublesome.

3.2 Caft - a middleware that extends Terracotta

The Caft middleware has two major components: **worker** and **master**. The former runs a Thread Service that provides the interface for instantiating new threads, as well as the operations provided by the Java Thread Class (whose methods can be regarded as an implicit interface), while the latter runs the main class of a runnable Jar containing a multi-threaded application, spawning threads in worker machines as necessary. It is assumed that the Jar needs to be available on both the master and the workers.

Both master and workers need to share the thread fields whose identity must be preserved across the cluster and its changes propagated. The master opens the Jar passed as argument, detects the class defined as the main entry point and runs the main method using the Java reflection API. The master uses a custom Classloader, also present in the worker, that applies the instrumentations necessary to make the Thread calls cluster aware, and/or adding synchronization. Bytecode instrumentations are made using the ASM framework [9], allowing us to add methods and changing calls without much overhead. For the master and worker communication, we use simple RMI calls supported by the Spring framework to ease development and configuration.

To simplify the implementation, the coordinator component that decides which node gets to execute the next thread is integrated as a singleton in both components. The data structures that compose the state of the coordinator, such as which nodes are available and their loads, are maintained as roots in Terracotta's Distributed Share Objects (DSO) space. This approach also avoids the need to have an extra node that serves as a coordinator and the persistence of its state is guaranteed by the Terracotta Server.

To better illustrate our design, we present the Terracotta architecture in Figure 1 with the Caft middleware, running a worker in one of the Terracotta clients and a master in another. The middleware runs on top of Terracotta, loading the application and performing bytecode instrumentations at load time. If configured for running a worker, Caft will start an RMI service using the Spring framework, keeping the Java application in its own class path to ensure everything works when it receives a Runnable target to execute in it. If configured to run a master, it will simply run the application, just as already described.

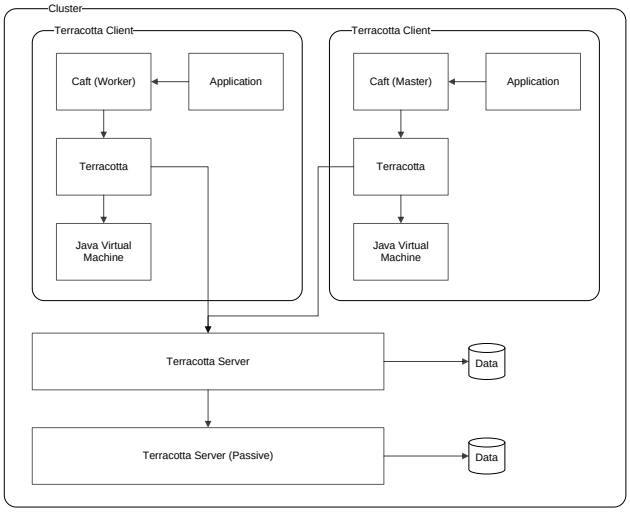

Figure 1: Terracotta architecture running Caft

Considering that we need to have a trade-off between transparency and performance, as less transparency should allow for better customization and tuning, we developed Caft with three different modes. The mode to be used is passed as an argument to both master and workers, and they should not be mixed. The modes supported are presented in the list below:

- **Identity:** Identity mode assumes that the application is properly synchronized, or at least, that the user has access to the source code and can add synchronization with more or less work. All thread fields are shared in the Terracotta DSO to ensure that the writes are propagated and all methods are annotated with the `AutolockWrite` Terracotta annotation, so that each synchronized block can be converted into a Terracotta transaction.

- **Full SSI (Single System Image):** Full SSI mode assumes that the application lacks proper synchronization for usage with Terracotta, or the source code is not available. Full SSI behaves just like Identity mode but with extra instrumentations that add getters and setters to each field, with proper synchronization, and it also synchronizes array writes.

- **Serialization:** Serialization mode allows the user to decide which fields of the Runnable class to be run in a Thread are meant to be clustered and have identity preserved, and the rest are simply serialized and copied via RMI, allowing for local thread variables that do not really need synchronization.

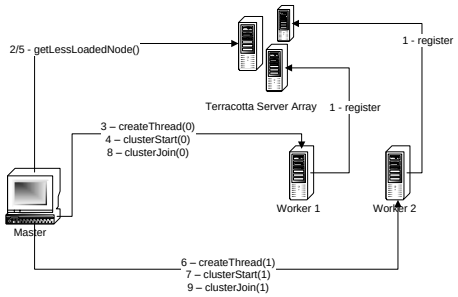

Figure 2: Terracotta deployment scenario with Caft

As an example, consider the following deployment scenario, illustrated by Figure 2, with two worker machines that will receive Java Runnable targets and use them to create local threads, one master that will run the application, and a Terracotta Server Array holding the DSO.

In this example, the master is running a multi-threaded application that launches two threads and waits for them to finish. The worker machines register themselves with the Coordinator in step one, whose state is shared in the Terracotta Server Array in order to be accessible by the other machines. The master checks the coordinator state in step two, which determines that **Worker 1** is the less loaded node (at this point, could be either of them as both never had any thread assigned). In step three, the master sends the thread ID to that worker and makes the Runnable target available to it, either copying it via RMI or putting it in the Terracotta DSO, depending on the mode used. The worker creates a local instance of a Java Thread using the Runnable target, which is started also by a remote call of the master in step four. The master will then attempt to create another thread, which after consulting the Coordinator in step five it returns the node **Worker 2** as being the most appropriate, as the master already assigned a thread to Worker 1. A thread is created and started in Worker 2 in steps six and seven, analogous to the first thread created in Worker 1. After this, the master joins both threads, illustrated by steps eight and nine, making an RMI call to the workers which will execute a local join in the Java Thread object corresponding to the thread. As we are using the DSO in Terracotta, the master will see every relevant change in the objects passed as a Runnable target to both threads.

4. IMPLEMENTATION

In Caft, we instrument the application classes using the ASM framework [9]. We developed a method adapter named `AddClusterThreadAdapter` for implementing the indirections necessary for replacing Java Thread instantiations and method calls with our special ClusterThread class. We also developed a class adapter named `ThreadClassAdapter` that applies method adapters and adds annotations, depending on the mode chosen by the user.

4.1 Thread instrumentations

The `AddClusterThreadAdapter` instrumentation replaces Java type opcodes that have the Java Thread type as argument with equal opcodes with the ClusterThread type. It also replaces the `getfield` and `getstatic` opcodes type with ClusterThread instead of Thread. As the ClusterThread class extends the original Java Thread class, type compatibility is guaranteed. For the method calls, some of the methods belonging to the Thread class are final, and therefore cannot be overridden. To circumvent this, we renamed the final methods and replaced Thread method calls with the renamed method. For example, if we have an `invokevirtual` opcode that invokes the final "join" method of the Thread class, we invoke the "clusterJoin" method instead.

The `ThreadClassAdapter` instrumentation is the class adapter responsible for adding Terracotta annotations and applying instrumentations, depending on the mode chosen. This instrumentation applies the `AddClusterThreadAdapter` to all methods of the Java multi-threaded application that is going to run in the middleware.

In Identity mode, the `ThreadClassAdapter` adds the Terracotta `AutolockWrite` annotation, in order to take advantage of the local synchronization to add a Terracotta transaction in every method. In Full SSI mode, the `ThreadClassAdapter` also applies the `GetterSetterAdapter` instrumentation for adding synchronization at its lowest level, on field access and array writes. To finalize, in Serialization mode, this instrumentation applies the `CaftRootAdapter` instead. The `CaftRootAdapter` instruments access to specific fields annotated by the programmer, and does not apply any Terracotta annotation, as it is not needed due to the fact that the remaining fields will be serialized. Also, this instrumentation adds the Serializable interface to every class automatically.

4.2 Fullssi instrumentations

For adding getters, we implemented an ASM class adapter transformation that adds a getter for each non-static field. Each getter has the Java `synchronized` method modifier and is annotated with the Terracotta `AutolockRead` annotation to allow for concurrent reads of the field, but still in the context of a Terracotta transaction. For generating setters, we implemented a similar class adapter, with the corresponding `AutolockWrite` annotation. We also developed equivalent instrumentations for static fields.

To use the getters and setters generated, we developed a method adapter that replaces direct field accesses with method calls. As such, the method adapter replaces the `getfield` and `putfield` instructions with `invokevirtual` instructions that will invoke the generated corresponding getters and setters. Equivalent `getstatic` and `putstatic` instructions will be replaced by `invokestatic` instructions that will invoke the corresponding static getters and setters. In array access, writes using array store instructions also need synchronization at some point if the array is shared by Terracotta. Considering this scenario, we developed a new class with static methods that consumes exactly the same arguments and performs the array store inside a synchronized block. Our method adapter will then replace the array store instruction by an invocation of the method corresponding to the data type

4.3 Serialization mode - Caft Root mapping

So far, both the Identity and Full SSI mode rely exclusively on the Terracotta DSO, sharing every field belonging to a Runnable target and guaranteeing object identity for the entire thread context. However, every field that is shared holds a communication cost, and in some cases we could simply copy the data and read it locally, without need for further synchronization with the master node for the program to work correctly. This assumption is the main motivation to add an extra mode, that relies on plain Java serialization for passing a Runnable target to a worker node.

In this mode, we use ASM to add the Java Serializable interface, along with the Serialization uid if it does not exist already, in order to avoid the scenario where the user or programmer has to manually change the source code to add a new interface that was not needed before. This allows us to instantiate threads on remote workers with Runnable targets that are serialized by RMI. However, to guarantee correctness in some applications, we need to provide a way to preserve object identity between fields of a Runnable target and other fields that remain in the home node. It should be noticed that the original Terracotta Root annotation for fields does not work in this case, as the changes in the Runnable object in the worker node are done in a serialized copy, which is considered by Terracotta as just another different object, and as such the synchronization is not done.

To solve this problem, we introduce a new Java annotation "CaftRoot". This annotation should be applied to all pairs of fields whose identity should be the same across the cluster. Pairs of fields are created by assigning the same string key in the key annotation parameter. We illustrate this concept with a code example below, taken directly from the Sunflow modification applied to Serialization mode. In this example, we show the `BucketThread` class and the `BucketRenderer` class. A bucket is a Sunflow concept corresponding to the set of pixels to be generated by a thread.

```
public class BucketThread implements
    Runnable {
        private int threadID;

        @CaftRoot(key="display")
        private Display display;
        @CaftRoot(key="bucketCounter")
        Integer bucketCounter;
        @CaftRoot(key="bucketCoords")
        int[] bucketCoords;

        ...

}

public class BucketRenderer implements
    ImageSampler {
        @CaftRoot(key="display")
        Display display;
        @CaftRoot(key="bucketCounter")
        Integer bucketCounter;
        @CaftRoot(key="bucketCoords")
        int[] bucketCoords;

        ...
```

In this example, we map the `Display` object corresponding to the shared data structure used by all threads to store the rendering

calculations, as well as the array used to determine the next bucket to render and an `Integer` counter to keep track of the number of buckets processed. When a new thread is instantiated for computing a new bucket, the instance of the `BucketThread` class will be serialized and sent to a worker node. However, the Caft middleware will use the Terracotta DSO to hold the fields annotated with our special `CaftRoot` annotation. This annotation has the same semantics as the original Terracotta `Root` annotation, meaning that any fields annotated with it will be shared among all cluster nodes. This way, the programmer gets the ability to choose pairs of fields to preserve identity, while the remaining fields will be copied and no synchronization will be done between them.

In practice, we implement the access to fields annotated with the `CaftRoot` annotation by having two concurrent hash maps, one that that associates a string composed by the concatenation of the class name plus the field name with the key specified by the user, and another that associates this key with the concrete object instance. Both maps will belong to the Terracotta DSO to be accessible in every node. The table 1 summarizes the mapping that will be done for this example.

Table 1: CaftRoot mapping

Map	Key	Value
fieldToKey	BucketRendererdisplay BucketThreaddisplay	display
	BucketRendererbucketCounter BucketThreadbucketCounter	bucketCounter
	BucketRendererbucketCoords BucketThreadbucketCoords	bucketCoords
clusteredFields	display	Display instance
	bucketCounter	Integer instance
	bucketCoords	int[] instance

4.4 Caft Root Adapter

Considering the mapping done in the previous subsection, all that is left to do is to find a way to fill the "fieldToKey" map, and add instrumentations that intercept field access and get the values from the "clusteredFields" map, leaving the stack in a correct state. The "fieldToKey" map is filled at class load-time, using our custom class loader and the Java reflection API to detect which fields have the mapping put down by the programmer. The field access instrumentations are done by a special method adapter that is applied to every method in the application, which after checking if there is a key for a class and field name pair replaces the coded accesses as follows:

```
//Getfield
ldc key
invokestatic org/terracotta/clusterthread/
    caftroot/CaftRootMap getField(Ljava/lang
    /Object;Ljava/lang/String;)Ljava/lang/
    Object;");
checkcast fieldType
```

For the `getfield` bytecode, it should be reminded that at this point we have the object instance on stack, so we simply generate code that pushes the key and invokes a static method that will consume both arguments and get the value present in the "clusteredFields" map. After this, we also generate a `checkcast` bytecode to ensure that the value put on stack is the same type of the field. The field type information is available via the ASM framework. For the `putfield` bytecode, we also need to push the key onto the stack, so it can be used when our static method that puts the new field value into the "clusteredFields" map. For the static versions, the instrumentations are similar to the ones described for non-static fields.

5. EVALUATION

In this section we are going to describe the methodology used for evaluating the prototype, and its results. We used up to three machines in a cluster, with Intel(R) Core(TM)2 Quad processors (with four cores each) and 8GB of RAM, running Linux Ubuntu 9.04, with Java version 1.6.0_16, Terracotta Open Source edition, version 3.3.0, and three multi-threaded Java applications that have the potential to scale well with multiple processors, taking advantage of the extra resources available in terms of computational power and memory (Fibonacci, Sunflow renderer and Matrix by vector multiplication). We are also concerned with the transparency of our approach, and how much is the impact of our bytecode instrumentations.

5.1 Fibonacci

For testing purposes, we developed a simple application that computes Fibonacci numbers using Binet's Fibonacci number formula. Our application takes the maximum number of Fibonacci to compute, along with the number of threads, and splits the workload by having each thread compute a number of Fibonacci numbers corresponding to the maximum given divided by the number of threads. Concerning the several modes of our middleware, in Full SSI mode we simply edited the `tc-config.xml` file of our middleware to add the classes necessary to be instrumented by Terracotta. For Identity mode, we needed to add some synchronized blocks using the corresponding keyword. For the Serialization mode, we used a `ConcurrentHashMap` shared by all threads and mapped by the `CaftRoot`, in order to store the private arrays of each thread.

5.1.1 Bytecode size

For the bytecode size measurements, we ran the application in our middleware in a single node, running both the master and the worker components, and used our custom Classloader to keep track of the bytecode size of each class, before and after applying our instrumentations in each mode. Since each mode needs different changes in the source code (or none), as described in the previous section, the original bytecode size before applying the bytecode instrumentations will be different, depending on the mode chosen. Also, we have taken all the measurements in the master node, which is the one that will load all the application classes needed. The results are shown in the table below:

Table 2: Fibonacci - Bytecode size

Mode	Original (bytes)	After instr. (bytes)	Overhead ratio
Serialization	7419	8313	1.12
Identity	6275	6702	1.06
Full SSI	6253	8174	1.31

As we can observe, the mode with the largest overhead is the Full SSI mode, followed by the Serialization and Identity. Considering the bytecode instrumentations defined in section 4, this is expected, as the Full SSI mode adds more methods to each class. The Serialization mode shares a `ConcurrentHashMap` as demonstrated in the previous section, so it is expected to have the largest bytecode in the end, while Identity mode gets to have the least impact.

5.1.2 Execution Time

For the execution time measurements, we configured our application to compute the first 1200 numbers of the Fibonacci sequence, with a number of threads directly proportional to the number of threads available. Also, we tested our application using only the Terracotta middleware, to have a general idea of how the usage of the original Terracotta platform impacts the performance. We considered two different scenarios for the tests: **Terracotta Inst. only** and **Terraocotta Inst + Sharing**. The former tested the application with only the Terracotta bytecode instrumentations activated, while the latter also shared the same data structures shared in the Identity and Full SSI modes. Finally, we tested our application in a standard local JVM, for comparison purposes with our distributed solution. The results are presented in Figure 3.

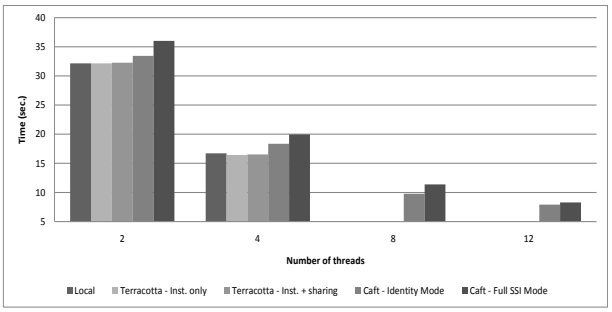

Figure 3: Fibonacci - Execution times for Identity and Full SSI modes

As we can observe in the graph, the overhead introduced by Terracotta is not much, as we only share a relatively small array in each thread for storing the Fibonacci numbers, along with some auxiliary variables. By adding our middleware, we introduce an extra overhead which is not very significant, even when running it in Full SSI mode and as such, it is possible to obtain smaller execution times by adding more nodes to the Terracotta cluster.

For Serialization mode, we made the necessary code changes and measured the execution time in a local JVM, in our middleware, in Terracotta with only the Terracotta instrumentations enabled, in Terracotta with instrumentations and sharing of data equivalent to the one needed for the Serialization mode to work in Caft. The results are presented in Figure 4.

As we can observe in the graph, the overhead introduced by Terracotta is very similar to the one in the previous case, despite the source code being slightly different. The overhead introduced by Serialization mode ends up being larger, but it is still able to achieve lower execution times that a single node in a local JVM.

In this case, the Identity mode scales very well, followed by the Full SSI and as last, Serialization. The Serialization mode ends up sharing almost the same structures as Identity mode, and the more fine grained approach does not compensate. In the next section, we are going to test our middleware with Sunflow, an Open Source Java multi-threaded renderer.

5.2 Sunflow

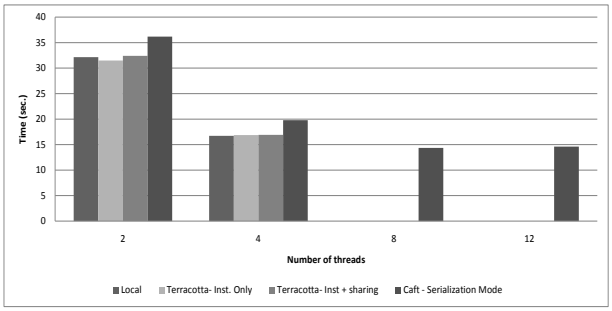

Figure 4: Fibonacci - Execution times for Serialization mode

Sunflow is an Open Source rendering system for photo-realistic image synthesis. It supports rendering of scenes to popular image formats such as PNG, TGA and HDR. The scenes can be specified using Java or a special scene graph language, typical of other similar applications such as POV-Ray.

5.2.1 Source code changes

As with the previous Fibonacci application, in Full SSI mode we simply edited the `tc-config.xml` file of our application to add the classes necessary to be instrumented by Terracotta. For Identity mode, we needed to add some synchronized blocks. For the Serialization mode, the code changes necessary were described in section 4.3 as a pratical example. We measured the bytecode size overhead of the several modes in the table below:

Table 3: Sunflow - Bytecode size

Mode	Original (bytes)	After instr. (bytes)	Overhead ratio
Serialization	400559	408518	1.01
Identity	399819	416429	1.04
Full SSI	405687	542191	1.34

In the Sunflow application case, the quantity of classes that need to be instrumented is larger than in the previous Fibonacci application. As such, the Serialization mode bytecode overhead is less than its counterparts, as the programmer annotates the specific methods that require synchronization directly in the source code. Identity mode will add the `AutolockWrite` annotation to every method, and as such, the original bytecode is slightly smaller than its Serialization counterpart, but after applying the instrumentations, it becomes larger. The Full SSI still remains the mode that generates the largest bytecode, due to the extra methods that need to be added in each class.

5.2.2 Execution time

For the execution time measurements, we configured Sunflow to render one of the example images, with a number of threads directly proportional to the number of threads available. The results are presented in figure 5:

As we can observe in the graph, the Terracotta bytecode instrumentations add a considerable overhead, even when we do not share any data in the DSO. By adding the same data structures that are shared in both Identity and Full SSI modes, the execution times

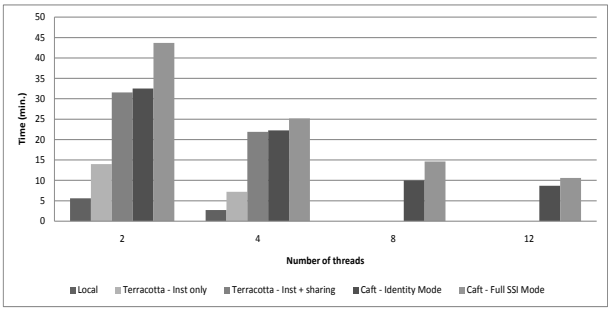

Figure 5: Sunflow - Execution times for Identity and Full SSI modes

of the application in Terracotta for two and four threads are very similar to the ones presented by Caft in Identity mode, for the same number of threads. Our middleware can then obtain better execution times by using the extra processors and obtain scalability compared to Terracotta by itself. The Full SSI mode adds a more significant overhead, having the greatest execution times.

For Serialization mode, we performed the necessary code changes and shared only the data structures necessary for storing the results. For comparison purposes, we measured the execution time in Terracotta with instrumentations enabled, and also with an equivalent sharing of data. The results are presented in Figure 6.

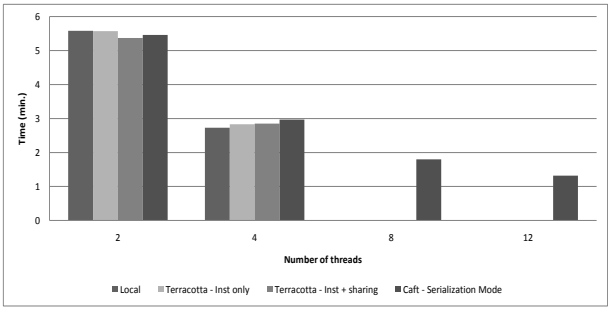

Figure 6: Sunflow - Execution times for Serialization mode

As we can observe in the graph, the overhead introduced by both the Terracotta instrumentations and sharing of data decreased and its execution time is comparable with a standard JVM. With our middleware on top, we are able to achieve better execution times than the ones that are possible with only one node and a standard JVM.

5.3 Matrix-vector multiplication

For testing purposes, we also developed a multi-threaded application that multiplies a matrix by a vector, splitting the matrix rows across the threads. As with the previous applications, in Full SSI mode we simply edited the `tc-config.xml` file of our application to add the classes necessary to be instrumented by Terracotta. For Identity mode, we simply ran the application and hoped that the synchronization present would suffice. For Serialization mode, we added the `CaftRoot` annotation to share the array responsible for storing the result of the multiplication of the matrix rows by the vector.

5.3.1 Bytecode size

As with the previous examples, we ran the application in our middleware in a single node, running both the master and the worker components, and used our custom Classloader to keep track of the bytecode size of each class, before and after applying our instrumentations in each node. The results are shown in the table below:

Table 4: Matrix-vector multiplication - Bytecode size

Mode	Original (bytes)	After instr. (bytes)	Overhead ratio
Serialization	6561	7466	1.13
Identity	6472	6863	1.06
Full SSI	6252	8813	1.41

In this case, the results concerning the bytecode size are similar to the ones obtained by the Fibonacci application. Before we apply the instrumentations, code size is slightly smaller in the Full SSI mode due to the fact that the programmer does not introduce extra synchronization or Java annotations for clustering the application. In Identity mode however, we require that the programmer adds some synchronized blocks, impacting the original size. Code size is further increased in Serialization mode, by also adding Java annotations. After we apply the instrumentations, Full SSI mode generates the largest code, followed by Serialization and Identity mode.

5.3.2 Execution time

For the execution time measurements, we tested our application by multiplying a matrix of 32768 rows by 32768 columns and a vector of 32768 positions. As with previous applications, we ran the matrix by vector multiplication with no more than one thread per processor and measured the time taken by each mode with two, four, eight and twelve processors. We also tested our application in a standard local JVM, for comparison purposes with our distributed solution. The results for Identity and Serialization mode are presented in Figure 7.

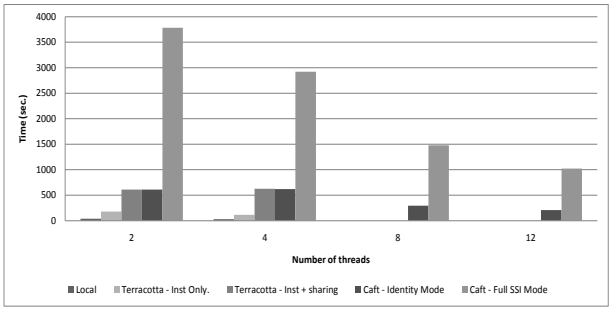

Figure 7: Matrix*vector - Execution times for Identity and Full SSI modes

As we can observe in the graph, the Terracotta bytecode instrumentations add a small overhead, even when we do not share any data in the DSO. By adding the same data structures that are shared in both Identity and Full SSI modes, the execution times of the application in Terracotta for two and four threads are very similar to the ones presented by Caft in Identity mode, for the same number of threads. Our middleware can then obtain better execution times by using the extra processors and obtain scalability compared to Terracotta by itself. The Full SSI mode adds a very significant overhead, having execution times much greater than any of its counterparts,

as every write in an array of results needs to be propagated to the Terracotta Server.

As with the Sunflow application, we still do not obtain scalability when compared to a standard JVM. Considering this, we performed the necessary code changes to run the application in Serialization mode, and sharing only the data structures necessary for storing the results. The results are presented in Figure 8.

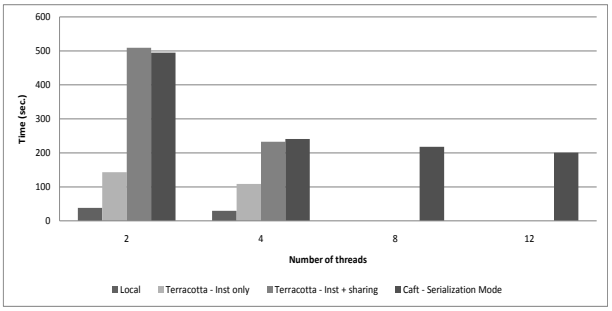

Figure 8: Matrix*vector - Execution times for Serialization mode

As we can observe in the graph, the results are slightly better than the previous modes, but the execution times are still not better than a local JVM. This can be explained by the fact that this application is much more memory-intensive than CPU intensive, spending a more considerable amount of time writing results to array positions instead of computing the matrix, which is pretty trivial in comparison with the previous applications, where we computed Fibonacci numbers or performed ray-tracing calculations. This type of applications should not scale very well in Terracotta, in terms of speed-up, but we believe the extra memory available in the cluster can still give a competitive advantage over a single node. This assumption shall be tested and measured in section 5.3.3.

5.3.3 Memory usage

In this section, we attempted to stress test our middleware from the memory usage perspective, to check if it was possible to take advantage of the extra memory provided by several nodes. For this purpose, we ran the application using three different matrix sizes: 32768x32768, 62556x32768, 53090x53090. The memory occupied by each of them was estimated considering that each `int` value has at least 4 bytes. The heap sized was fixed at 7 GB of data, as the machines were limited to 8 GB of RAM and we wanted to save some space for other JVM objects and other applications running in each node. The results are shown in Figure 9:

In this example, we managed to allocate a matrix of 62556x32768 with two nodes, corresponding to 8 GB of data, while with one node only we would get a `java.lang.OutOfMemoryError`. In a similar way, with three nodes we could allocate about 10.5 GB of data, while with only two nodes we got the same exception. In conclusion, adding more nodes allowed us to perform computations with a matrix in memory split across several machines, which would not be possible if the application was running in a local JVM.

6. CONCLUSION

In this work, we explored a different use case for Terracotta, the running of simple, multi-threaded applications, that were not designed with Terracotta or clustering in mind. We attempted to develop an approach with the best transparency possible that would not require deep changes in the application, and still be powerful enough to achieve good performance.

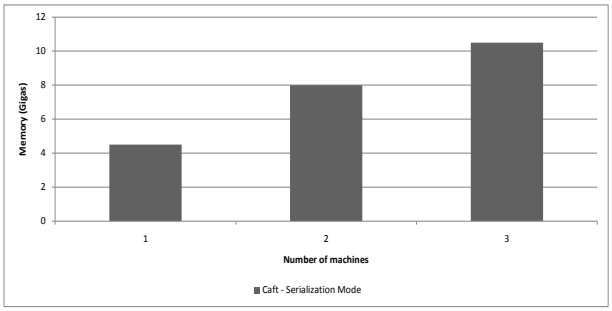

Figure 9: Matrix*vector - Memory Stress

The Identity mode allows the programmer to run multi-threaded applications in a distributed way, using only pure Java and adding synchronization as necessary. The overhead will be very dependent on the thread context itself, concerning the amount of data shared and manipulated.

By implementing a bytecode approach that adds synchronization on field and array access, we concluded that it is possible to improve the Terracotta DSO usage by automatically adding some extra synchronization that is needed for defining transaction boundaries. This mode should be kept optional in our middleware, due to the extra overhead observed in the section 5.

And last, the Serialization approach allows for a more fine-grained sharing of objects on the global heap. This provides a "mixed" semantic that is not very typical of Terracotta, as the most common use case is to use "Ehcache" replication which either serializes every object put on cache or preserves identity. We concluded, in section 5, that this approach can hold very good results in real-world applications such as the Sunflow renderer.

7. REFERENCES

[1] C. Amza, A. L. Cox, S. Dwarkadas, H. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29:18–28, 1996.

[2] J. Andersson, S. Weber, E. Cecchet, C. Jensen, V. Cahill, J. A. Ý, S. W. Ý, E. C. Þ, C. J. Ý, V. C. Ý, and T. College. Kaffemik - a distributed jvm on a single address space architecture, 2001.

[3] G. Antoniu, L. Boug, P. Hatcher, M. MacBeth, K. Mcguigan, and R. Namyst. The hyperion system: Compiling multithreaded java bytecode for distributed execution, 2001.

[4] Y. Aridor, M. Factor, and A. 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.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. 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.

[6] B. Bershad, M. Zekauskas, and W. Sawdon. The midway distributed shared memory system. In *Compcon Spring '93, Digest of Papers.*, pages 528–537, Feb 1993.

[7] G. E. Blelloch, P. B. Gibbons, G. J. Narlikar, and Y. 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.

[8] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[9] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[10] R. Buyya, T. Cortes, and H. Jin. Single system image. *Int. J. High Perform. Comput. Appl.*, 15(2):124–135, 2001.

[11] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. *SIGOPS Oper. Syst. Rev.*, 25(5):152–164, 1991.

[12] P.-C. Chen, C.-I. Lin, S.-W. Huang, J.-B. Chang, C.-K. Shieh, and T.-Y. 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.

[13] W. Z. Cho-Li, C. li Wang, and F. C. M. Lau. Lightweight transparent java thread migration for distributed jvm. In *In International Conference on Parallel Processing*, pages 465–472, 2003.

[14] B. Dimitrov and V. Rego. Arachne: A portable threads system supporting migrant threads on heterogeneous network farms. *IEEE Transactions on Parallel and Distributed Systems*, 9:459–469, 1998.

[15] M. Factor, A. Schuster, and K. 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.

[16] T. 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.

[17] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. 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.

[18] M. Herlihy, J. E. 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.

[19] L. Iftode, C. Dubnicki, E. W. Felten, and K. 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.

[20] L. Iftode, J. P. Singh, and K. 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.

[21] F. Informatik, T. H. Darmstadt, T. Kunz, and T. Kunz. The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering*, 17(17):725–730, 1991.

[22] P. Keleher, A. L. Cox, and W. 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.

[23] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programm. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[24] K. Li. Ivy: a shared virtual memory system for parallel computing. pages 94–101, Aug. 1988.

[25] M. J. M. Ma, C.-L. Wang, and F. C. M. Lau. Jessica: Java-enabled single-system-image computing architecture. *J. Parallel Distrib. Comput.*, 60(10):1194–1222, 2000.

[26] G. 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.

[27] L. Parziale, E. M. Dow, K. Egeler, J. J. Herne, C. Jordan, E. L. Alves, E. P. Naveen, M. S. Pattabhiraman, and K. Smith. *Introduction to the new mainframe: z/vm basics*. IBM Corp., Riverton, NJ, USA, 2007.

[28] J. Protić, M. Tomašević, and V. Milutinović. *Distributed Shared Memory: Concepts and Systems*. John Wiley & Sons, 1998.

[29] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *Computer*, 25(12):33–44, 1992.

[30] B. H. Sirac, S. Bouchenak, and D. Hagimont. Approaches to capturing java threads state. In *In Middleware 2000*, 2000.

[31] E. Speight and J. K. Bennett. Brazos: A third generation dsm system. In *IN PROCEEDINGS OF THE 1ST USENIX WINDOWS NT SYMPOSIUM*, pages 95–106, 1997.

[32] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A bytecode translator for distributed execution of "legacy" java software. pages 236–255. Springer-Verlag, 2001.

[33] Terracotta. A technical introduction to terracotta. 2008.

[34] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic java application partitioning. pages 178–204. Springer-Verlag, 2002.

[35] R. Veldema, R. Bhoedjang, and H. 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.

[36] W. YU and A. COX. Java/dsm: A platform for heterogeneous computing. 1997.

[37] M. Zenger. Javaparty - transparent remote objects in java, 1997.

[38] W. Zhu, C.-L. Wang, and F. C. M. Lau. Jessica2: A distributed java virtual machine with transparent thread migration support. *Cluster Computing, IEEE International Conference on*, 0:381, 2002.