A^2-VM : A Cooperative Java VM with Support for Resource-Awareness and Cluster-Wide Thread Scheduling*

José Simão[2,1], João Lemos[1], and Luís Veiga[1]

[1] Instituto Superior Técnico, Technical University of Lisbon / INESC-ID Lisboa
[2] Instituto Superior de Engenharia de Lisboa

Abstract. In today's scenarios of large scale computing and service providing, the deployment of distributed infrastructures, namely computer clusters, is a very active research area. In recent years, the use of Grids, Utility and Cloud Computing, shows that these are approaches with growing interest and applicability, as well as scientific and also commercial impact.
This work presents the design and implementation issues of a cooperative VM for a distributed execution environment that is resource-aware and policy-driven. Nodes cooperate to achieve efficient management of the available local and global resources. We propose A^2-VM , a cooperative cluster-enabled virtual execution environment for Java, to be deployed on Grid sites and Cloud data-centers that usually comprise a number of federated clusters. This cooperative VM has the ability to monitor base mechanisms (e.g. thread scheduling, garbage collection, memory or network consumptions) to assess application's performance and reconfigure these mechanisms in run-time according to previously defined resource allocation policies.
We have designed this new cluster runtime by extending the Jikes Research Virtual Machine to incorporate resource awareness (namely resource consumption and restrictions), and extending the TerraCotta DSO with a distributed thread scheduling mechanism driven by policies that take into account resource utilization. In this paper we also discuss the cost of activating such mechanisms, focusing on the overhead of measuring/metering resource usage and performing policy evaluation.

1 Introduction

In today's scenarios of large scale computing and service providing, the deployment of distributed infrastructures, namely computer clusters, is a very active research area. In recent years, the use of Grids, Utility and Cloud Computing, shows that these are approaches with growing interest and applicability, as well as scientific and commercial impact.

Managed languages (e.g., Java, C#) are becoming increasingly relevant in the development of large scale solutions, leveraging the benefits of a virtual

* This work is available at http://link.springer.com

execution environment (VEE) to provide secure, manageable and componentized solutions. Relevant examples include work done in various areas such as web application hosting, data processing, enterprise services, supply-chain platforms, implementation of functionality in service-oriented architectures, and even in e-Science fields (e.g., with more and more usage of Java in the context of physics simulation, economics/statistics, network simulation, chemistry, computational biology and bio-informatics [16, 15, 18], there being already many available Java-based APIs such as Neobio).[3]

To extend the benefits of a local VEE, while allowing scale-out regarding performance and memory requirements, many solutions have been proposed to federate Java virtual machines [24, 2, 25], aiming to provide a single system image where the managed application can benefit from the global resources of the cluster.

A VEE cluster-enabled environment can execute applications with very different resource requirements. This leads to the use of selected algorithms for runtime and system services, aiming to maximize the performance of the applications running on the cluster. However, for other applications, for example the ones owned by restricted users, it can be necessary to impose limits on their resource consumption. These two non functional requirements can only be fulfilled if the cluster can monitor and control the resources it uses both at the VEE and distributed level, and whether the several local VEE, each running on its node, are able to cooperate to manage resources overall.

Existing approaches to resource-awareness in VMs, cluster-enabled runtimes, and adaptability are still not adequate for this intended scenario (more details in Section 5) as they have not been combined into a single infrastructure for unmodified applications. Existing resource-aware VMs do not target popular platforms, and cluster-enabled runtimes have support neither for global thread scheduling nor for checkpointing/restore mechanisms. Furthermore, lower-level mechanisms and VM parameters cannot be governed by declarative policies.

In this paper, we report on the design and implementation of A^2-*VM* (*Autonomous and Adaptive Virtual Machine*), a distributed execution environment where nodes cooperate to make an efficient management of the available local and global resources. We propose a cluster-enabled VEE with the ability to monitor base mechanisms (e.g. thread scheduling, garbage collection, memory or network consumptions) at different nodes in order to assess an application's performance and resource usage, and reconfigure these mechanisms in run-time, in one or more nodes, according to previously defined resource allocation policies (or *quality-of-execution* specifications) . These policies regulate how resources should be used by the application in the cluster, leading to the adaptation of components at different levels of the cluster in order to enforce it.

We propose a layered approach to resource monitoring, management and restriction enforcement. While restrictions to resources are effectively enforced at the level of each of the individual cooperating VMs, overall performance

[3] http://www.bioinformatics.org/neobio/

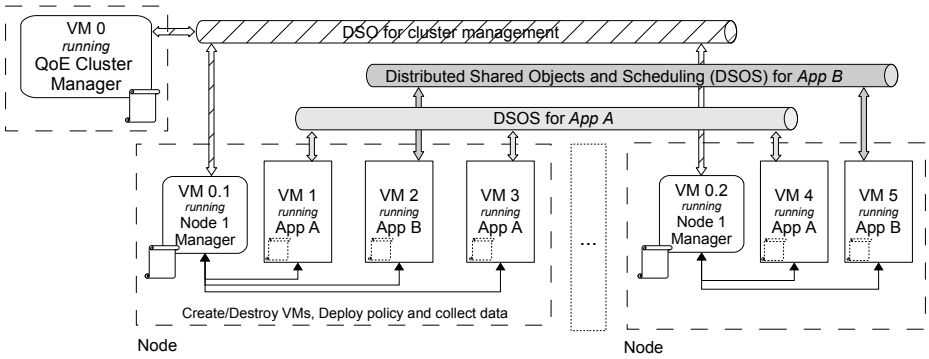


Fig. 1: Architecture of the A^2-VM

assessment and policy-driven resource management are carried out by node and cluster-wise manager components.

The rest of this paper is organized as follows. Section 2 describes the overall architecture of our proposal for a resource-aware and policy-driven cooperative VM, introducing the main components of A^2-VM , and details on the specific internal mechanisms and functionality offered. Section 3 describes the main aspects of the current development and implementation of A^2-VM . In Section 4, we assess and evaluate A^2-VM , regarding: i) the impact of resource awareness and policy support (measuring overhead of resource monitoring and policy engine performance), and ii) the cluster-wide cooperative thread scheduling offered (overheads and performance improvements). In Section 5, we discuss our research in light of other systems described in the literature, framing them with our contribution. Section 6 closes the paper with conclusions and future work.

2 Architecture

The overall architecture of A^2-VM (*Autonomous and Adaptive Virtual Machine*) is presented in Figure 1. We consider a *cluster* as a typical aggregation of a number of *nodes* which are usually machines with one or more multi-core CPUs, with several GB of RAM, interconnected by a regular LAN network link (100 Mbit, 1 Gbit transfer rate). We assume there may be several applications, possibly from different users, running on the cluster at a given time, i.e., the cluster is not necessarily dedicated to a single application. The cluster has one top-level coordinator, the *QoE Manager* that monitors the *Quality-of-Execution* of applications.[4]

[4] We opt for this notion instead of hard *service-level agreements* usually employed in commercial application hosting scenarios, because we intend to target serval types of applications in shared infrastructures, without necessarily strict contractual requirements.

Each *node* is capable of executing several instances of a Java VM, with each VM holding part of the data and executing part of the threads of an application. As these VMs may compete for the resources of the underlying cluster node, there must be a *node manager* in each node, in charge of VM deployment, lifecycle management, resource monitoring and resource management/restriction. Finally, in order for the node and cluster manager to be able to obtain monitoring data and get their policies and decisions carried out, the Java VMs must be resource-aware, essentially, report on resource usage and enforce limits on resource consumption. Cooperation among VMs is carried out via the *QoE Manager*, that receives information regarding resource consumption in each VM, by each application, and instructs VMs to allow or restrict further resource usage.

In summary, the responsibilities of each of these entities are the following:

- Cluster QoE Manager
  - collect global data of cluster applications (i.e. partitioned across VMs and nodes)
  - deploy/regulate nodes based on user's QoE
- Node QoE Manager
  - report information about node load
  - deploy new policies on VMs
  - create or destroy new instances
  - collect VM's resource usage data
- (resource-aware) VM
  - enforce resource usage limits
  - give internal information about resource usage

A^2-*VM* is thus comprised of several components, or building blocks. Each one gives a contribution to support applications with a global distributed image of a virtual machine runtime, where resource consumption and allocation is driven by a high-level policies, system-wide, application or user related. From a bottom-up point of view, the first building block above the operating system in each node is a process-level managed language virtual machine, enhanced with mechanisms and services that are not available in regular VMs. These include the accounting of resource consumption.

The second building block aggregates individual VMs, as the ones mentioned above, to form within the cluster a distributed shared object space assigned to a specific application. It gives running applications support for single system image semantics, across the cluster, with regard to the object address space. Techniques like bytecode enhancement/instrumentation or rewriting must be used, so that unmodified applications can operate in a partitioned global address space, where some objects exist only as local copies and others are shared in a global heap.

The third building block turns A^2-*VM* into a cluster-aware cooperative virtual machine. This abstraction layer is responsible for the global thread scheduling in the cluster, starting new work items in local or remote nodes, depending on a cluster wide policy and the assessment of available resources. This layer is

the A^2-VM boundary that the cluster-enabled applications interface with (note that for the applications, the cluster looks like a single, yet much *larger*, virtual machine). Similarly to the previous block, application classes are further instrumented/enhanced (although the two sets of instrumentation can be applied in a single phase), in order to guarantee correct behavior in the cluster. Finally, it exposes the underlying mechanisms to the adaptability policy engine, and accepts external commands that will regulate how the VM's internal mechanisms should behave.

The resource-aware VM, the distributed shared object layer, and the cluster level scheduler are all sources of relevant monitoring information to the policy engine of A^2-VM . This data can be used as input to declarative policies in order to determine a certain rule outcome, i.e. what action to perform when a resource is exhausted or has reached its limit, regarding a user or application. The other purpose of collecting this data is to infer a profile for a given application. Such profiles will result in the automatic use of policies for a certain group of applications, aiming to improve their performance. The effects, positive or negative, of applying such policies are then used to confirm, or reject, the level of correlation between the profile and the applications.

Currently, thread scheduling tries first to collocate threads of the sample application to the same VM, until the specified CPU load, wait time, and memory usage thresholds are reached. After that, subsequent threads are allocated elsewhere within the same node or across the cluster, balancing the load. Application performance is monitored by a combination of black-box and grey-box approaches. Black-box consists in the monitoring of the parameters mentioned above that allows us to determine, roughly, whether an application is experiencing poor performance due to resource shortage or contention. Grey-box approach consists in monitoring advancement of file cursors (for sequential reading and writing), and data transferred in order to estimate current progress against previous executions of the same application.

On top of this distributed runtime are the applications, consuming resources on each node and using the services provided by the resource-aware VM that is executing on each one. A^2-VM targets mainly applications with a long execution time and that may spawn several threads to parallelize their work, as usual in e-Science fields such as those mentioned before.

The following sections will describe the depicted architecture, explaining the specific contributions of each component.

2.1 Resource Awareness and Control

The Resource Aware virtual machine is the underlying component of the proposed infrastructure. It has two main characteristics: *i)* resource usage monitoring, and *ii)* resource usage restriction or limitation. Furthermore, there are checkpointing, restore and migration mechanisms, used for more coarse-grained load-balancing across the cluster, that are out of the scope of this paper [14].

Current virtual machines (VM) for managed languages can report about several aspects of their internal components, like used memory, number of threads,

classes loaded [20, 19]. However they do not enforce limits on the resources consumed by their single node applications. In a cluster of collaborating virtual machines, because there is a limited amount of resources to be shared among several instances, some resources must be constrained in favor of an application or group of applications.

Extending a managed language VM to be aware of the existing resources must be done without compromising the usability (mainly portability) of application code. The VM must continue to run existent applications as they are. This component is an extended Java virtual machine with the capacity to extract high and low level VM parameters, e.g., heap memory, network and system threads usage. Along with the capacity to obtain these parameters, they can also be constrained to reflect a cluster policy. The monitoring system is extensible in the number and type of resources to consider.

2.2 Cluster-Wide Cooperative VM

To enable effective distribution of load among different nodes of the cluster, our system relies on a cluster level load balancer capable of spawning new threads (or work tasks) on any cluster node based on a cluster wide policy. When an application asks for a new thread to be created (e.g., by invoking the `start` method on a `Thread` object), the request can be either denied or granted based on the resource allocation decided for the cluster. If it is granted, the load balancer will create the new thread in the most appropriate node to fulfill the cluster policy. For example, if the application has a high priority order compared to other applications of the cluster, then the thread could be created in a lesser loaded node (preferably, one with a VM already assigned to the application's DSO; if needed and allowed, a new VM on any lesser loaded node) . The decision on what node new threads are created is left to the policy engine to decide with current information. Nevertheless, the resource-aware VM has an important role in this process by making it possible to impose a hard limit on resources, e.g., the number of running threads of the application at a specific node, or globally.

A^2-VM aims to accommodate applications developed by users with different levels of expertise regarding the development of multi-threaded applications and cluster architectures, giving a performance versus transparency trade-off. To this end, the thread scheduler has two operation modes: i) Identity and ii) Full SSI. **Identity** mode should be used if we have the byte-code of a multi-threaded Java application that is *explicitly* synchronized (e.g., using Java monitors), or there is access to the source code and synchronization code can be added with ease. **Full SSI** mode should be used if we have the byte-code of a multi-threaded Java application that is not explicitly synchronized (mainly applications comprised of cooperating threads that make use of *volatile* object fields that the Java VM specification assures to be updated in a single memory write operation, while Terracotta does not honour this) and there is no access to the source code. For instance, in DaCapo 2009 benchmarks, 6 out of 14 applications do indeed use such *volatile* fields, and rely on the VM to uphold this semantics, hence the

```
<?xml version="1.0" encoding="UTF-8"?>
<RAMConfiguration>
  <ResourceAttributes name="NumberOfThreads" initalLimit="15" />
  <ResourceAttributes name="CpuUsage" initalLimit="75%" />
  ...
  <Rule target="NumberOfThreads">
    <!-- Determines how accumulation is done -->
    <OnConsume> <Counter/> </OnConsume>
    <!-- Determines what happens if limit is reached -->
    <OnLimit> <ResourceException/> </OnLimit>
    <!-- Determines what happens if consumption is successful -->
    <OnAfterComsumption>
      <UseCluster threshold="AllCpus"/>
    </OnAfterComsumption>
  </Rule>
  <Rule target="CpuUsage">
    <OnConsume> <HistoryAverage window="5"/> </OnConsume>
    <OnLimit> <Suspend miliseconds="500"/> </OnLimit>
  </Rule>
  ...
</RAMConfiguration>
```

Fig. 2: Declarative policy

relevance of the Full SSI mode to ensure compatibility, transparency and correct functionality when deploying such applications with A^2-VM on a cluster.

2.3 Adaptability and the Policy Engine

The policy engine is responsible for loading and enforcing the policies provided by administrators and possibly users regarding resource management. It achieves this by, globally, sending the necessary commands to the resource-aware VMs in order for them to modify some runtime parameters, or the type of algorithm used to accomplish a cluster related task, as well as instructing them to spawn threads or activate checkpointing/restore and migration mechanisms. A special focus of this component of A^2-VM is also on the improvement of applications' performance, and what can be adapted in the underlying resource-aware VMs in order to achieve it.

It operates autonomously or in reaction to a given resource outage in the VMs. Autonomous behavior is governed by maintaining knowledge about the applications' previous execution, and adjusting the VMs and cluster parameters to achieve better performance for that specific application. Reactive operation is driven by declarative policies that determine the response to a resource outage. This response may result in a local adaptation (e.g. restrain the resources of another VM in the same node, or change the GC algorithm to consume less memory but eventually taking more time to execute) or have cluster wide impact (e.g. migrate the entire application to a VM in another node).

Figure 2 presents a declarative policy to be used by VM instances represented in Figure 1 (i.e. $VM_{1..5}$). It defines limits for CPU usage, and the number of threads and sockets the application is allowed to use. CPU usage and threads

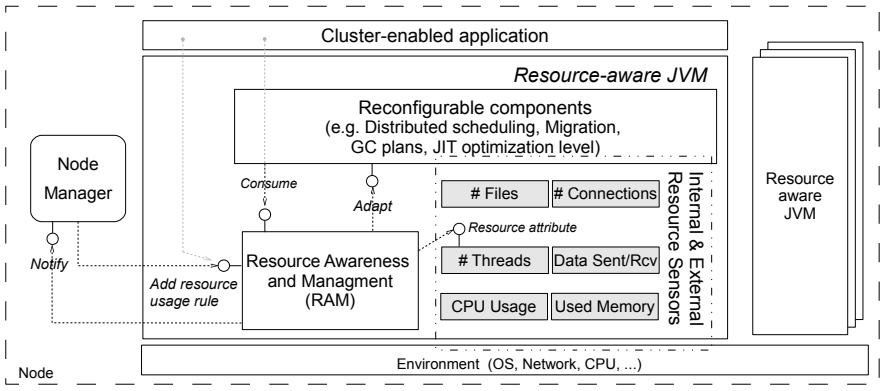


Fig. 3: Resource-aware JVM

are monitored and managed by specific rules but using a similar, reusable approach: i) CPU usage is monitored with a sliding window in order to filter irrelevant peaks, while ii) the number of active threads is also monitored with a sliding window in order to trigger rescheduling only when the limit is consistently exceeded.

3 Implementation

Some of the building blocks of A^2-*VM* 's architecture are partially available in the research community but do not operate in an ensemble. Nevertheless, although some essential functionalities needed in our architecture are missing, the available components constitute a good starting point we can leverage and extend with our own work.

Our first implementation effort is centered on developing a managed language virtual machine with the capacity to monitor and restraint the use of resources based on a dynamic policy, defined declaratively outside the VM. Some work has been done in the past aiming to introduce resource-awareness in such high level virtual machines (details in Section 5). Nevertheless, to the best of our knowledge, none of them is publicly available or usable with popular software, operating systems and hardware architectures. Based on this observation, we have chosen to extend the Jikes RVM [1] to be resource-aware. Thus, in the next subsection we will describe different aspects of our current work on Jikes RVM. Later on, we describe the main implementation aspects of our system regarding the spawning and scheduling of threads in other nodes.

3.1 Extending Jikes RVM with resource-awareness

Figure 3 depicts further details on the architecture of the resource-aware VM we developed for A^2-*VM* . The resource-ware VM has a specific module for

each type of manageable resource (e.g., files, threads, CPU usage, connections, bandwidth, and memory). Each of the module exports to the RAM (Resource Awareness Management) module an *attribute* that abstracts the specifics of the resource. This way, when the RAM decides to limit, reduce or block the usage of a resource by the application, it can instruct the respective attribute without worrying about the details of applying limitation to that specific resource (e.g., disallowing file open, or take a thread out of scheduling). The RAM consumes profile information from the main VM and A^2-*VM* mechanisms (GC and JIT level, and distributed scheduling and migration, respectively). These mechanisms can be adapted and reconfigured by command of the RAM.

Being RAM the *engine* that enables awareness and adaptation, all its decisions are carried out according to the evaluation of rules in the policies loaded by the node manager. The node manager is also notified by the RAM, in each VM, about the application's performance and outcome of RAM's decisions.

The management of a given resource implies the capacity to monitor its current state and to be able to directly or indirectly control its use and usage. The resources that can be monitored in a virtual machine can be either specific of the runtime (e.g. number of threads, number of objects, amount of memory) or be strongly dependent in the operating system (e.g. CPU usage). To unify the management of such disparate types of resources, we carried out the implementation of JSR 284 - The Resource Management API [13] in the context of Jikes RVM, previously not implemented in the context of any widely usable virtual machine.

The relevant elements to resource management as prescribed by JSR 284 are: *resources*, *consumers* and *resource management policies*. Resources are represented by their attributes. For example resources can be classified as *Bounded* or *Unbounded*. *Unbounded* resources are those that have no intrinsic limit (or if it exists, it is large enough to be essentially ignored) on the consumption of the resource (e.g. number of threads). The limits on the consumption of unbounded resources are only those imposed by application-level resource usage policies. Resources can also be *Bounded* if it is possible to reserve a priori a given number of units of a resource to an application. A *Consumer* represents an executing entity which can be a thread or the whole VM. Each consumer is bound to a resource through a *Resource Domain*. *Resource domains* impose a common resource management policy to all *consumers* registered. This policy is programmable through callback functions to the executing application. Although *consumers* can be bound to different *Resource Domains*, they cannot be associated to the same *resource* through different *Domains*. When a resource is about to be consumed, the resource-aware VM, implementing JSR 284, delegates this decision, via a callback, that can be handled by RAM, and either allowed, delayed or denied (with an exception thrown).

Figure 4 shows a notification, `ThreadsCreationNode`, which can be used to configure an A^2-*VM* instance. This callback would be called on each local thread allocation (in Jikes RVM, Java threads are backed by a native class, `RVMThread` that is shown, and that interacts with the host OS threads). It determines that

```
public class ThreadsCreationNode implements Notification {
  long _threshold;
  public ThreadsCreationNode(long threshold) { _threshold = threshold; }
  public void postConsume(
    ResourceDomain domain,
    long previousUsage, long currentUsage) {
    if (currentUsage >= _threshold)
      Scheduler.getInstance().changeAllocationToCluster();
} }
```

Fig. 4: A sample notification handling to change thread allocation to the cluster

if the number of threads created in the local node reaches a certain threshold new threads will be created elsewhere in the cluster. The exact node where they will be placed is left to be determined by the distributed scheduler own policy.

```
public class HistoryAverage implements Constraint {
  ...
  long[] _samplesHistory;
  public HistoryAverage(int wndSize, long maxConsumption) { ... }
  public long preConsume(ResourceDomain domain,
         long currentUsage, long proposedUsage) {
    long average = 0;
    if (_nSamples == _samplesHistory.length) {
      average = _currentSum / _nSamples;
      _currentSum -= _samplesHistory[_idx];
    }
    else { _nSamples   += 1; }
    _currentSum += proposedUsage;
    _samplesHistory[_idx] = proposedUsage;
    _idx = (_idx + 1) % _samplesHistory.length;
    return average > _maxConsumption ? 0 : proposedUsage;
} }
```

Fig. 5: Regulate consumption based on past *wndSize* observations

Figure 5 shows a *constraint*, `HistoryAverage`, which can be used to regulate a CPU usage policy. Consider a scenario where the running application cannot use the CPU above a threshold for a given time window, because the remaining CPU available is reserved for another application (e.g., as part of the quality-of-execution awarded to it). In this case, when the CPU usage monitor evaluates this rule, it would suspend all threads (i.e. return 0 for the allowed usage) if the intended usage is above the average of the last `wndSize` observations. A practical case would be to suspend the application if the CPU usage is above 75% for more than 5 observations.

3.2 Cluster-Wide Cooperative Thread scheduling

In A^2-VM , to achieve distributed thread scheduling, we need to be able to spawn threads in a node different from where the thread's start method is invoked.

Our mechanism to distribute threads among the cluster is built by leveraging and extending the Terracotta [7] Distributed Shared Objects. This middleware uses the client/server terminology and calls the application JVMs that are clustered together Terracotta *clients* or *Terracotta cluster nodes*. These clients run the same application code in each JVM and are clustered together by injecting cluster-aware bytecode into the application Java code at runtime, as the classes are loaded by each JVM. This bytecode injection mechanism is what makes Terracotta transparent to the application. Part of the cluster-aware bytecode injected causes each JVM to connect to the Terracotta server instances. In a cluster, a Terracotta server instance handles the storage and retrieval of object data in the shared clustered virtual heap. The server instance can also store this heap data on disk, making it persistent just as if it were part of a database. Multiple terracotta server instances can exist as a cohesive array.

In a single JVM, objects in the heap are addressed through references. In the Terracotta clustered virtual heap objects are addressed in a similar way, through references to clustered objects which we refer to as distributed shared objects or managed objects in the Terracotta cluster. To the application, these objects are just like regular objects on the heap of the local JVMs, the Terracotta clients. However Terracotta knows that clustered objects need to be handled differently than regular objects. When changes are made to a clustered object, Terracotta keeps track of those changes and sends them to all Terracotta server instances. Server instances, in turn, make sure those changes are visible to all the other JVMs in the cluster as necessary. This way, clustered objects are always up-to-date whenever they are accessed, just as they are in a single JVM. Consistency is assured by using the synchronization present in the Java application (with monitors), which turns into Terracotta transaction boundaries. Piggybacked on these operations, Terracotta injects code to update and fetch data from remote nodes at the beginning and end of these transactions.

Therefore we need to perform additional byte-code enhancement on application classes as a previous step to the byte-code enhancing performed by the Terracotta cluster middleware before applications are run. To do this we used the ASM framework [8]. Creation of threads in remote nodes is a result of invoking JSR 284 in order to attempt to consume a thread resource at that node. The most intricate aspects deal with the issue of enforcing thread transparency (regarding its actual running node) and identity across the cluster, as we explain next.

The instrumentation replaces Java type opcodes that have the Java Thread type as argument with equal opcodes with our custom type ClusterThread. 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 aliased the final methods and replaced Thread method calls with the aliased method. For example, if we have an `invokevirtual` op-

code that invokes the final "join" method of the Thread class, we invoke the "clusterJoin" method instead.

In Identity mode, the instrumentation process adds the Terracotta `Autolock-Write` annotation, in order to take advantage of the local synchronization (Java monitors) to add a Terracotta transaction in every method. In Full SSI mode, we apply 'getters and setters' instrumentation, in order to add synchronization at its lowest level, on field access and array writes. We transform individual get and set operations into invocations to synchronized methods, automatically generated, that perform the equivalent (now synchronized) get and set operation. Therefore, 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.

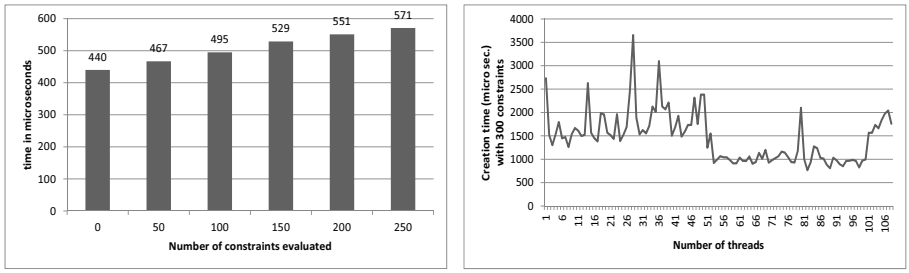
To use the getters and setters generated, we developed a method adapter that replaces direct field accesses with method calls. As such, the method adapter replaces the `getfield` and `putfield` instructions with `invokevirtual` instructions that will invoke the generated corresponding getters and setters. 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 in shared object space. 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 Evaluation

In this section we are going to describe the methodology used for evaluating the A^2-*VM* 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. Each machine was running Linux Ubuntu 9.04, with Java version 1.6.0_16 and Jikes RVM base code version 3.1.1, Terracotta Open Source edition, version 3.3.0, and 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.

4.1 Policy Evaluation and Resource Monitoring

The first part of our performance evaluation regards the resource-aware VM and its impact on rules' evaluation during regular VM operations.Therefore we



(a) Thread creation time with increasing number of constraints to evaluate

(b) Thread creation time during execution with 200 constraints (GC spikes omitted)

Fig. 6: Policy evaluation cost

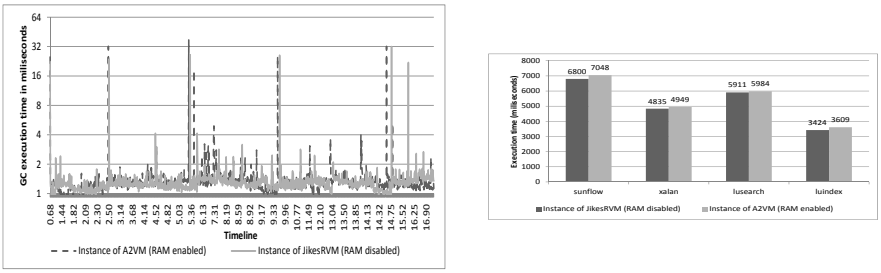
conducted a series of tests, measuring different aspects of a running application, starting from: i) the overhead introduced in the consumption of a specific resource, to ii) the overhead of our JSR 284 implementation, and to iii) policy evaluation in a complete benchmark scenario. All these evaluations are made locally in a single modified Jikes RVM (version 3.1.1), compiled with the `production` profile[5].

In Figure 6.a we can observe the evolution of the overhead introduced to thread creation, by measuring average thread creation and start time, as the policy engine has increasingly larger numbers of rules to evaluate, up to 250 (simulating a highly complex policy). The graph shows that this overhead, while increasing, does not hinder scalability as it is very small, ranging around 500 microseconds.

In Figure 6.b we evaluate whether resource monitoring and policy evaluation (with 200 constraints) introduce any kind of performance degradation as more and more threads are created, resources consumed. Figure 6.b clearly shows (omitting Garbage Collection spikes) that thread creation time does not degrade during application execution, being around 1 millisecond; although subject to some variation, it presents no lasting degradation.

The previous results were obtained monitoring only a single resource, i.e. number of application threads. For other counted resources, e.g. number of bytes sent and received, similar results are expected. Although the allocation of new objects can also be seen as a counted resource, e.g. number of bytes allocated in heap, it is more efficient to evaluate it differently. The cost of checking for constraints regarding object allocation was thus transferred to the garbage collection process, leaving the very frequent allocation operation free of additional verifications.

[5] it includes a two-generation garbage collector [6] and the optimized and adaptive compilation system.



(a) GC execution time during Dacapo's LuSearch benchmarck

(b) Four Dacapo's multi threaded benchmarks with RAM enabled and disabled

Fig. 7: Macro evaluation of an instance of A^2-*VM*

Figure 7.a presents the duration of each GC cycle during the execution of DaCapo's benchmark [5] [6]. `lusearch`, with and without evaluating constraints on heap consumption (i.e. RAM enabled and disabled). The `lusearch` benchmark was configured with a `small` data set, one thread for each available processor (i.e. four threads) and the convergence option active, resulting in some extra warm up runs before the final evaluation. Because of the generational garbage collection algorithm used in our modified Jikes RVM, we can observe many small collection cycles, interleaved with some full heap transversal and defragmentation operations. The two runs share approximately the same average execution time and a similar average deviation: $1.38 \pm 0.31ms$ and $1.38 \pm 0.27ms$, where the former value is when the RAM module is enabled and the last when RAM is disabled. With these results we conclude that performance of object allocation and garbage collection is not diminished with the extra work introduced.

To conclude the evaluation of the RAM module, we stressed an instance of our resource-aware VM with four macro benchmarks, as presented in Figure 7.b. These four benchmarks are multi-threaded applications, which allows us to do a macro evaluation of the proposed modifications. During the execution of these benchmarks there were three resources being monitored (and eventually constrained): the number of threads, the total allocated memory and the CPU usage. The constraints used in evaluation did not restrain the usage of resources so that the benchmarks could properly assess the impact of monitoring different resources simultaneously in real applications (as opposed to the specific benchmarks presented previously in Figures 6 and 6. The results show only a negligible overhead: 3% in average.

[6] The version 9.12 used in the evaluation of A^2-*VM* 's RAM module is available at http://www.dacapobench.org/

4.2 Cooperative Scheduling

For micro-benchmarking purposes, we developed two sample applications (Fibonacci, Matrix by vector multiplication). The Fibonacci application computes Fibonacci numbers using Binet's Fibonacci number formula. It 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. 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 (this is the price to pay for the memory scalability and elasticity it provides). 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 8 (note that results for 2 and 4 threads refer to execution on a single quad-core node).

As we can observe in the graph, the overhead introduced 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.

We also developed a multi-threaded application that multiplies a matrix by a vector, splitting the matrix rows across the threads. 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 Full SSI mode are presented in Figure 9. Recall that distributed scheduling is only used for threads above 4; and that local execution without Terracotta (although not scalable w.r.t. memory and CPU) naturally beats local execution with Terracotta instrumentation in the limited scenario of a single-node.

As we can observe in the graph, the Terracotta bytecode instrumentations adds 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 Identity mode, for the same number of threads. Therefore, we can obtain better execution times by using the extra processors. The Full SSI mode adds a very significant overhead (albeit

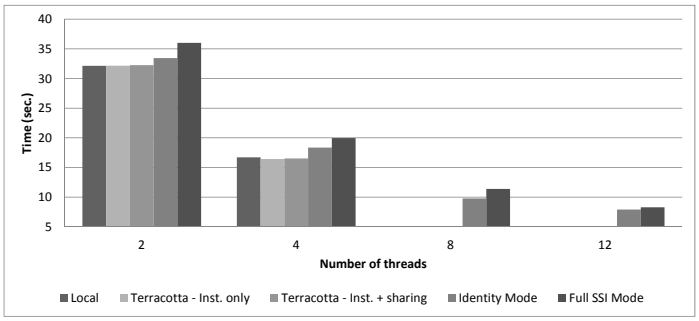


Fig. 8: Fibonacci - Execution times

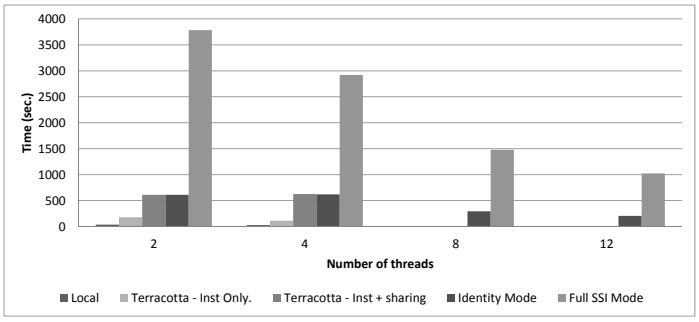


Fig. 9: Matrix*Vector - Execution times

only necessary for applications that are not explicitly synchronized, probably a minority), 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.

5 Related Work

Monitoring low level aspects of a computer system regarding the execution of a given application must be done with low impact in the overall application's performance. Sweeney et al. [23] aims to accomplish these goals using hardware performance counters. They extended the Jikes RVM with a performance monitor layer, interacting with a native C library. Although relevant, in fact, they do not support any kind of restriction on resource consumption. Regarding implementation, they rely on a previous version of Jikes RVM and its N-M thread mapping (where there were N VM threads mapped by the runtime to M native threads), while the current version already uses a 1-1 mapping to native threads.

For runtime mechanisms, Price et al. [21] describes a method for modifying the garbage collector to measure the amount of live memory reachable from each group of threads. Their implementation is also based on an older version of Jikes

RVM but the algorithms proposed could be applied to our system and further extended (i.e. the work presented in [21] does not support tracing collectors). They give some usage scenarios for the information accounted, but leave as an open issue the building of a policy driven framework.

Some system exchange low level precision and additional overhead for the sake of portability. Binder's profiling framework [4] statically instruments the core runtime libraries, and dynamically instruments the rest of the code. The instrumented code periodically calls pure Java agents to process and collect profiling information.

Some high level virtual machines have been augmented or designed from scratch to integrate resource accounting [12, 22, 3, 11]. MVM [11] is based on the Hotspot virtual machine. It supports isolated computations, akin to address spaces, to be made in the same instance of the VM. This abstraction is called *isolate*. Another distinguishing characteristic is the capacity to impose constraints regarding consumption of *isolates*. MVM resource management work is related to the Java Specification Request 284 [13]. Our work builds on this JSR and uses a widely accessible VM. MVM only runs on Solaris on top of SPARC's hardware. The work in [22] and [3] enables precise memory and CPU accounting. Nevertheless they do not provide an integrated interface to determine the resource consumption policy, which may involve VM, system or class library resources.

Cluster-aware high language virtual machines have been a topic of interest for some time. They generically address three main problems: the resource monitoring problem, the migration of workload and a global address reference space. The architecture presented in [10] federates the multi-task virtual machine [11], forming a cluster where there are local and global resources that can be monitored and constrained. However, Czajkowski's work lacks the capacity to relocate workload across the cluster. Regarding policies, their's are only defined programmatically and cannot be changed without recompiling the programs/libraries responsible by clustering mechanisms (e.g. load balancer).

The Jessica VM thread migration schemes have recently been improved to take into account the dependency between threads [17]. To preserve locality of objects, a stack-based mechanism is proposed to profile the set of objects which are tightly coupled with a migrant thread. The mechanisms and algorithms presented in this work can be explored in our system to determine the node where to spawn new threads. Moreover, by leveraging the support for a distributed shared object space in A^2-*VM* , thread migration needs not know in advance, with so fine-grained detail, which objects are more tightly coupled with a thread, as they can be fetched later on when accessed again.

In [24], Zhang et al. present VCluster, a thread migration middleware addressing both tightly coupled shared-memory multiprocessors and loosely coupled distributed memory multiprocessors. Their work focus on thread intercommunication and migration mechanisms. To use the VCluster middleware, the programmer must explicitly define what is the high-level thread state, relevant to be migrated to other node. In our work, the application source code does not need to be modified.

Grid systems have also been designed to take into account each node's own resources and task requirements. The work in [9] employees a multi-layer (CPU, node and site) set of reconfiguration strategies to dynamically adapt grid users' jobs to changes in the available CPU resources at each node. This adaptation is focused solely on scheduling the task to a different node, but once the task is scheduled, no further adaptation is possible. The task, and all its comprised threads, are run until completion on the same node. Our research also aims to dynamically adapt the runtime parameters and/or algorithms activated at the virtual machine in each node. Furthermore, resource monitoring is carried out during task execution and its threads can be spawned on less loaded nodes in the cluster.

6 Conclusion

In this document we described the architecture, implementation issues, and evaluation of A^2-*VM* , a research effort to design a cooperative Java virtual machine, to be deployed on clusters, able to manage resources autonomously and adaptively. It aims at offering the semantics of distributed execution environment transparently across clusters, each executing an instance of an extended resource-aware VM for the managed language Java.

Semantically, this execution environment provides a partitioned global address space where an application uses resources in several nodes, where objects are shared, and threads are spawned and scheduled globally. Regarding its operation, A^2-*VM* resorts to a policy-driven adaptability engine that drives resource management, global scheduling of threads, and determines the activation of other coarse-grained mechanisms (e.g., checkpointing and migration among VMs).

In summary, the goal of such an infrastructure is to provide more flexibility, control, scalability and efficiency to applications running in clusters.

Acknowlegments This work was partially supported by national funds through FCT - Fundação para a Ciência e a Tecnologia, under projects PTDC/EIA-EIA/102250/2008, PTDC/EIA-EIA/108963/2008, PEst-OE/EEI/LA0021/2011 and PROTEC program of the Polytechnic Institute of Lisbon (IPL).

References

1. B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The jikes research virtual machine project: building an open-source research community. *IBM Syst. J.*, 44:399–417, January 2005.
2. 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.
3. Godmar Back, Wilson C. Hsieh, and Jay Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *In Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 333–346, 2000.

4. Walter Binder, Jarle Hulaas, Philippe Moret, and Alex Villazón. Platform-independent profiling in a virtual execution environment. *Softw. Pract. Exper.*, 39:47–79, January 2009.
5. 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.
6. Stephen M. Blackburn and Kathryn S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 22–32, New York, NY, USA, 2008. ACM.
7. Jonas Bonr and Eugene Kuleshov. Clustering the Java Virtual Machine using Aspect-Oriented Programming. In *AOSD '07: Industry Track of the 6th international conference on Aspect-Oriented Software Development*. Conference on Aspect Oriented Software Development, March 2007.
8. Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: a Code Manipulation Tool to Implement Adaptable Systems. In *Proceedings of the ASF (ACM SIGOPS France) Journées Composants 2002 : Systèmes à composants adaptables et extensibles (Adaptable and Extensible Component Systems)*, November 2002.
9. Po-Cheng Chen, Jyh-Biau Chang, Tyng-Yeu Liang, and Ce-Kuen Shieh. A progressive multi-layer resource reconfiguration framework for time-shared grid systems. *Future Gener. Comput. Syst.*, 25:662–673, June 2009.
10. G. Czajkowski, M. Wegiel, L. Daynes, K. Palacz, M. Jordan, G. Skinner, and C. Bryce. Resource management for clusters of virtual machines. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid - Volume 01*, CCGRID '05, pages 382–389, Washington, DC, USA, 2005. IEEE Computer Society.
11. Grzegorz Czajkowski, Stephen Hahn, Glenn Skinner, Pete Soper, and Ciarán Bryce. A resource management interface for the Java platform. *Softw. Pract. Exper.*, 35:123–157, February 2005.
12. Grzegorz Czajkowski and Thorsten von Eicken. JRes: a resource accounting interface for Java. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 21–35, New York, NY, USA, 1998. ACM.
13. Grzegorz Czajkowski et al. Java specification request 284 - resource consumption management api, 2009.
14. Tiago Garrochinho and Luís Veiga. CRM-OO-VM: a checkpointing-enabled Java VM for efficient and reliable e-science applications in grids. In *Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science*, MGC '10, pages 1:1–1:7, New York, NY, USA, 2010. ACM.
15. Dominik Gront and Andrzej Kolinski. Utility library for structural bioinformatics. *Bioinformatics*, 24(4):584–585, 2008.
16. Richard C. G. Holland, Thomas A. Down, Matthew R. Pocock, Andreas Prlic, David Huen, Keith James, Sylvain Foisy, Andreas Dräger, Andy Yates, Michael Heuer, and Mark J. Schreiber. Biojava: an open-source framework for bioinformatics. *Bioinformatics*, 24(18):2096–2097, 2008.

17. King Tin Lam, Yang Luo, and Cho-Li Wang. Adaptive sampling-based profiling techniques for optimizing the distributed JVM runtime. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –11, April 2010.
18. Ivan López-Arévalo, René Bañares-Alcántara, Arantza Aldea, and A. Rodríguez-Martínez. A hierarchical approach for the redesign of chemical processes. *Knowl. Inf. Syst.*, 12(2):169–201, 2007.
19. Microsoft. CLR Profiler for the .NET framework 2.0, http://www.microsoft.com/download/en/details.aspx?id=13382, 2007.
20. Oracle. Java virtual machine tool interface (JVMTI), http://download.oracle.com/javase/6/docs/technotes/guides/jvmti/.
21. David W. Price, Algis Rudys, and Dan S. Wallach. Garbage collector memory accounting in language-based systems. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03, pages 263–, Washington, DC, USA, 2003. IEEE Computer Society.
22. Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Paul T. Groth, Gregory A. Hill, and Raul Saavedra. State capture and resource control for java: the design and implementation of the aroma virtual machine. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 11–11, Berkeley, CA, USA, 2001. USENIX Association.
23. Peter F. Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David Grove, and Michael Hind. Using hardware performance monitors to understand the behavior of Java applications. In *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3*, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.
24. Hua Zhang, Joohan Lee, and Ratan Guha. Vcluster: a thread-based java middleware for smp and heterogeneous clusters with thread migration support. *Softw. Pract. Exper.*, 38:1049–1071, August 2008.
25. 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.