CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2012; **24**:1421–1442
Published online 21 December 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1879

SPECIAL ISSUE PAPER

A checkpointing-enabled and resource-aware Java Virtual Machine for efficient and robust e-Science applications in grid environments

José Simão [2,3], Tiago Garrochinho [1,3] and Luís Veiga [1,3,*,†]

[1] *Instituto Superior Técnico, UTL, Portugal*
[2] *INESC-ID Lisboa, 1000 Lisboa, Portugal*
[3] *Instituto Superior de Engenharia de Lisboa, Portugal*

SUMMARY

Object-oriented programming languages presently are the dominant paradigm of application development (e.g., Java, .NET). Lately, increasingly more Java applications have long (or very long) execution times and manipulate large amounts of data/information, gaining relevance in fields related with e-Science (with Grid and Cloud computing). Significant examples include Chemistry, Computational Biology and Bio-informatics, with many available Java-based APIs (e.g., Neobio).

Often, when the execution of such an application is terminated abruptly because of a failure (regardless of the cause being a hardware of software fault, lack of available resources, etc.), all of its work already performed is simply lost, and when the application is later re-initiated, it has to restart all its work from scratch, wasting resources and time, while also being prone to another failure and may delay its completion with no deadline guarantees.

Our proposed solution to address these issues is through incorporating mechanisms for checkpointing and migration in a JVM. These make applications more robust and flexible by being able to move to other nodes, without any intervention from the programmer. This article provides a solution to Java applications with long execution times, by extending a JVM (Jikes research virtual machine) with such mechanisms. Copyright © 2011 John Wiley & Sons, Ltd.

Received 2 March 2011; Revised 9 August 2011; Accepted 1 September 2011

KEY WORDS: virtual machines; checkpointing; migration; JVM; e-Science; resource-awareness; quality of execution

1. INTRODUCTION

Object-oriented programming languages are, in current days, the dominant paradigm of application development (mostly Java and .NET languages). They prevail in desktop applications, application development itself (Eclipse), web application servlets, components, beans in application servers, and even in games, mostly in mobile scenarios. More recently, there are also increasingly more applications that have long (or very long) execution times and manipulate large amounts of data/information. This is becoming more and more relevant in various fields related with e-Science (mostly in the context of Grid and Cloud computing) where Java is becoming the dominant language, albeit used by researchers (programmers) who are often not computer engineers/computer scientists. Relevant examples include Chemistry, Computational Biology and Bio-informatics [1–3], with many available Java-based APIs (e.g., Neobio [4])

Often, when the execution of one of those applications is terminated abruptly because of a failure (regardless of it being caused by hardware of software fault, lack of available resources, etc.), all

*Correspondence to: Luís Veiga, INESC–ID, Rua Alves Redol 9 1000–029 Lisboa.
†E-mail: luis.veiga@inesc-id.pt

of its work already carried out is simply lost, and when the application is later re-executed with the same parameters and input (e.g., as in the case of a data-processing job), it has to restart its work from scratch, wasting resources and time, and being prone to another failure, to delay its completion with no deadline guarantees.

In a grid environment, applications running on a given node compete for the finite resources of that machine (e.g. CPU, memory, input/output (I/O)). Each application is running to produce a set of results on behalf of a given user, but not all users have some of the execution requirements or some of the priorities to complete their work. The work of Silva *et al*. [5] classifies users in four different types in order to apply differentiating policies to the work of these users. In academic institutions, for example, the same grid can be used to run e-Science applications by students in different academic levels. Using the same infrastructure will have less costs and will be easy to maintain. Nevertheless, the managers of the infrastructure will want to impose a high-level policy and give a distinguished execution quality to different types of students. The mechanisms to obtain this quality of execution can range from restraining resource consumption in coarse grain (e.g. CPU usage, physical memory allocated) to the migration of the application to another node.

A possible solution to solve these problems is through mechanisms of checkpoint and migration of applications made available through a resource aware-enabled object-oriented virtual machine (OO VM). With these mechanisms, an application becomes more robust as most of the work or calculations already performed can be recovered, and the execution can be resumed from an earlier point in time. It gains flexibility by being able to move to other nodes, without intervention from the programmer, regulated by a global policy enforced to each OO VM.

Traditional mechanisms of checkpoint and migration are supported at different levels: (i) process level (whether initiated by application with its own code or via specific libraries or as a facility offered by the modified or extended operating system (OS) [6]); and (ii) system virtual machine (System VM, e.g., Robert Bradford *et al.* [7]). These approaches are insufficient for the following limitations: (i) they either require to store/transfer information that is not on the application itself (e.g., information on the OS on which it runs); or (ii) limit the portability of it. Therefore, as the majority of the object-oriented programming languages execute their applications on OO VM (also known as high-level language virtual machine, e.g., JVM, .NET CLR), our solution proposes an approach to the checkpoint and migration mechanisms at this level.

In particular, regarding System VM checkpointing technology, it overshoots our intended scenarios of OO-based e-Science applications because of the inherent overhead of saving the whole memory of the machine (including the application, code and data of other running applications, the VM code itself, all of the OS kernel, buffers and dynamic libraries, full hard-disk images). Our solution, being tailored to OO VMs, saves only the relevant data to be able to restore the application on another OO VM in another host.

There is already some research in the area of checkpoint and migration solutions at the OO VM level; however, some existing solutions are embedded in the context of mobile agents and for that, are very limited; that is, they only portray a single thread on a very limited and controlled environment (e.g., MobileJikesRVM [8]). Other solutions either have efficiency problems (e.g., JavaGo [9], JavaGoX [10], Brakes [11], ITS [12], that have a performance penalty in applications runs exceeding an average of 300%), or they pass the responsibility to the programmer (e.g., Web Agent based Service Providing (WASP) [13], Object Broker Infrastructure for Wide Area Networks (OBIWAN) [14]) that must cooperate with the checkpointing mechanisms, which limits transparency, or are solutions in which completeness is not well addressed, and the problem is specifically related to the external state of an application (e.g., files, client sockets).

This article provides a novel solution to Java applications with long execution times by incorporating checkpoint and migration mechanisms in a JVM (Jikes RVM [15]). It is able to checkpoint multithreaded applications, ensuring the checkpoint is a consistent snapshot of the execution taking into account thread concurrency and synchronization, while avoiding application pause by performing the checkpoint concurrently (or incrementally) alongside with the application execution.

Our techniques rely on two base mechanisms: on-stack replacement (OSR) and yield points, existent in many other VM implementations (e.g. Sun HotSpot) and other VM technologies

DOI: 10.1002/cpe

(e.g. .NET CLR, Mono). Therefore, our techniques could be applied to other VMs. The main objectives are focused on the problems of transparency and completeness and how these mechanisms can be activated according to low-level resource management and monitoring driven by policies. Our proposed solution takes into account the following set of properties:

- **Transparency**: The mechanisms should not be constructed in such a way that impose responsibilities to the programmer. So, no application changes are required. A controller program (command line input) is provided, which communicates with the application-running environment (VM) to take advantage of these mechanisms (that can be used by other users than the developer himself). Applications should not realize changes of environment or that were recovered using checkpoint or were transported to another environment using migration.
- **Flexibility**: Although there are no mandatory responsibilities to the programmer, we propose an API that allows himself to control checkpoint and migration mechanisms in his application.
- **Consistency**: The state of an application remains free of inconsistencies, even after a resume/rescue operation. In functional terms, the application continues its execution as if the checkpoint or migration never happened (does not include temporal matters).
- **Completeness**: The mechanisms must portray the whole state of an application: code, data (e.g. heap), execution state (e.g. stack, threads), external links (files, client sockets), state regarding native execution (java native interface, JNI), and Java synchronization monitors. Note that it is not intended to store/carry the whole virtual machine (VM) as a block. It is intended to only take into account the minimum state relative to the application itself so that this minimum is enough to reconstruct the execution state of the application on another virtual machine instance (local or remote).
- **Portability**: Partly provided by the OO VM approach but it is necessary to have the VM modified/extended in all machines/nodes. It is also desirable that the same VM source code compiled in a particular OS and architecture may be able to use checkpoints generated by the same VM compiled on other systems.
- **Efficiency**: The additional constant overhead, imposed during error-free execution, to the performance of running applications should be minimized. The performance cost during the activation of the checkpointing mechanisms should be proportional to the applications itself.
- **Robustness**: The mechanisms, at least, shall not affect the application or be a source of new exceptions that were not envisioned by the developer (e.g., when it is not possible to do a checkpoint or migration, the application must continue normally).

The rest of this paper is organized as follows. In Section 2, we overview the related work. In Section 3, we describe the architecture and general vision of the components of the proposed solution. In Section 4, we address the most relevant details of the implementation. In Section 5, we present results obtained in the evaluation of the developed mechanisms, as a form to illustrate their feasibility, efficiency, and sources of overhead. In Section 6, we close the paper with some conclusions.

2. RELATED WORK

In this section, we address the related work on checkpointing mechanisms and on resource management in the context of OO VMs.

Checkpointing, restore, and migration mechanisms

Existent mechanisms for checkpoint and migration are implemented at different levels: **System VM**, **Process level**, and **OO VM**, the main subject of this work. Naturally, the level of implementation influences the type of information maintained; that is, depending on the level of implementation, we can obtain checkpoint and migration of **OSs** (i.e., a complete machine or platform installation), **applications**, or **threads**. Regardless of the level of implementation, the **execution state** that the mechanisms have to persist can be divided into two parts: internal and external state. The internal state includes pending signals, address space (heap, stack and any region mapped), and internal registers. External state covers file descriptors, the actual contents of the files, and sockets.

DOI: 10.1002/cpe

Regarding the internal state, the problem in general is more or less well addressed; however, although not explicitly stated, some of the existing solutions require the execution environment to be well defined and controlled (e.g., mobile agents, single-thread applications, required cooperation of application code with checkpointing mechanisms). Conversely, for external state, solutions already have some problems. They are either incomplete and may not work around the issue of files mobility or address it simply by imposing the usage of a distributed file system. For some scenarios, such as large-scale settings, it can be costly in terms of performance, or plainly incompatible, to be dependent on a distributed file system [7].

As our work performs checkpointing at the OO VM level, we focus our discussion of related work on VM level schemes. At this level, the vast majority of checkpoint and migration solutions use a serialization mechanism provided by the VM itself. As an example, the serialization mechanism of JVMs allows to store and retrieve the state of an object and also allows the transfer of the same object between different machines/nodes. With only one mechanism, we can have information persistence and transfer.

Object-oriented virtual machine checkpoint and migration solutions can be further subdivided into two classes regarding their approach (both address threads and application data—object heap):

- **OO VM internal level**: this approach fulfills the requirement of completeness by having access to the whole execution state. However, these solutions have problems of portability (other VM implementations also have to incorporate code changes in order to make the checkpointing and restore mechanisms work). This approach is usually accomplished through modifications or extensions of the OO VM's own internal code, introducing new features from libraries and providing checkpoint or migration. Examples of such solutions include Merpati [16], OCaml Virtual Machine (OCVM) [17], Collaboration and Coordination Infrastructure for Personal Agents (CIA) [18], MobileJikesRVM [8], Sumatra [19], JavaThread [20], Nomads [21], ITS [12], and Jessica2 [22].
- **OO VM application level**: this approach has the main advantage of being portable (it needs no modifications to be applied to VM code) but has serious efficiency issues (code expansion) and does not meet the requirement of completeness. At this level, the application code (source code or bytecode) is transformed by a preprocessor (or a bytecode enhancer) that adds new instructions to the application code (instructions which serve to capture or restore the application state or trigger other code that performs it). Examples of such solutions include WASP [13], JavaGo [9], JavaGoX [10], Brakes [11], and M-JavaMPI (Message Passing Interface) [23].

Additionally, some of these solutions, such as CIA [18] and M-JavaMPI [23], take advantage of the debugging library provided by the JVM architecture, known as the Java Platform Debugger Architecture (JPDA), to store and retrieve the execution state of an application. Nonetheless, JPDA, when used to implement mechanisms of checkpoint and migration, has some limitations, the most significant being that these solutions are only able to extract the state of a single thread.

Among the solutions already mentioned at the OO VM level, there are few that support checkpoint or migration of applications. Next, we offer a brief comparison of those solutions with the general view of the solution we propose in this article.

Merpati [16] is an application checkpoint solution. It cannot deal with application threads already blocked prior to performing a checkpoint, and it cannot handle a state that does not belong to the VM, as is the case of a native state. Our approach can deal with threads already blocked, and JNI-related state is processed in such a way that it is not explicitly saved, but at the same time, we ensure consistency of the VM and application upon restore.

OCVM [17] is a checkpoint solution designed for applications at the OO VM internal level, although it is a very high level, which compromises its completeness, and is also very restricted in scope because it does not target a VM with a widely used programming language such as Java.

Web analytics solution profiler [13] and OBIWAN [14] are solutions for the migration of mobile agents, but they can deal with multiple threads. This solution manipulates Java source code in order to add additional instructions to support migration. This has two main disadvantages: it does not support applications whose Java source code is not provided, and, maybe worse, it needs the assistance

DOI: 10.1002/cpe

of the programmer to address limitations of the solution, regarding when and where a checkpoint can be performed and what data to be included in it. Our approach does not suffer from these limitations.

Lastly, M-JavaMPI [23] was designed to support application migration, but it employs JPDA as the core of the solution, and for that reason, it can only support applications with a single thread. In general, Merpati, WASP, and M-JavaMPI have transparency problems (in the worst case, they force the programmer to modify his program in order to explicitly invoke the provided mechanisms, or the programmer has to be aware of an additional programming model, e.g., MPI). The extraction of external state is also an issue. Most solutions support neither sockets nor files; in most cases, applications are relocated simply by using a distributed file system, which raises performance, scalability, and administrative issues in large-scale settings.

Resource monitoring and management in object-oriented virtual machines

Resource monitoring and management are required in OO VMs for Grid environments because of two major reasons: (i) monitoring is required to obtain some kind of measurement of resource usage by an application; and (ii) management is required in order to determine the amount of resources should be awarded to an application and to enforce those limits somehow. Such mechanisms, regarding low-level aspects of an execution environment, may need to be continuously or at least frequently activated, enforced, or inquired. Therefore, their implementation must aim at minimizing the impact to the overall application's performance. The work of Sweeney *et al.* [24] aims to accomplish these goals using hardware performance counters. Thus, the Jikes RVM was extended with a performance monitor layer through a native C library. Although effective in monitoring, it does not support any kind of enforcement on resource consumption restriction. Regarding the implementation, it is dependent on a previous version of Jikes RVM employing a scheduling algorithm that maps N VM threads to M native threads.

Some systems relax requirements on low-level precision in exchange by portability. The profiling framework of Binder [25] performs static instrumentation to core runtime libraries and dynamically instruments the rest of the code. Thus, the instrumented code can periodically call pure Java agents to process and collect profiling information.

Other high-level VMs have been either extended or designed explicitly to integrate some form of resource accounting [26–28]. An influential work was the Multi-tasking Virtual Machine (MVM) [26], on the basis of the HotSpot VM. It supports isolated computations (*isolates*), similar to different address spaces, to be performed within a single instance of the VM. Furthermore, MVM is able to impose different constrains regarding consumption of specific *isolates*. MVM resource management work is related, and probably inspired, with the Java Specification Request (JSR) 284 [29]. Still, MVM only runs on Solaris on top of SPARC's hardware. Our work builds upon this JSR work, implementing it in the context of a widely accessible VM.

The work in [27] and [28] enables precise memory and CPU accounting. Nevertheless, they do not provide an integrated interface to define any resource consumption policy, which may involve VM, system, or class library resources.

3. ARCHITECTURE

In a grid where nodes maybe running multiple e-Science applications, it is important to regulate the *Quality of Execution* (QoE) of each application because they may be running workloads with different user and application priorities and deadline requirements. Although related, the goals expressed by the QoE requirements are of a different nature than those of the requirements expressed in Service Level Agreements and QoS. Both service level agreements and QoS are normally associated with commercial service providers. These agreements focus on the quality or number of transactions that the provider is expected to accomplish and on the downtime a user is willing to accept. They are expressed as legal contracts, and failure to meet them results in penalties for the provider.

In e-Science infrastructures, the goals are more relaxed because no commercial service is being provided. Nevertheless, we can still express the intended (or ideal) amount of resources an application should be provided in order to run according to its priority. Therefore, it is of most relevance

DOI: 10.1002/cpe

that such systems incorporate mechanisms to enforce usage policies or even resource booking, as emphasized by the recommendations of the Networking, Computing Capacity, and Data Storage System subgroups formed by the Department of Trade and Industry Steering Group to address the issues and challenges related to the development of a UK e-Infrastructure for research [30].

The major element of our work is an enhanced OO VM capable of checkpointing, restoration, and migration of applications. The activation of these mechanisms is regulated by a QoE Controller. On the basis of execution requirements (e.g., CPU, memory and network usage), the QoE controller can apply two coarse grained measures: (i) checkpoint and suspension of a VM; and (ii) migration of the application execution state to another node. To avoid disturbing the applications whose QoE is to be favored, our solution is to apply the former mechanisms to the VMs executing applications with lower priority, restraining them from using the shared resources or promoting their migration to another available or default node.

In case of migration for load balancing, our scheduling is based on receiving periodic information regarding the load of each host, broadcast over the local area network. Each host selects a target randomly from those whose load is below the median, to balance the load while preventing suddenly overloading one or two least loaded hosts.

Next, we describe the main aspects of the architecture of the checkpoint-enabled OO VM. We start with an overview description of the mechanisms for checkpoint/restore/migration and resource regulation and then with the internal architecture of the extended VM. In Figure 1, we present the overview of the features supported by this work:

1. Checkpoint and its corresponding restore. This operation collects all the necessary information (and only that) about the application and VM runtime so that it can be suspended and later restored. The checkpoint can be made to the local or distributed file system.
2. Migration between two nodes. The migration is done directly between two nodes.
3. Monitoring and control of VM resources. Resource usage is reported to the QoE controller, which can restrain their usage or apply coarse grain actions (i.e. checkpoint/suspend or migrate) to guarantee execution quality, in that node, to relevant applications.

The architecture of this work presented in Figure 2 consists of a set of components that focus on the transparency and completeness properties described earlier and/or on data transfer. There are three primary components:

- **Application**: Executing in the context of an extended VM with mechanisms to support checkpoint, restore, and migration.

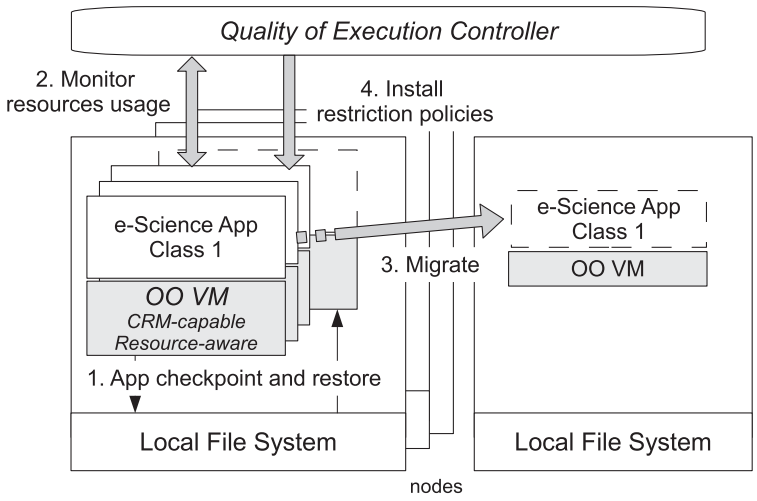


Figure 1. High-level architecture with mechanisms usage overview.

DOI: 10.1002/cpe

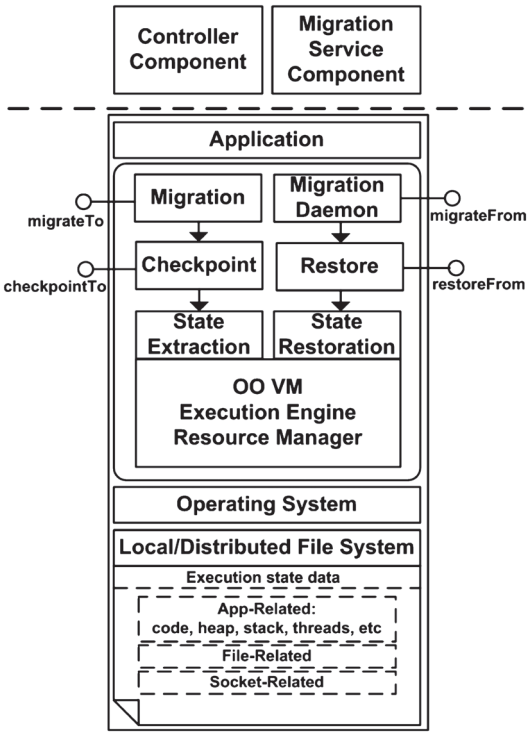


Figure 2. Virtual machine architecture in one node.

State extraction captures the execution state related with all threads within the application. Checkpoint has the obligation to stop all threads (to guarantee consistency), calls the state extraction, and finally saves the state persistently into a file system. Migration calls the checkpoint and sends that execution state via network.

State restoration has the responsibility to rebuild the execution state in a newly created application, which corresponds to reconstructing and resuming the execution of all stack frames (a stack frame corresponds to a call to a subroutine that has not yet terminated with a return) for all threads and when ready, restart execution. Restore guarantees that the newly created application can initiate state restoration and additionally, if requested, obtains the state from a file system. A migration daemon is used in migration, which receives the state from the network. When the state is available, the migration daemon calls a restoration with that state.

`CheckpointTo` and `migrationTo` methods/services are triggered by the controller. A special thread is listening in a specific socket, which makes it possible to receive external orders. When triggered, corresponding commands are passed into the checkpoint or the migration internal components. `MigrateFrom` and `restoreFrom` triggers are just simple input channels to receive execution state information.

- **VM controller**: Command line program, which communicates with the application to take advantage of the mechanisms developed. It is from this that a user controls the mechanisms developed on an application, parameterizing it with commands/instructions depending on the desired result. This lower-level controller, one for each running VM, is instructed and inquired by the QoE Controller running on the node.
- **Migration service**: Service present on all nodes, which aims to receive migrated applications. This server also responds to requests for classes and files transfer that are handled on demand.

In order to use checkpoint and migration mechanisms externally, the VM controller must discover the socket port that the application (i.e., the VM running it) is listening (to receive `checkpointTo` or `migrationTo` trigger instructions). The VM controller also listens in a specific socket port and sends an OS signal [31] to the application (using its process identification, e.g., Unix Process ID).

DOI: 10.1002/cpe

Note again that in this case, we regard as the application the whole process running the VM instance executing the application. Interactions are actually performed by VM code. When the application receives that signal, it performs a callback by connecting to the controller by socket, and from that moment, it is possible to exchange messages between both.

In case of application migration, the interaction between the various components is done as follows. The application VM instance communicates with the remote migration service to initiate a migration. This service is responsible for starting a newly created VM instance (that will restore the checkpointed application) that will listen on a given socket (internal Migration Daemon). Once the state can be transferred, the migration service responds to the original application with the port it must send the state to. From this moment on, the state is transferred from the original application to the newly created application directly.

4. IMPLEMENTATION

The mechanisms of checkpoint, migration, and resource management were developed in the Jikes RVM (release 3.1.0). Jikes RVM is a VM designed to run Java programs, whose distinctive feature compared with other JVMs is also implemented in Java. Nonetheless, unlike other JVMs, Jikes RVM does not need to rely on a second JVM to bootstrap and run.

In this section, we focus on the implementation details that provide better understanding of the solution developed. First, we describe how the execution state can be saved persistently on a disk and transferred across networks. State extraction and restoration is detailed taking into account the consistency property. Then, we highlight the most relevant implementation differences regarding the concurrent version of the checkpoint mechanism. Finally, the mechanisms used by the QoE controller to monitor and control resources are described.

Execution state: disk persistence and transfer

Most solutions discussed in related work at OO VM level use a serialization mechanism supported by the VM, on which they develop the checkpoint or migration mechanisms. This solution is no different. We are taking advantage of the Java serialization mechanism, implemented by `GNU's Not Unix (GNU) CLASSPATH` (http://www.gnu.org/software/classpath/) and supported in Jikes RVM, to persist and carry the information related to the execution state of a running application.

However, Java serialization requires that every class that must be serialized implements the `Serializable` interface. Thus, we would have to trust all application classes that implemented that interface, and consequently, we would give responsibilities to the programmer to do it so (violating the property of transparency). However, this interface serves only as a tag to mark which classes are serializable or not because taking, for example, the `Thread` class, this class has dependencies on the environment it runs on (OS, system calls, or native library dependencies) and cannot be automatically serialized. Because our solution addresses the issue of mobility of such objects, activating the default serialization internally, there is no need for application code to explicitly implement this or any other interfaces.

Consistency

To obtain a consistent state of the VM for its checkpoint, it is required to ensure that all nonsystem threads are stopped (we only need to take into account application threads). Jikes RVM has support for yield points, which are inserted automatically by the just-in-time (JIT) compiler, on method prologues, epilogues, and loop back edges. These yield points are safe points where the VM can take control over a thread in order to make it stop because in such points, threads are not changing the VM state or executing any application instructions (bytecodes).

In the Jikes RVM, threads are classified either as system threads (e.g., GC, Finalization) or as application threads. The threads of the Checkpoint Restore and Migration (CRM) mechanisms are also classified as a system. When there is a checkpoint request, all nonsystem threads are signaled

DOI: 10.1002/cpe

to stop, and this is handled by the method `checkBlock`, called when the execution reaches a yield point.

This would be sufficient for threads that are not blocked. But, if a thread is already blocked (e.g., in a read from input), then it cannot reach a yield point. This would prevent the VM from ever being able to perform the checkpoint. However, if a thread is already blocked, it is in a safe point by the same reasons of yield points (thus called effectively safe). So, if all threads are in safe points or are effectively safe, the VM can be stopped in a consistent state, with some additional care.

It is true that effectively safe threads are indeed running (as far as the VM is concerned, they are in the midst of executing a bytecode instruction). But, when the native operation is finished, the thread returns to the control of the VM, and it will be blocked on a yield point before continuing, enforcing the desired property of consistency.

Execution state: stack frames saved

Figure 3 shows which stack frames are saved for each type of thread. Shaded stack frames are the ones only saved. The black fill marks the first frame to be saved.

A thread in a safe point has always the same first stack frame but effectively safe threads do not. However, every time a thread is effectively safe it enters into internal VM code and is forced to save the frame pointer (FP, stack pointer that points to the last created frame) and the instruction pointer (IP, pointer that points to the next instruction to be executed) pointers.

Effectively safe threads are always restored in the same safe point. If a thread returns from its effectively safe state while in state extraction (Figure 3, transition from thread #2 to #3), then, on restore, it will appear as if it never advanced the execution (it will look exactly like thread #2), which is the desired state.

Execution state: extraction

We are taking advantage of OSR [32] to extract and restore the execution state of the threads of a running application. OSR makes it possible to take a stack frame from the stack and substitute it with another one.

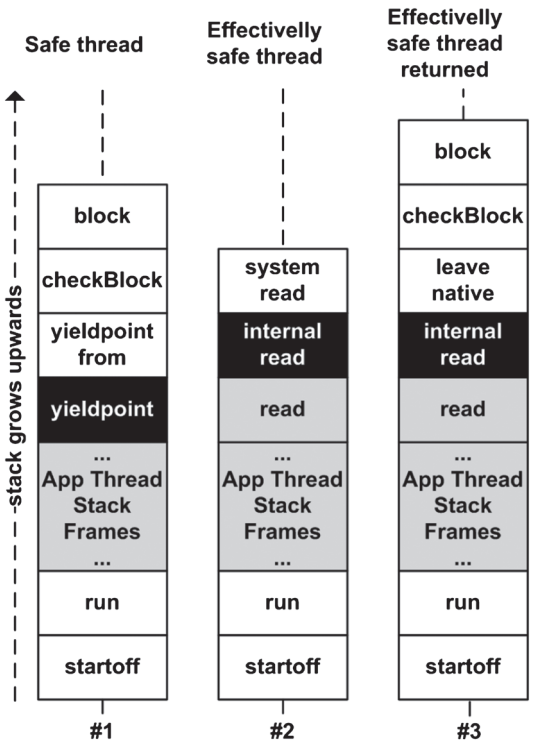


Figure 3. Safe and effectively safe thread examples.

DOI: 10.1002/cpe

Jikes RVM has support for two *JIT* compilers: baseline and optimized. Baseline stack frames are fully observable and easily extracted, but optimized ones are not. The optimized compiler has a more intricate operation and chooses points in a code where OSR can occur (points where the relevant OSR maps are created), and for that, we currently only support baseline stack frame extraction. We are aware of the work done in [33] that disabled some optimizations without losing significative performance, but we also know that this solution only works 60% of the times. So, future work must address the full support of optimized stack frame extraction.

Baseline stack frame extraction is done as follows. First, it analyzes bytecodes in order to determine the type of locals and stack operands at a certain bytecode index, just like a common bytecode verifier. The produced result has to be adjusted with GC maps because there can be object variables (references) that may be uninitialized at the current bytecode index. Additionally, the numbers of locals and stack operands are counted. When ready, the baseline state extractor uses both type/number of locals and stack operands to retrieve the full data from the stack frame. The structures of the information retrieved are the following:

- Local and stack operand variables (includes method arguments and reference to object `this`).
- Bytecode IP.
- Compiler type (baseline or optimized).
- Method name (composed by class/descriptor/method (e.g. mypackage.myclass/(I)V/mymethod)).
- Next stack frame execution state.

Execution state: restoration

On restore, every stack frame execution state is recompiled with a special prologue with the following additional bytecode instructions that have the following responsibilities:

- Recover local and stack operand variables from the checkpointed stack frame
- For every checkpointed stack frame, recursively, recreate it by invoking its prologue code.
- And finally after that, recover the bytecode IP preserved.

After recompilation, for every checkpointed thread, a new thread reruns all compiled stack frames in the same order as they were before, and when finished, the thread automatically blocks to make the restore consistent. When all threads are ready, the VM can restart with the previously checkpointed execution state.

Execution state: threads and thread synchronization state

The checkpoint must include information identifying the application threads and their synchronization-related state in order to resume later the application execution correctly, with the same number of threads, with the same synchronization context (e.g., monitors owned by each thread, waiting or blocked on a monitor). Because neither the `Thread` nor the `RVMThread` (thread internal representation, object that represents a thread within the VM) class is serializable, and also the information regarding internal synchronization locks has very strong environment dependencies, we are faced with two alternatives:

1. Create a special externalized version of the object with the minimum required information (represented only by primitive data types), which allows the reconstruction of the object on restore, include all the monitors; or
2. Avoid serializing the object, walking back the thread stack to a frame position where the object had not yet been created. This is only possible when code on restore is deterministic and can recreate all the information like it was before. We prefer this kind of solution for the thread synchronized state.

A thread within synchronized methods or statements can be in one of the three states: monitor owner, blocked in the entry set, or blocked in the wait set. Because we avoid saving internal synchronization locks on restore, they have to be recreated. The owner of the lock on restore must always reacquire it again, and this is enforced. Entry and wait set threads are walked back to a stack frame

DOI: 10.1002/cpe

that on restore reruns the lock and wait code again, which makes them lock in the right set. It is true that the order in both sets can be different from the old state, but monitor owner competition is very implementation dependent, and for that, this is not a requirement as it is usually regarded as bad programming to rely on relative speed of threads for correctness in an application.

Saving and restoring external files and connections

In the Java Class Library, all streaming I/O is done through implementations of the interfaces `InputStream` and `OutputStream`. The classes `FileInputStream` and `FileOutputStream` are the less specialized ones that provide the abstraction to deal with streaming file I/O. Although a file can be accessed through other types, which decorates the behavior of these two (e.g., `BufferedInputStream`), all reads or writes will eventually come down to the two previously mentioned classes. In this hierarchy, the only types with dependencies to native resources are the classes `FileInputStream` and `FileOutputStream`, namely the operative system file descriptor.

When the checkpoint mechanism is activated and there are open files, a thread can be either blocked in a read/write operation or just have the file opened. In the former case, the thread is effectively safe because it is blocked in a native operation. In the latter, it is in a safe state because it is blocked following a yieldpoint. To handle both cases, the serialization of objects of type `FileInputStream` and `FileOutputStream` has to be carried out in a proper manner. In Listing 1, we show an example of how to serialize the type `FileInputStream`.

```
1  public class FileInputStream extends InputStream implements Serializable
   {
3    // native representation of the descriptor
     transient private FileDescriptor fd;
5    transient private FileChannelImpl ch;
     //...
7    private void writeObject(ObjectOutputStream out) throws IOException
     {
9                  // first, we save every non native information
       out.defaultWriteObject();
11                 // save relative path and cursor of file
       String filePath = ch.getPath();
13     long fileCursor = ch.getSavedPosition();
       // write native information
15     out.writeObject(filePath);
       out.writeLong(fileCursor);
17     out.flush();
     }
19
     private void readObject(ObjectInputStream in) throws IOException
21   {
       // read all non native information
23     in.defaultReadObject();
       // recover path and cursor
25     String filePath = (String) in.readObject();
       long fileCursor = in.readLong();
27     File file = new File(filePath);
       try {
29       // open file descriptor
         ch = FileChannelImpl.create(file, FileChannelImpl.READ);
31     }
       catch (FileNotFoundException fnfe) { ... }
33     catch (IOException ioe) { ... }
       }
35     // reset the cursor to the correct position (previous one before chekpoint)
       ch.position(fileCursor);
37   }
     //...
39 }
```

Listing 1. Serialization for instances of FileInputStream.

DOI: 10.1002/cpe

When a thread is reading a file, it is effectively safe. In this case, the thread holds (indirectly) a reference to a `FileChannelImpl` and a `FileDescriptor`. To avoid having to serialize these two objects, the execution state extraction begins only in the caller of the method that was reading or writing the file when a checkpoint was requested. When restored, the thread will repeat the I/O operation from the beginning, avoiding consistency problems.

The running thread is essentially unaware that the actual OS file descriptor has changed because the the file was reopened. It uses the same stream reference whose file cursor was reset to the previous position before the checkpoint.

The same takes place with streaming over client sockets; they are reconnected on restore to the previous addresses (except there are no cursors), for example for applications that are fetching data or sending results to a key-value store. Thus, while preventing immediate exception and failure on the next send/receive operation after restore, we still depend on the application communication protocol to be able to progress correctly. Our current work does not target server-like applications, and so, we do not aim for that functionality.

The contents of, alternatively according to configuration, open files or the current directory are eagerly copied to the destination node, or if so configured, file accesses are redirected back to the original host (without supporting further levels of chaining).

Additional issues with the Java Native Interface

A thread with JNI state is just like an effectively safe thread. State extraction starts on the last frame that makes the JNI call. If JNI returns, it will block. On restore, it will happen as if that JNI call never happened and so is repeated again. This stays consistent within the VM.

4.1. Concurrent checkpointing

Our checkpoint mechanism can also run concurrently with the main program, preventing full pause of the application during checkpointing, thus further reducing the overhead experienced by applications. There are two main implementation issues regarding concurrent (or incremental) checkpointing: (i) ensuring checkpoint consistency because the application continues executing while the checkpoint is created; and (ii) avoiding excessive resource consumption (CPU, memory) because of the extra load of executing the application and the checkpointing mechanism simultaneously, which could lead to thrashing and precluding the very performance gains sought by executing the checkpointing concurrently.

The first issue is related with isolation and atomicity. The checkpoint, while being carried out concurrently, must still be atomic with regard to the running application. This means that it must reflect a snapshot of the execution state that would also be obtained with the application paused or suspended (while the application is not modifying its state). Otherwise, there could co-exist in the snapshot objects checkpointed at different times, making the whole object graph inconsistent and violating application invariants. In essence, the challenge in this operation is that the application's working set (and VM's internal structures) will change while the checkpointing is being carried out. If the changes were to be reflected into the data being saved, the checkpoint would be useless for being inconsistent.

The second issue stems from the fact that if we want to simultaneously freeze a *clone* of the application state in time (to be able to save it in the checkpoint concurrently), while the application keeps executing and accessing the *original* object graph, it would potentially almost double the memory occupied by the VM. Furthermore, performing the serialization of the *clone* object graph will cause contention for the CPU, with the application code that is simultaneously being executed (although the OS is able to interleave their execution with some degree of efficiency).

Fortunately, two aspects of current architectures help when dealing with these issues: (i) lazy memory duplication, as embodied in *copy-on-write* mechanisms provided by the memory management modules in modern OSs; and (ii) the increasing prevalence of multicore hardware, available in most computers today. These two aspects are leveraged to ensure concurrent checkpointing offers smaller overhead to applications running.

DOI: 10.1002/cpe

In fact, the *original* and *clone* version of the object graph need not exist physically in their entirety. To efficiently support this, we use the *copy-on-write* mechanism that allows two processes to share the whole of the address space, with pages modified by one of them copied on demand. Currently, our implementation in Linux relies on Linux's system call, `fork()`, which has the desired semantics [34]. In Windows, the same primitive and semantics is available through the Portable Operating System Interface for Unix subsystem, thus ensuring portability across the two OSs. Therefore, the memory overhead will be bounded to the memory pages containing objects that are modified during the checkpointing. Because of the locality in memory accesses during application execution (locality-of-reference and working set principles), this amount is limited.

Figure 4 illustrates how the concurrent checkpoint progresses, along with the application, in comparison with the serial (nonconcurrent) approach. With serial checkpointing, the total execution time of an application is, expectably, the sum of the time performing its calculations or processing (hereafter calculation time), with the time to perform a checkpoint (once in the figure) multiplied by the number of checkpoints taken. Therefore, checkpointing is always in the critical path regarding the total execution time, precluding so frequent checkpointing (for instance, very large working sets and not very long executions, probably only once at mid execution time).

With concurrent checkpointing, most of the checkpointing time is removed from the critical path regarding total execution time (only the time to setup the child VM remains). This makes it feasible to perform checkpoints more frequently, without significantly penalizing application execution times, thus reducing even more the amount of lost computation (lost work) whenever a failure takes place.

The internal functioning is as follows. When checkpoint is triggered, the VM calls `fork()` to create a *child* VM, that is, another process, sharing the whole address space, responsible solely for carrying out the actual checkpointing operation. The *copy-on-write* semantics ensures that the *child VM*'s working set will be consistent, even while the *parent* VM continues to update data because of application execution. When checkpointing is complete, the child VM terminates as it is no longer required. The additional overhead of creating a new process is counterbalanced, manyfold, by the fact that the application no longer needs to be paused during checkpoint creation.

This strategy can be used in other VMs besides Jikes RVM. In most cases, it will even be simpler to do so because most other VMs are implemented using the operative system native language, C. Nevertheless, in the Jikes RVM, these operations (`fork` and akin) are also efficiently supported by the available JIT compilers. When a properly annotated method is called, the JIT compiler will generate a call to a C language *stub*, using the platform's underlying calling convention. Our stub then calls `fork()`, with reduced overhead, and returns the result to the calling VMs (i.e., *parent* or *child*).

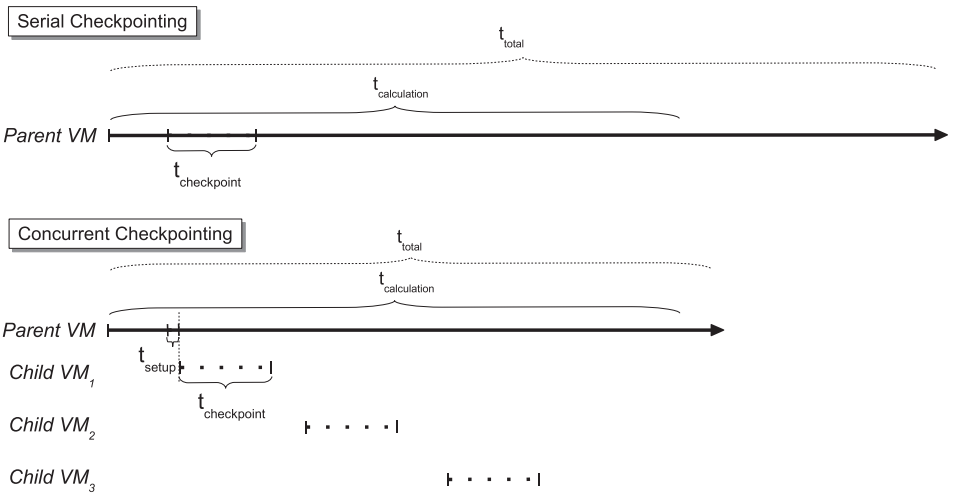


Figure 4. Each concurrent checkpoint runs in a *child* virtual machine. $t_{\text{calculation}}$ is the free run time, without any checkpoint. t_{total} is the total execution time, considering either serial or concurrent checkpoint.

DOI: 10.1002/cpe

```
1
public class HistoryBasedNotification implements Notification {
3    int _idx, _nSamples;
     long _currentSum;
5    long[] _samplesHistory;
     private long _maxConsumption;
7    private IAction _onLimitAction;

9    public HistoryBasedNotification(
            int historySize, long maxConsumption, IAction onLimitAction) {
11       _onLimitAction   = onLimitAction;
         _samplesHistory  = new long[historySize];
13       _idx = _nSamples = 0;
         _maxConsumption  = maxConsumption;
15   }

17   public void postConsume(
            ResourceDomain domain, long previousUsage, long currentUsage) {
19       if (_nSamples < _samplesHistory.length)
            _nSamples++;
21       else
            _currentSum -= _samplesHistory[_idx];
23       _samplesHistory[_idx] = currentUsage;
         _currentSum += currentUsage;
25       _idx = (_idx + 1) % _samplesHistory.length;
         if ((_currentSum / _nSamples) >= _maxConsumption &&
27          _nSamples == _samplesHistory.length)
               _onLimitAction.exec();
29   }
}
31
HistoryBasedNotification onCPULimit = new HistoryBasedNotification(
33   5, 75, new MigrationAction()
);
```

Listing 2. Notification policies of CPU usage.

4.2. *Policies in the QoE controller*

The management of a given resource implies the capacity to monitor its current state and to be directly or indirectly in control of its use and usage. The resources that can be monitored in a VM can be either specific of the runtime (e.g., number of threads, number of objects) or be strongly dependent on the underlying architecture and OS (e.g., CPU usage). To unify the management of such disparate types of resources, we have started the implementation of JSR 284—The Resource Management API [29] in the context of Jikes.

The JSR 284 elements are *resources*, *consumers*, and *resource management policies*. Resources are represented by their attributes (`ResourceAttributes` interface). For example, resources can be classified as *Bounded* or *Unbounded*. *Unbounded* resources have no intrinsic limit 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 that 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*.

The consumption of resources can be monitored or regulated by policies outside the VM, namely by the QoE controller. When instantiated, the notification and regulation policies are implementations of the `Notification` and `Constraint` interfaces of JSR 284, respectively. Notification policies will determine which mechanism will be activated. Listing 2 shows an example of a general notification policy that takes into account a window of n observations. This policy can be bounded to different resource domains, including the one that regulates CPU usage. At the end of

DOI: 10.1002/cpe

the code snippet, we illustrate how this policy could be instantiated to determine the migration of the application running in a VM reporting a CPU usage above 75% for the last 5 observations.

Changes to the VM and classpath
Our first experiences were done in order to have control on the spawning of new threads, a common source of CPU contention and performance degradation when multiple applications are running. We made modifications to the Jikes runtime classes and extended the GNU classpath. The Jikes boot sequence was augmented with the setup of a *resource domain* to manage the creation of application level threads. VM threads (e.g., GC, finalizer) are not accounted. The Jikes component responsible for the creation and representation of system level threads was extended to use the callbacks of the previously mentioned *resource domain*, such that the number of new threads is determined by a policy defined declaratively outside the runtime.

All native system information, including CPU usage, is currently obtained using the kernel `/proc` filesystem. Calls are made using the mechanism already presented in Section 4.1.

Finally, a new package of classes was integrated in the GNU classpath in order for applications to specify their policies. These classes interact with the resource-aware underlying VM so that the application can add their own resource consumption policies, if needed. Nevertheless, policies can be installed with total transparency to the application. With this infrastructure, all consumable resources monitored, or directly controlled by the VM and class library, can be constrained by high-level policies defined externally to the VM runtime.

5. EVALUATION

The evaluation of our work focuses mainly on those mechanisms that can have a greater impact on performance, namely the checkpointing, restore, and migration mechanisms introduced in the Jikes RVM. These mechanisms are evaluated using a combination of the following: (i) synthetic micro-benchmarks to evaluate the performance and bottlenecks of individual mechanisms; and (ii) macro-benchmarks to evaluate the perceived impact of our solution in the execution of real life applications.

Micro-benchmarks: state extraction, checkpointing, restore, and migration
Currently implemented mechanisms have been tested regarding their performance. We created test programs that carry out a micro-benchmark of all internal components (state extraction and restoration, checkpoint, restore, and migration) presented in CRM-OO-VM architecture.

The results of micro-benchmarks depicted in Figures 5 and 6 were executed with several combinations of the three relevant parameters manipulated by the implemented mechanisms:

- Number of stack frames in the running application (50–300) for all tests.
- Number of heap objects referenced by those stack frames (50–300 for regular sized objects, 750–1250 for large objects, 50–300 for very large objects).
- Average size of the objects in the heap.

The times measured in these figures are expressed in seconds, unless otherwise noted, and are average values computed across multiple runs with outliers discarded on an Intel(R) Core(TM)2 Duo CPU T9300 @ 2.50GHz, with 1 GB RAM, in a local network with a transfer speed of 100 MB/s.

Three groups of tests were made in order to discover possible bottlenecks and evaluate the cost associated with the operation of each internal component. The first group evaluates the typical application with few data (first three samples in the graphs). The second group evaluates applications with more objects of larger size. Finally, the third group evaluates applications with few objects but of very large size. This allows us, in summarized form, to study the load caused by increasing numbers of objects and of increased size, both individually as well as combined.

From these results, some conclusions can be drawn. First, the results are very encouraging because the imposed latency is very small regarding the long execution times of the intended applications. Secondly, applications with large graphs of objects referenced from the stacks only suffer performance penalties in the state extraction component. This can be observed in Figure 5, samples from

DOI: 10.1002/cpe

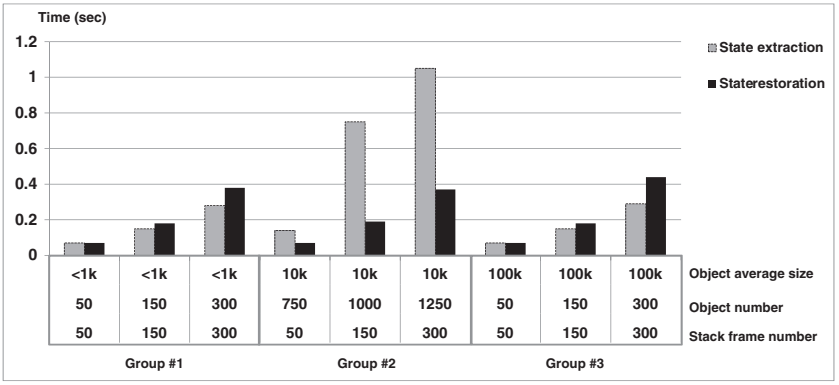


Figure 5. Internal component benchmarks: state extraction and restoration.

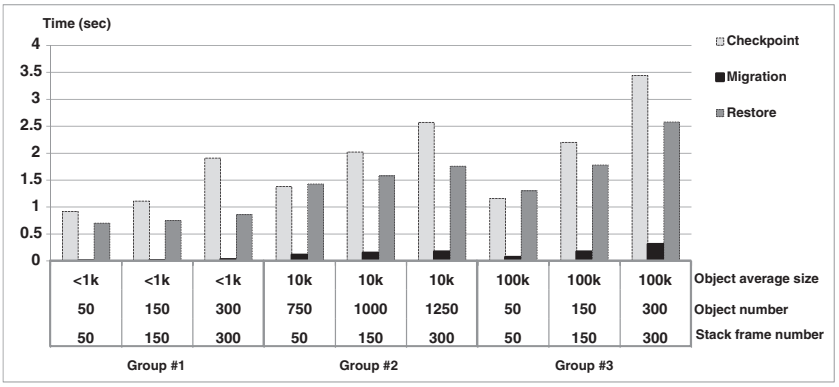


Figure 6. Internal component benchmarks: checkpoint, restore, and migration.

group number 2. The reason why this happens is that we need to perform type inference over stack frames. This still is a time consuming operation because we need to transverse the runtime type information of each analyzed object.

In Figure 6, serialization and de-serialization mechanisms (used in the checkpointing and restore components, respectively) caused the greatest overhead to the mechanisms, which is expected. For applications that have some large objects, checkpointing time rose above 2 s. For this reason, it is worth to explore an incremental/differential checkpointing approach as we intend to pursue. We highlight that, within a cluster setting, the actual cost of migration with data transfer is very reduced as can be observed in more detail in Figure 7. We need to highlight that the serialization mechanisms implemented in the GNU classpath have much lower performance than those in Sun JVM; thus, we believe these results can be further improved.

We also performed some tests on migration over the network to illustrate the usage of the implemented mechanisms in a large scale scenario (such as in Grid and Cloud computing), where jobs consisting of applications running on OO VMs can be checkpointed, restored, and migrated (or replicated), to more available nodes, possibly on a different location. These operations can be done without the complexity and overhead of having to checkpoint the entire operating system where the OO VM is executing (as it would be the case with System VM checkpointing).

The results presented in Figure 8 show, as it was expected, that the most significant source of overhead is migration itself, that is, the transfer of the checkpoint data. In order to reduce this, we compressed the checkpoint, and the checkpoint, compression, and migration combined took 25% less time.

DOI: 10.1002/cpe

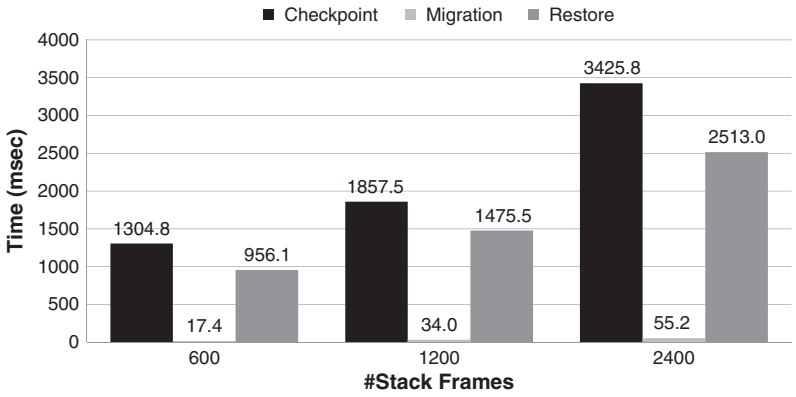


Figure 7. Checkpoint, restore, and migration execution for a large number of stack frames.

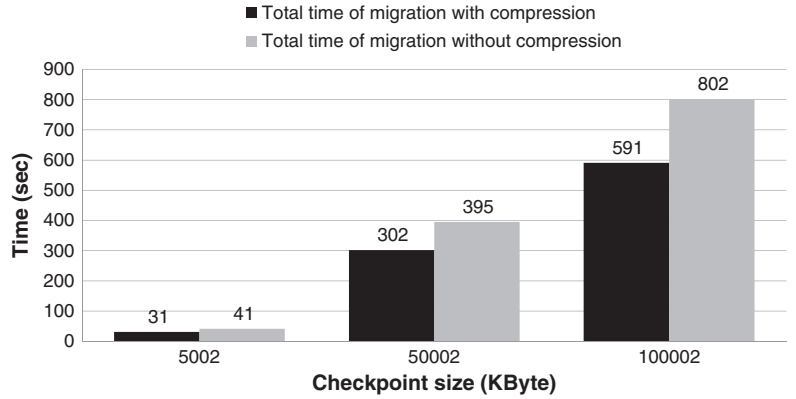


Figure 8. Migration, with and without compression.

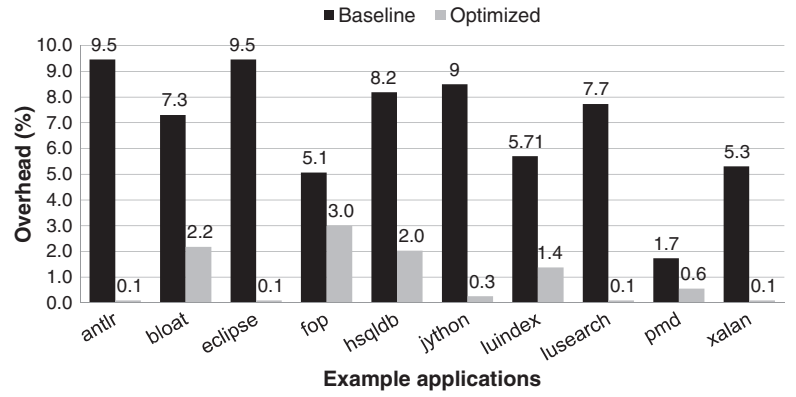


Figure 9. DaCapo benchmarks of the CRM-enabled virtual machine.

Macro-benchmarks: impact of solution on running applications

To evaluate the global overhead of our solution in the execution of real workloads, we used the DaCapo [35] benchmarks. The results presented in Figure 9 are very encouraging because the overhead is always below 10%, much lower than other solutions presented in Section 2 (e.g., [9–12]) that add an average 300% to the execution time of the applications. This overhead reflects the cost of updating internal information, during application execution, regarding the ownership of synchronization objects (as described in Section 4), which is needed when checkpointing is activated.

DOI: 10.1002/cpe

Figure 10. Application with a very long running time. SOR, successive over-relaxation.

Although this a constant cost, results show that the relative overhead to maintain this information is limited, with both compilers used.

Nonetheless, the applications used in the DaCapo benchmarks do not fully reflect applications with a long running time or within the scientific domain. For these reasons, the last series of tests regarding the CRM mechanisms were made in applications implementing the Successive Over-Relaxation (SOR) and the MonteCarlo Pi algorithms, both with a long running time. The results are presented in Figure 10. These applications use the same type of elementary tasks of e-Science applications, showing that any application with a long uptime period can benefit from checkpointing without a significant overhead. The setup for these tests was the following: for the SOR application, we used a matrix of 100×100 and 1×10^6 iterations; for the MonteCarlo Pi, the iteration count was 100×10^{12}. The checkpoint was performed every 5 min.

Because the time spent in the checkpoint operation is much smaller than the application total execution time, the total time of the SOR application, with and without checkpointing, differs in less than half a minute. For the MonteCarlo Pi, the difference is even smaller. These results demonstrate the contribution of our solution to a fast recovery in the presence of failures. That is, if a failure occurs during the execution of these long running applications, the last checkpoint could be used, avoiding starting execution from scratch.

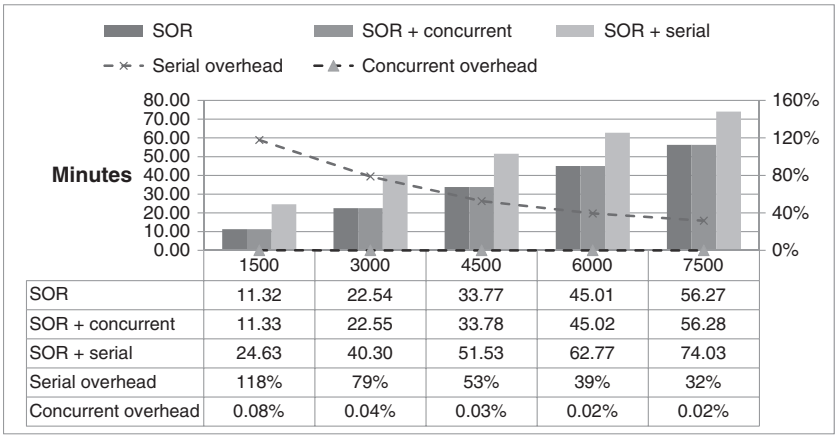
Concurrent checkpoint evaluation

To evaluate the concurrent checkpoint specifically, we set up two different checkpoint scenarios using SOR, which we identify as *Test 1* and *Test 2* checkpoints. Compared with the previous test using SOR (in Figure 10), these tests use a much larger array (instead of 100 equations) so that larger amounts of data need to be saved while keeping the number of iterations to 7500, for running times close to 2 h. We intend to show that with concurrent checkpointing available, it is possible and efficient to do checkpoints on larger applications and/or do it more frequently. Thus, for each of these classes of tests, SOR was run with a matrix of 3000, 3600, and 4200 equations. The two available cores were used to fully exploit the concurrent checkpointing. We averaged 5 executions of each test.

The distinguishing factor between these two types of tests is the event or reason triggering each checkpoint. In *Test 1*, the checkpoint is done when a percentage of the work is completed. In Figures 11, 13, and 15, checkpoint is done at 20%, 40%, 60%, and 80% of the computation progress. From this data, we conclude the following: (i) the overhead of concurrent checkpoint is negligible—less than 0.5% in all configurations; and (ii) the overhead of the serial checkpoint has a decreasing impact on the application's execution time as the number of total iterations increases. This evident decrease is because of the fact that as computation time increases, the fixed number of serial checkpoints taken (4) will have progressively smaller impact on the total execution time.

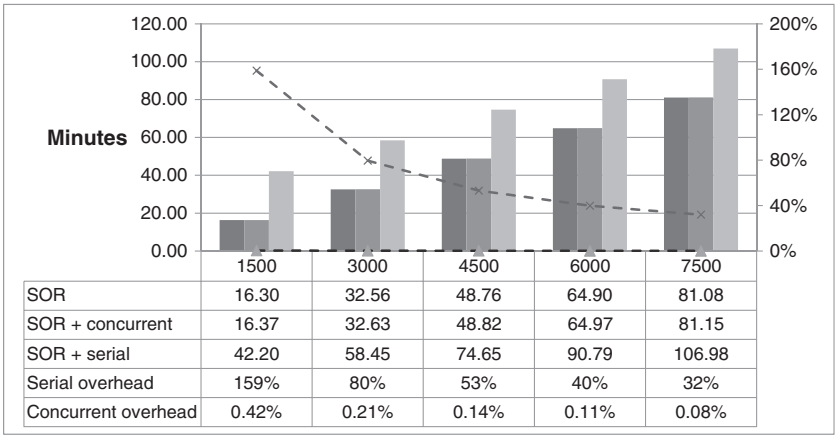
Nevertheless, as application total execution time increases, triggering checkpoint with percentage of progress may lead, in case of a failure, to significant loss of work performed and of data (i.e., all the computation done since the previous checkpoint and its results). Furthermore, the percentage of progress may be difficult to estimate in most applications and would require explicit checkpoint invocation by programmers.

DOI: 10.1002/cpe



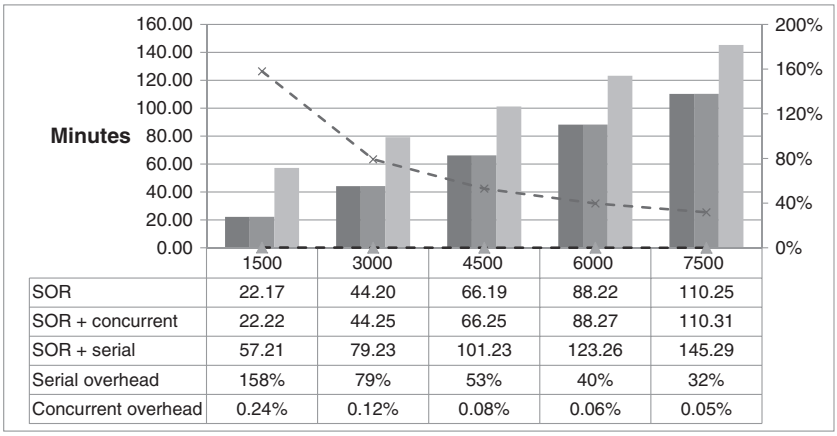
	1500	3000	4500	6000	7500
SOR	11.32	22.54	33.77	45.01	56.27
SOR + concurrent	11.33	22.55	33.78	45.02	56.28
SOR + serial	24.63	40.30	51.53	62.77	74.03
Serial overhead	118%	79%	53%	39%	32%
Concurrent overhead	0.08%	0.04%	0.03%	0.02%	0.02%

Figure 11. Test 1 - checkpoint - 3000 equations.



	1500	3000	4500	6000	7500
SOR	16.30	32.56	48.76	64.90	81.08
SOR + concurrent	16.37	32.63	48.82	64.97	81.15
SOR + serial	42.20	58.45	74.65	90.79	106.98
Serial overhead	159%	80%	53%	40%	32%
Concurrent overhead	0.42%	0.21%	0.14%	0.11%	0.08%

Figure 12. Test 2 - checkpoint - 3000 equations.

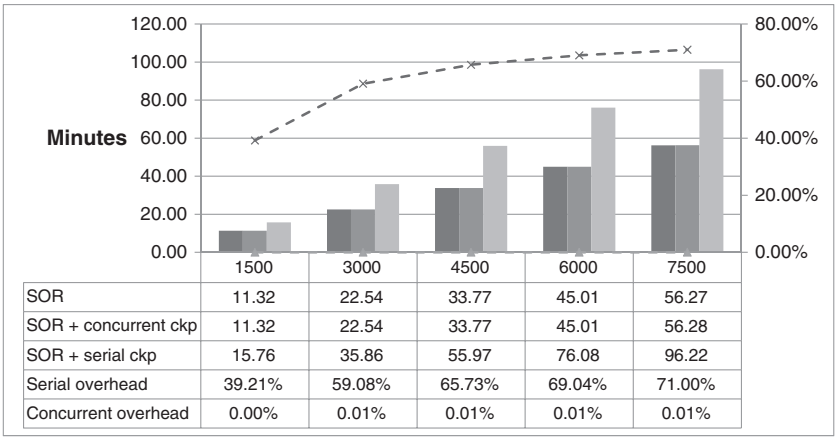


	1500	3000	4500	6000	7500
SOR	22.17	44.20	66.19	88.22	110.25
SOR + concurrent	22.22	44.25	66.25	88.27	110.31
SOR + serial	57.21	79.23	101.23	123.26	145.29
Serial overhead	158%	79%	53%	40%	32%
Concurrent overhead	0.24%	0.12%	0.08%	0.06%	0.05%

Figure 13. Test 1 - checkpoint - 3600 equations.

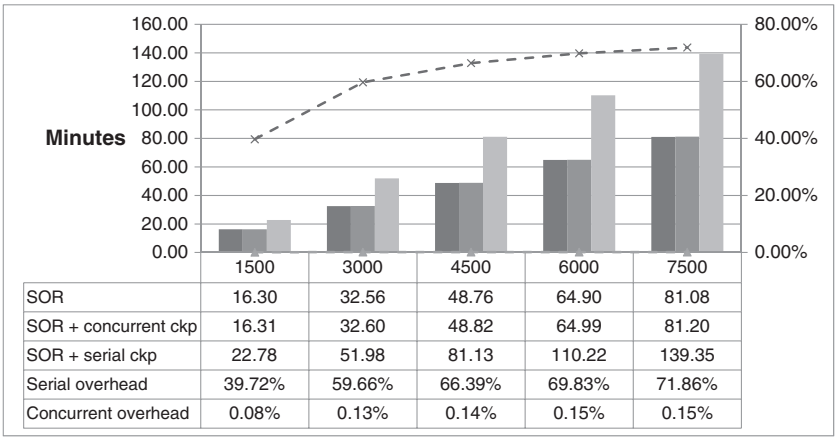
DOI: 10.1002/cpe

To avoid all these, the checkpoint should be triggered whenever a given time has elapsed, for example roughly every 5 min. This scenario is represented by *Test 2* checkpointing. Results are presented in Figures 14, 15, and 16. Here, because longer executions imply more checkpoints taken



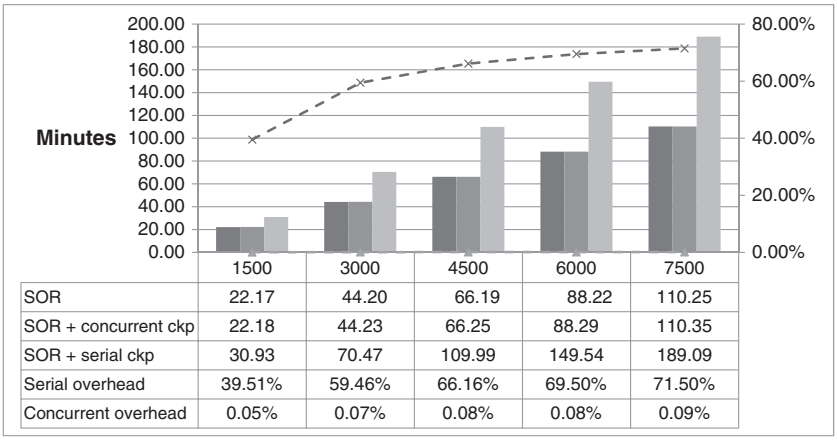
	1500	3000	4500	6000	7500
SOR	11.32	22.54	33.77	45.01	56.27
SOR + concurrent ckp	11.32	22.54	33.77	45.01	56.28
SOR + serial ckp	15.76	35.86	55.97	76.08	96.22
Serial overhead	39.21%	59.08%	65.73%	69.04%	71.00%
Concurrent overhead	0.00%	0.01%	0.01%	0.01%	0.01%

Figure 14. Test 2 - checkpoint - 3600 equations.



	1500	3000	4500	6000	7500
SOR	16.30	32.56	48.76	64.90	81.08
SOR + concurrent ckp	16.31	32.60	48.82	64.99	81.20
SOR + serial ckp	22.78	51.98	81.13	110.22	139.35
Serial overhead	39.72%	59.66%	66.39%	69.83%	71.86%
Concurrent overhead	0.08%	0.13%	0.14%	0.15%	0.15%

Figure 15. Test 1 - checkpoint - 4200 equations.



	1500	3000	4500	6000	7500
SOR	22.17	44.20	66.19	88.22	110.25
SOR + concurrent ckp	22.18	44.23	66.25	88.29	110.35
SOR + serial ckp	30.93	70.47	109.99	149.54	189.09
Serial overhead	39.51%	59.46%	66.16%	69.50%	71.50%
Concurrent overhead	0.05%	0.07%	0.08%	0.08%	0.09%

Figure 16. Test 2 - checkpoint - 4200 equations.

DOI: 10.1002/cpe

(with 5 min periodicity), the serial checkpoint now increasingly stretches the total execution time of the application (up to 70% more, broadly), whereas the overhead introduced by the concurrent checkpoint always remains very low.

So, to applications that need frequent checkpoints, given their longer total execution time and larger working set size, the concurrent checkpoint is a very effective alternative. Furthermore, given that all approaches described in the literature are serial in nature, their performance would always be much worse than our new proposal, added to the fact that they also lack on transparency and completeness, namely (i) either imposing the usage of an API, (ii) requiring extension of class code by programmers, or (iii) not supporting multithreaded and cooperative synchronized applications.

6. CONCLUSION

Today, more and more applications in e-Science fields (Chemistry, Bio-informatics) are developed in Java. They usually have long execution times and process vast amounts of data. When they fail during long executions, all performed work is lost, unless programmers explicitly implement some form of intermediate save of results already calculated. However, they are often designed by non-computer scientists, and such an explicit approach is frequently limited and incomplete and must be re-implemented each time over.

In this paper, we described a solution to these problems (CRM-OO-VM) by extending a JVM with checkpointing (serial and concurrent), restore, and migration mechanisms that can be employed with transparency to the programmers that need not modify their applications. The proposed solution was implemented, and we evaluated its adequacy and performance, with encouraging results.

In the future, we intend to test our solution in more demanding scenarios of load balancing across clusters and investigate the adoption of a similar approach in the context of .NET-related VMs.

ACKNOWLEDGEMENT

This work has been supported by FCT (INESC-ID multi annual funding) through the PIDDAC Program funds, and FCT projects PTDC/EIA-EIA/108963/2008, and PTDC/EIA-EIA/113613/2009.

REFERENCES

1. Holland RCG, Down TA, Pocock MR, Prlic A, Huen D, James K, Foisy S, Dräger A, Yates A, Heuer M, Schreiber MJ. Biojava: an open-source framework for Bioinformatics. *Bioinformatics* 2008; **24**(18):2096–2097.
2. Gront D, Kolinski A. Utility library for structural Bioinformatics. *Bioinformatics* 2008; **24**(4):584–585.
3. López-Arévalo I, Bañares-Alcántara R, Aldea A, Rodríguez-Martínez A. A hierarchical approach for the redesign of chemical processes. *Knowledge and Information Systems* 2007; **12**(2):169–201.
4. Sérgio Anibal de Carvalho J. Sequence alignment algorithms. *Master's thesis*, School of Physical Sciences & Engineering, King's College London, September 2003.
5. João Nuno S, Paulo F, Luís V. Service and resource discovery in cycle-sharing environments with a utility algebra. *International Symposium on Parallel & Distributed Processing*, 2010; 1–11.
6. Milojicic DS, Douglis F, Paindaveine Y, Wheeler R. Process migration. *ACM Computing Surveys* 2000; **32**:241–299.
7. Bradford R, Kotsovinos E, Feldmann A, Schiöberg H. Live wide-area migration of virtual machines including local persistent state. *Proceedings of the 3rd international conference on virtual execution environments - VEE '07*, 2007; 169–179.
8. Cabri G, Leonardi L, Quitadamo R. Enabling Java mobile computing on the IBM Jikes research virtual machine. *Proceedings of the 4th international symposium on Principles and practice of programming in Java*, 2006; 62–71.
9. Sekiguchi T, Masuhara H, Yonezawa A. A simple extension of Java language for controllable transparent migration and its portable implementation. *Coordination Models and Languages*, 1999.
10. Sakamoto T, Sekiguchi T, Yonezawa A. Bytecode transformation for portable thread migration in Java. *Lecture Notes in Computer Science* 2000; **1882**:16–28.
11. Truyen E, Robben B, Vanhaute B, Coninx T, Joosen W, Verbaeten P. Portable support for transparent thread migration in Java. *Lecture notes in computer science* 2000; **1882**:29–43.
12. Bouchenak S, Hagimont D. Pickling threads state in the Java system. *Third European Research Seminar on Advances in Distributed Systems*, 1999.
13. Ffinfrocken S. Transparent migration of Java-based mobile agents. *Springer* 1998; **147**:26–37.
14. Ferreira P, Veiga L, Ribeiro C. Obiwan: design and implementation of a middleware platform. *IEEE Transactions on Parallel and Distributed Systems* 2003; **14**(11):1086–1099.

DOI: 10.1002/cpe

15. Alpern B, Attanasio CR, Barton JJ, Burke MG, Cheng P, Choi JD, Cocchi A, Fink SJ, Grove D, Hind M, *et al*. The Jalapeno virtual machine. *IBM Systems Journal* 2000; **39**(1):211.
16. Suezawa T. Persistent execution state of a Java virtual machine. *Proceedings of the ACM 2000 conference on Java Grande*, 2000; 160–167.
17. Agbaria A, Friedman R. Virtual-machine-based heterogeneous checkpointing. *Software: Practice and Experience* 2002; **32**(12):1175–1192.
18. Illmann T, Krueger T, Kargl F, Weber M. Transparent migration of mobile agents using the Java platform debugger architecture. *Lecture Notes in Computer Science* 2001; **2240**:198–212.
19. Acharya A, Ranganathan M, Saltz J. Sumatra: a language for resource-aware mobile programs. *Lecture Notes in Computer Science* 1997; **1222**:111–130.
20. Bouchenak S, Hagimont D, Krakowiak S, De Palma N, Boyer F. Experiences implementing efficient Java thread serialization, mobility and persistence. *Software: Practice and Experience* 2004; **34**(4):355–393.
21. Suri N, Bradshaw JM, Breedy MR, Groth PT, Hill GA, Jeffers R. Strong mobility and fine-grained resource control in NOMADS. *Lecture Notes in Computer Science* 2000; **1882**:2–15.
22. Lau FCM. JESSICA2: a distributed Java virtual machine with transparent thread migration support. *Proceedings of the IEEE International Conference on Cluster Computing*; 381–388.
23. Ma RKK, Wang CL, Lau FCM. M-JavaMPI: A Java-MPI binding with process migration support. *The Second IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002; 1–9.
24. Sweeney PF, Hauswirth M, Cahoon B, Cheng P, Diwan A, Grove D, Hind M. Using hardware performance monitors to understand the behavior of Java applications. *Proceedings of the Third USENIX Virtual Machine Research and Technology Symposium*, 2004; 57–72.
25. Binder W, Hulaas J, Moret P, Villazón A. Platform-independent profiling in a virtual execution environment. *Software Practice and Experience* 2009; **39**:47–79.
26. Czajkowski G, Hahn S, Skinner G, Soper P, Bryce C. A resource management interface for the Java platform. *Software Practice and Experience* 2005; **35**:123–157.
27. Suri N, Bradshaw JM, Breedy MR, Groth PT, Hill GA, Saavedra R. State capture and resource control for Java: the design and implementation of the aroma virtual machine. *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1, JVM'01*, USENIX Association, Berkeley, CA, USA, 2001; 11–11.
28. Back G, Hsieh WC, Lepreau J. Processes in kaffeos: isolation, resource management, and sharing in Java. *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, California, 2000; 333–346.
29. Czajkowski G *et al.* Java specification request 284—resource consumption management API, 2009.
30. Backway P *et al.* The vision for networks, data storage systems and compute capability, January 2006.
31. Stevens RW, Rago SA. *Advanced Programming in the UNIX(R) Environment*, 2nd Edition. Addison-Wesley Professional: Upper Saddle River, NJ, 2005.
32. Fink SJ, Qian F. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the international symposium on code generation and optimization: feedback-directed and runtime optimization,*CGO '03. IEEE Computer Society: Washington, DC, USA, 2003; 241–252.
33. Quitadamo R, Leonardi L. The issue of strong mobility: an innovative approach based on the IBM Jikes research virtual machine. *PhD thesis*, University of Modena and Reggio Emilia, 2008.
34. Tanenbaum AS. *Modern Operating Systems*, 3rd edition. Prentice Hall Press: Upper Saddle River, NJ, USA, 2007.
35. Blackburn SM, Garner R, Hoffman C, Khan AM, McKinley KS, Bentzur R, Diwan A, Feinberg D, Frampton D, Guyer SZ, Hirzel M, Hosking A, Jump M, Lee H, Moss JEB, Phansalkar A, Stefanović D, VanDrunen T, von Dincklage D, Wiedermann B. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. ACM Press: New York, NY, USA, October 2006; 169–190.

DOI: 10.1002/cpe