CRM-HLL-VM: A Checkpointing-enabled Java VM for Efficient and Reliable e-Science Applications in Grids*

Tiago Garrochinho
INESC-ID / IST
Av. Prof. Cavaco Silva
Oeiras, Portugal
tiago.garrochinho@ist.utl.pt

ABSTRACT

Object-oriented programming languages are in current days, the dominant paradigm of application development (mostly Java and .NET languages). Recently, 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 one of those applications is terminated abruptly due to a failure (regardless of it being caused by hardware of software fault, lack of available resources,...), all of its work already carried out is simply lost and, when the application is later re-executed, 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.

A possible solution to solve these problems, is through mechanisms of checkpoint and migration. This makes applications more robust and flexible by being able to move to other nodes, without intervention from the programmer. This article provides a solution to Java applications with long execution times, by incorporating such mechanisms in a Java VM (JikesRVM).

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Applications; D.4.5 [**Reliability**]: Checkpoint/restart; D.3.4 [**Processors**]: Run-time environments; D.3.2 [**Language Classifications**]: Object-oriented languages

Keywords

Virtual machines, checkpointing, migration, Java VM, e-Science

*Expanded version of the article published in MGC2010, Middleware for Grids, Clouds and e-Science, integrated in the conference Middleware 2010.

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 more and more popular, albeit used by researchers (programmers) who are often not computer engineers/computer scientists. Relevant examples include chemistry, computational biology and bio-informatics [11, 10, 14], with many available Java-based APIs (e.g., Neobio).[1]

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

A possible solution to solve these problems, is through mechanisms of checkpoint and migration. With these mechanisms, an application becomes more robust, as it is fault tolerant and gains flexibility by being able to move to other nodes, without intervention from the programmer.

Prior and existing mechanisms of checkpoint and migration are supported at different levels: 1) 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 [16]); 2) system virtual machine (System VM, e.g. Robert Bradford et al. [7]). These approaches are insufficient. They either: i) require to store/transfer information that is not on the application itself (e.g. informa-

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

tion on the operating system on which it runs), or ii) limit application portability. Therefore, as the majority of the object-oriented programming languages execute their applications on object-oriented virtual machines (OO VM, also known as HLL VM,[2] e.g. Java VM, .NET CLR), our solution proposes an approach to the checkpoint and migration mechanisms at this level.

There is already some research in the area of checkpoint and migration solutions at OO VM level, however, existing solutions are embedded in the context of mobile agents and for that, are very limited, i.e. only portray a single thread on a very limited and controlled environment (e.g. MobileJikesRVM [8]). Other solutions either: i) have efficiency problems (e.g. JavaGo [19], JavaGoX [18], Brakes [23], ITS [5], with performance penalty in application runs exceeding an average of 300%); or ii) pass the responsibility to the programmer (e.g. WASP [9]) that must cooperate with the checkpointing mechanisms, which limits transparency; or iii) 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 and sockets).

This article provides a solution to Java applications with long execution times, by incorporating checkpoint and migration mechanisms in a Java VM (JikesRVM [3]). The main objectives are focused on the problems of transparency and completeness. Our proposed solution takes into account the following set of properties:

- **Transparency**: The mechanisms should not be constructed in such a way that gives responsibilities to the programmer. So, no application changes are required. It is provided a controller program (command line input) which communicates with the applications 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 they were recovered using checkpoint or transported to another environment using migration.

- **Flexibility**: Although there are no mandatory responsibilities to the programmer, we propose an API[3] 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 rescue operation. In functional terms, the application continues its execution as 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 and sockets), state regarding native execution (JNI), and Java synchronization monitors. Note that it is not intended to store/carry the whole virtual machine as a block. It is intended to take only 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 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 particular operating system and architecture may be able to use checkpoints generated by the same VM compiled on other systems.

- **Efficiency**: The additional cost of performance when running the application should be minimal or none. The cost performance during the execution of the mechanisms should be proportional to size of running applications.

- **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).

This paper is organized as follows: in Section 2 we will overview the related work. In Section 3, we describe the architecture and general vision of the components of the proposed solution. Section 4 addresses 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 measure their efficiency. Section 6 closes the paper with some conclusions.

2. RELATED WORK

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, i.e. depending on the level of implementation, we can obtain checkpoint and migration of: **operating systems** (i.e., a complete machine or platform installation), **applications**, or **threads**. Regardless the level of implementation, the **execution state** which 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.

Regarding the internal state, the problem in general is more or less well addressed, however, although not explicitly stated, some of existing solutions require the execution environment

[2] High-Level Language Virtual Machine.
[3] Application Programming Interface.

to be well defined and controlled (e.g., mobile agents). 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].

Due to limited space we address specifically only the OO VM level. 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 Java VMs 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.

OO VM checkpoint and migration solutions can be further subdivided in 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 whole execution state. However, these solutions have problems of efficiency and 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 own internal code, introducing new features from libraries, providing checkpoint or migration. Examples of such solutions include: Merpati [21], OCVM [2], CIA [12], MobileJikesRVM [8], Sumatra [1], JavaThread [6], Nomads [22], ITS [5], Jessica2 [13].

- **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 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 [9], JavaGo [19], JavaGoX [18], Brakes [23], M-JavaMPI [15].

Additionally, some of these solutions, such as CIA [12] and M-JavaMPI [15], take advantage of the debugging library provided by the Java VM architecture, known as the JPDA,[4] 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.

Analysis: 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 solution we propose in this article.

Merpati [21] is an application checkpoint solution. It cannot deal with application threads already blocked prior to performing checkpoint, and it can not handle state that does not belong to the virtual machine, as is the case of native state. Our approach can deal with threads already blocked, and JNI-related[5] state is processed in such a way that it is not explicitly saved, but at the same time we ensure consistency of the virtual machine and application upon restore.

OCVM [2] is a checkpoint solution designed for applications at OO VM internal level, although, it is 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.

WASP [9] is a solution for the migration of mobile agents, but it 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 of the programmer to address limitations of the solution, regarding when and where checkpoint can be performed and data to be included in it. Our approach does not suffer from these limitations.

Lastly, M-JavaMPI [15] 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, both Merpati, WASP and M-JavaMPI have transparency problems (in 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).[6] 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.

3. ARCHITECTURE

In this section 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 and migration, and then with the internal architecture of the

[4] Java Platform Debugger Architecture.

[5] Java Native Interface.

[6] Message Passing Interface.

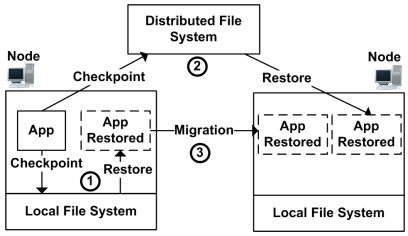

Figure 1: CRM-HLL-VM mechanisms usage overview.

extended VM. Figure 1 presents an overview of the features supported by this work:

1. Checkpoint to local disk and its corresponding restore.
2. Checkpoint for a distributed file system, and any node that is connected to this system, can restore the persisted state. This can be considered as an indirect migration and/or indirect replication, as more instances can be launched with the restored checkpoint and their replicas of files.
3. Migration between two nodes. The migration is done directly between two nodes.

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

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

  State extraction captures the execution state related with all threads within the application. Checkpoint has the obligation to stop all threads (to guarantee consistency), then calls state extraction and finally saves the state persistently into a file system. Migration calls 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 reconstruct and resume the execution of all stack frames[7] 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. Migration daemon is used in migration, and receives the state from the network. When the state is available, migration daemon calls restore with that state.

[7] A stack frame corresponds to a call to a subroutine which has not yet terminated with a return.

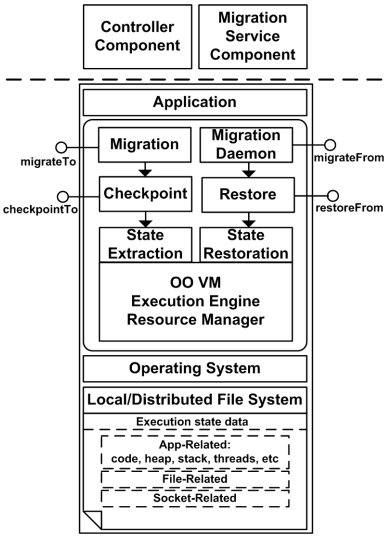

Figure 2: CRM-HLL-VM Architecture in one node.

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

- **Controller**: Command line program, which communicates with the application to take advantage of the mechanisms developed. With it, a user controls the mechanisms developed on an application, parameterizing it with commands/instructions depending on the desired result.
- **Migration Service**: Server 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 controller must discover the socket port that the application is listening (to receive `checkpointTo` or `migrationTo` trigger commands). Basically, the controller also listens in a specific socket port, and sends an operating system signal [20] to the application (using its process identification, e.g. Unix Process ID). Note that in the 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 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 and migration are being developed in the JikesRVM virtual machine (3.1.0). JikesRVM is a virtual machine designed to run Java programs, whose distinctive feature compared to other Java virtual machines is that is implemented in Java. Unlike other Java in Java virtual machines, JikesRVM does not need to rely on a second Java virtual machine to bootstrap and run.

In this section, we focus on implementation details that provide better understanding of the solution created. We describe how the execution state can be saved persistently on disk and transferred across network. Also, state extraction and restoration is detailed taking into account the consistency property.

Execution state: disk persistence and transfer. Most solutions discussed in related work at OO VM level, use a serialization mechanism supported by the virtual machine, 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 CLASSPATH`,[8] supported in JikesRVM, to persist and carry the information related to the execution state of a running application.

However, Java serialization requires that any class that must be serialized implements the `Serializable` interface. Thus, we would have to trust that all application classes 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 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 (operating system system-calls or native libraries dependencies) and cannot be automatically serializable. Because our solution addresses the issue of mobility of such objects, there is no need for application code to explicitly implement this or any other interface.

Additionally, if a class is not easily serialized (e.g. `RVMThread`[9] and internal synchronization locks, both with very strong environment dependencies), then we have two solutions:

1. Create an special externalized version of the object with the minimum required information (represented by primitive data types), that allows the reconstruction of the object on restore.

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. This type of solution was used for the thread synchronized state. We present more details in section 4 (Execution state: thread synchronized state).

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

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 an yield point. Although, 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 effectively safe, the virtual machine 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), but if they return some result to the virtual machine, they are blocked 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 only ones saved. Black fill marks the first frame to be saved.

A thread in a safe point has always the same first stack frame. Effectively safe threads do not, but every time a thread is effectively safe it enters into native code (internal VM code, not JNI) and is forced to save the FP[10] and the IP[11] pointers. This FP marks the first stack frame to be saved in a effectively safe thread.

Effectively safe threads are always restored in the same safe point. If a thread returns from its effectively safe state while

[8] http://www.gnu.org/software/classpath/

[9] Object that represents a thread within the virtual machine.

[10] Frame pointer, stack pointer that points to the last created frame.

[11] Instruction pointer, points to the next instruction to be executed.

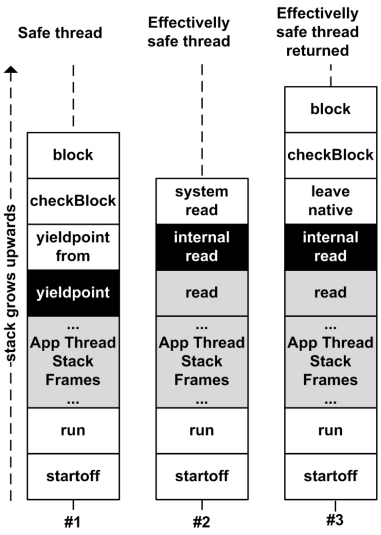

Figure 3: Safe and effectively safe thread examples.

in state extraction (Figure 3, transition from thread #2 to #3), then on restore it will appear as if it never advanced execution (it will look exactly like thread #2), which is the desired state.

Execution state: extraction. We are taking advantage of OSR (on-stack-replacement) 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.

JikesRVM has support for two compilers: baseline and optimized. Baseline stack frames are fully observable and easily extracted, but optimized are not. The optimized compiler chooses points in code where OSR can occur (points where important OSR maps are created), and for that, we currently only support baseline stack frame extraction. We are aware of the work made in [17] which disabled some optimizations without losing significative performance, but we also know that this solution only works 60% of the times. So, future work must me made to support 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 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 can be uninitialized at the current bytecode index. Additionally, the numbers of locals and stack operands are counted. When ready, the baseline extractor uses both type/number of locals and stack operands to retrieve the full data from the stack frame. The structure of the information retrieved is 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 responsibility to:

- Recover local and stack operand variables.
- 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 they were before, and when finished, the thread automatically blocks, to make the restore consistent. When all threads are ready, the virtual machine can restart with the previously checkpointed execution state.

Execution state: thread synchronized state. A thread within synchronized methods or statements can be in one of 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 always reacquires it again. Entry and wait set threads are walked back to a stack frame that on restore reruns the lock and wait code again, that 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.

Additional issues. 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 virtual machine.

Finally, current work is focused mainly on external state. This problem should not be a responsibility for the programmer, even because who uses the checkpoint/restore and migration mechanisms may not be the programmer itself. We want to take care of common external state like files and sockets.

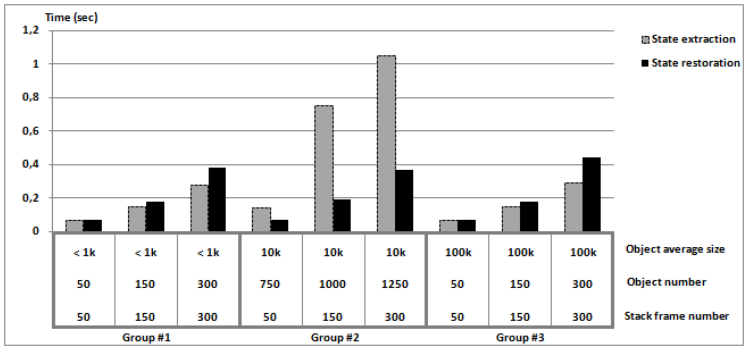

Figure 4: State extraction and restoration internal component benchmarks.

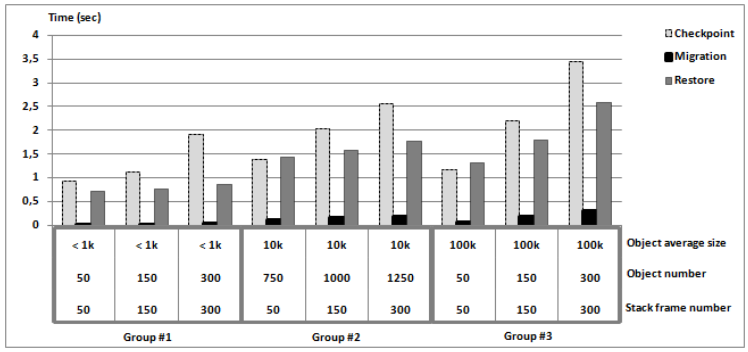

Figure 5: Checkpoint, restore and migration internal component benchmarks.

5. EVALUATION

Current implemented mechanisms have been tested for performance evaluation. We created test programs that carry out a microbenchmark of all internal components (state extraction and restoration, checkpoint, restore and migration)[12] presented in the CRM-HLL-VM architecture.

The results are depicted in Figures 4 and 5. The microbenchmarks 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 (representing individual atoms, molecules, particles, ...) 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 are expressed in seconds and are average values computed across multiple runs with outliers discarded, on a 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/sec.

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 large. This allows us, in summarized form, to study the load caused by increasing numbers of objects and of increased size, both individually and combined.

[12]Migration is the time taken to transfer the execution state between two different nodes.

From these results, some conclusions can be taken. First, the results are very encouraging since the imposed latency is very small regarding the long execution times of the intended applications; therefore, checkpointing can be performed often with significant gains in reliability and long-term performance in the presence of failures.

Secondly, applications with large graphs of objects referenced from the stacks only suffer performance penalties in the state extraction component (Figure 4, samples from group number 2). The reason why this happens is because we perform type inference over stack frames. This is very time consuming for lots of objects.

Finally, in Figure 5, serialization and de-serialization mechanisms (used in the checkpointing and restore components respectively) caused the greatest overhead to the mechanisms. For applications that have some large objects, checkpointing time rose above 2 seconds. For this reason, it is worth to explore an incremental/diferential 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.

Additionally, we performed some tests on migration 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. This, without the complexity and overhead of having to checkpoint the entire operating system instance where the OO VM is executing (as it would be with System VM checkpointing).

The results showed, as it was expected, that the most significant source of overhead is migration (70 second average time on migration on a 80 Kbit/sec line, for the same tests presented). In order to reduce this, we compressed the checkpoint and, checkpoint, compression and migration combined took 25% less time. Moreover, in Figure 6, the overhead during normal execution against original JikesRVM is under

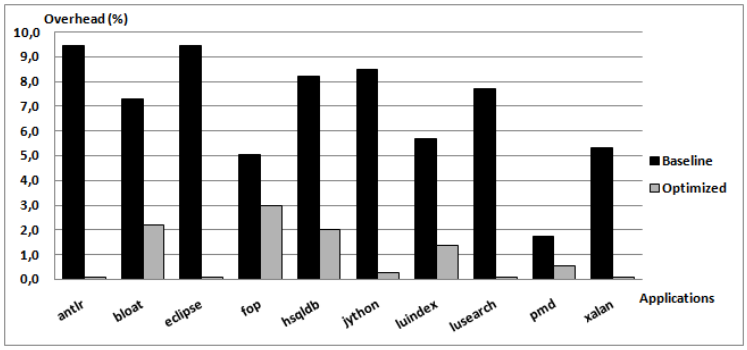

Figure 6: DaCapo benchmarks [4], overhead in runtime imposed by the checkpoint and migration mechanisms.

10% in all runs, for the baseline compiler. For optimized compiler (not yet supported, but we intend to), the overhead is even less, only under 5% in all runs.

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 must be re-implemented each time over.

In this paper, we described a solution to these problems (CRM-OO-VM) by extending a Java VM with checkpointing, restore and migration mechanisms that can be employed with transparency to the programmers which 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 integrate with better serialization mechanisms, give full support in extraction for optimized compiled methods and explore an incremental/diferential checkpointing approach. We also 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 virtual machines.

7. REFERENCES

[1] A Acharya, M Ranganathan, and J Saltz. Sumatra: A language for resource-aware mobile programs. *Lecture Notes in Computer Science*, 1222:111–130, 1997.

[2] Adnan Agbaria and Roy Friedman. Virtual-machine-based heterogeneous checkpointing. *Software: Practice and Experience*, 32(12):1175–1192, Outubro 2002.

[3] B. Alpern, C.R. Attanasio, J.J. Barton, M.G. Burke, P. Cheng, J.D. Choi, A. Cocchi, S.J. Fink, D. Grove, M. Hind, et al. The Jalapeno virtual machine. *IBM Systems Journal*, 39(1):211, 2000.

[4] S.M. Blackburn, R. Garner, C. Hoffmann, A.M. Khang, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S.Z. Guyer, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190. ACM, 2006.

[5] S Bouchenak and D Hagimont. Pickling threads state in the Java system. In *Third European Research Seminar on Advances in Distributed Systems*, 1999.

[6] S. Bouchenak, D. Hagimont, S. Krakowiak, N. De Palma, and F. Boyer. Experiences implementing efficient Java thread serialization, mobility and persistence. *Software: Practice and Experience*, 34(4):355–393, 2004.

[7] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. *Proceedings of the 3rd international conference on Virtual execution environments - VEE '07*, pages 169–179, 2007.

[8] G. Cabri, L. Leonardi, and R. Quitadamo. 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*, pages 62–71, 2006.

[9] S. Ffinfrocken. Transparent migration of Java-based mobile agents. *Springer*, Volume 147:26–37, 1998.

[10] Dominik Gront and Andrzej Kolinski. Utility library for structural bioinformatics. *Bioinformatics*, 24(4):584–585, 2008.

[11] 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.

[12] T. Illmann, T. Krueger, F. Kargl, and M. Weber. Transparent migration of mobile agents using the java platform debugger architecture. *Lecture Notes in Computer Science*, pages 198–212, 2001.

[13] F.C.M. Lau. JESSICA2: a distributed Java Virtual Machine with transparent thread migration support. *Proceedings. IEEE International Conference on Cluster Computing*, pages 381–388.

[14] 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.

[15] R.K.K. Ma, C.L. Wang, and F.C.M. Lau. M-JavaMPI: A Java-MPI binding with process migration support. *The Second IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 1–9, 2002.

[16] DS Milojicic, F Douglis, Y Paindaveine, and R. Process migration. *ACM Computing Surveys*, 2000.

[17] R. Quitadamo and L. Leonardi. *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.

[18] T Sakamoto, T Sekiguchi, and A Yonezawa. Bytecode transformation for portable thread migration in Java. *Lecture Notes in Computer Science*, pages 16–28, 2000.

[19] T Sekiguchi, H Masuhara, and A Yonezawa. A simple extension of Java language for controllable transparent migration and its portable implementation. *in Coordination Models and Languages*, 1999.

[20] Richard W. Stevens and Stephen A. Rago. *Advanced Programming in the UNIX(R) Environment (2nd Edition)*. Addison-Wesley Professional, 2005.

[21] T. Suezawa. Persistent execution state of a Java virtual machine. *Proceedings of the ACM 2000 conference on Java Grande*, pages 160–167, 2000.

[22] N Suri, J M Bradshaw, M R Breedy, P T Groth, G A Hill, and R Jeffers. Strong mobility and fine-grained resource control in NOMADS. *Lecture Notes in Computer Science*, pages 2–15, 2000.

[23] E Truyen, B Robben, B Vanhaute, T Coninx, W Joosen, and P Verbaeten. Portable support for transparent thread migration in Java. *Lecture notes in computer science*, pages 29–43, 2000.