A Progress and Profile-driven Cloud-VM for Resource-Efficiency and Fairness in e-Science Environments

José Simão
Instituto Superior de Engenharia de Lisboa
(ISEL) / INESC-ID Lisboa
jsimao@cc.isel.ipl.pt

Luís Veiga
INESC-ID Lisboa
Instituto Superior Técnico - UTL
luis.veiga@inesc-id.pt

ABSTRACT

Cloud platforms are becoming more prevalent in e-Science domains, also by encompassing new and existing Grid infrastructures into private, hybrid and federated clouds. Clouds are inherently multi-tenant as they run workloads from multiple users. Resources can be initially allocated statically, as for job scheduling in Grids previously, but they can also be changed elastically at runtime to meet the application effective needs.

When allocation needs to be changed, and resources are scarce, determining from which tenants resources must be taken to impact performance the least is a non-trivial and often deemed intractable problem, when outside the realm of batch scheduling and full prior information on resource requirements for each task, job, or VM instance.

In this paper we present a Java-based platform for cloud environments that is able to : i) monitor application progress with different levels-of-detail and allowing full applications transparency, ii) account and restrict resource consumption, such as CPU and memory, by applications, and iii) A cluster-wide and decentralized algorithm that, based on the progress of different workloads, can redistribute resources among different JVM instances. Evaluation shows it is able to improve resource-efficiency and fairness across e-Science private cloud infrastructures, by managing and migrating resources according to the previous criteria, driven by a number of novel proposed metrics inspired in Economics.

1. INTRODUCTION

Grid Computing flourished to a great extent due to its widespread adoption in many e-Science domains. Grids ease resource sharing, pooling and are amenable to both simple as well as sophisticated scheduling approaches. Currently, there is an analogous ongoing trend, this time to encompass new and existing Grid infrastructures into private, hybrid and federated clouds for e-Science. Clouds inherit the potential for resource sharing and pooling due to their inherent multi-tenancy support. But while in Grids, resource allocation and scheduling can be performed online, mostly based on initially predefined and static job requirements, in Clouds, resource allocation can also be changed elastically (up or down) at runtime in order to meet the application effective needs at each time, improving flexibility and resource usage.

Public cloud infrastructures and supporting middleware for private/hybrid clouds (e.g., eucalyptus, open-stack) offer APIs to allow explicit allocation and deallocation of instances, predominantly of system VMs that run full-fledged guest OS instance and applications in each instance. Deciding and dealing with this programming is cumbersome for e-scientists that are required to invoke cloud API besides writing their own codes. This can be partially mitigated in the relevant, yet specific, case of Bag-of-Tasks applications where multiple tasks can be successively assigned to a given VM, while the number of active VMs is managed automatically by the middleware [11].

When allocation needs to be changed, and resources are scarce, determining from which tenants resources must be taken to impact performance the least is a non-trivial and often deemed intractable problem, when outside the realm of batch scheduling and full prior information on resource requirements for each task, job, or VM instance. Other systems have addressed resource allocation in a shared or multi-tenant environment. Nevertheless they lack the notion of *resource effectiveness* in the sense that when there are scarce resources, there is no attempt to determine where to take such resources from applications (i.e. either isolation domains or the whole VM) where they hurt performance the least.

In this paper we present the current results on ARA-JVM , an *Adaptive and Resource-Aware Java Virtual Machine* for cloud environments that is able to: i) monitor application progress with different levels-of-detail and allowing full applications transparency, ii) account and restrict resource consumption, such as CPU and memory, by applications, and iii) a cluster-wide and decentralized algorithm, that based on the progress of different workloads, redistributes resources among different JVM instances. This paper extends our previous work presented in [13]. Ongoing evaluation of ARA-JVM shows improvements on resource-efficiency and fairness across e-Science private cloud infrastructures, by managing and migrating resources following the previous criteria, ruled by novel metrics inspired in Economics.

The rest of the paper is organized as follows. Section 2 frames our work with related work regarding resource allo-

SAC'13 March 18-22, 2013, Coimbra, Portugal.

cation and progress monitoring. Section 3 presents the economic rationale of the resource management strategies used in ARA-JVM and the enabling architecture to apply them. Section 4 discusses different metrics to measure progress with an increasing semantic quality. Section 5 discusses implementation issues regarding the core components of our system. Section 6 presents preliminary results. Finally, section 7 concludes the paper.

2. RELATED WORK

Because measuring application progress is an important step in any adaptation process there are several contributions on this topic, ranging from low level system information such as performance counters to information about the progress of specific variables inside applications and different types of execution environments and deployment scenarios.

Performance counter have been used to analyze object oriented applications [14, 5]. Nevertheless, these works do not attempt to adapt the behavior of the application or the high level VM as they focus only on the study of different workloads to better understand how major runtime components behave. The utilization of performance counters in full virtualized systems (using system level VMs) have similar problems because applications running in a guest VM will see information about the hypervisor instructions as their own [3].In [6], an application programming interface (API) is proposed to enable applications reporting progress through heartbeats. PowerDial [7] monitors the performance of applications using the Heartbeat framework. The system can dynamically adapt the application configuration (e.g. parameters given in the command line) in response to changes of load or power, threatening the ability to deliver results in effective time. In these cases, results will eventually be delivered with less accuracy. In ARA-JVM the progress information is used to transparently adapt the application execution runtime, restricting or giving more resources, without depending on the application parameters.

Task driven workloads, typical in grid infrastructures, must also be monitored by the execution runtime to adapt the relevant system parameters and achieve the desired goals (e.g. improve performance, save energy). Cushing et al. [1] propose a prediction-based framework to automatically scale the number of tasks running in scientific workflow management systems. The prediction of the number of tasks is based on the size of the input queues of each task and the data processing rate. Silva et al. [10, 11] focus on choosing the best number of hosts to run Bag-of-Tasks workloads, in an attempt to find a trade off between performance and cost effectiveness (regarding the host renting time). Their heuristics is based on the tasks execution time. These approaches not only require more expertise to organize programs but they are also sensible to long running workloads where finishing time among different tasks (or length of the input queue and the size of each element) can have large variations. Unlike Grid infrastructures, Cloud infrastructures depend on virtual machines to provide the two basic service models, either IaaS or PaaS. In [9], Mc Evoy et al. discuss implications of scheduling work in such environments showing the importance of knowing more about the workloads profile so that the execution environment can be adapted to provide improved performance.

3. AN ECONOMICS-INSPIRED MODEL

In this section we present the high-level motivation for our implementation work by referring to some Economics-inspired notions and variables. Then, we try to map them to a computational and cloud computing framing. In Economics, there are typically two major classes of variables that drive business performance or its *processing*: those related with: i) *Value* and its equivalent *output, revenue*, ii) those related with *input* and its associated *cost*. Depending on the specific kind of economic activity, revenue may be the value of sales of a shop or factory, or financial gains in banking, stock market, etc. Costs may be associated with labour, resources, raw materials, energy, investment, capital expenditures, etc.

Economists sometimes need to take into account non-direct monetary aspects such as externality, opportunity cost, risk, trade-offs in capital investment, etc. This leads to two inherent notions in Economics (even to those uninitiated) that: i) an activity consumes/costs resources and creates value, and ii) faced with limited resources, these should be geared towards the activities that at a given moment provide more return, and should be taken from activities where they will harm their return the least, i.e. commit and transfer resources in order to achieve a global positive (or maximized) *yield* from the whole process.

In many shared or multi-tenant infrastructures, such as private clouds, there may be no money and even when there is a credit-based system, we are left with more *down-to-earth* notions of *Progress* and *Resource Usage*. These are more easily comparable over time, and sometimes across applications and application classes. Resources usage is easily established while still open to some debate. Memory, CPU and storage are mostly obvious and should be accounted for. The notion of progress, while intuitive, is more elusive and application semantics dependent (we address this in Section 4). For the moment, let us consider progress as units of work carried out by the application. Additionally, we can also measure how progress and resource usage vary, at what rate, how fast, and determine the *effectiveness* by relating both.

Regarding *yield* we are mostly interested in determining how to identify two specific situations: i) when an application is making reduced progress due to resource shortage and could use more resources effectively, and ii) when an application is not taking full advantage of its resources and could make similar progress with fewer resources. Obviously, we want to transfer resources from applications in ii) (least effective first) to applications in i) (more hurt first).

This should be performed incrementally based on derivative of progress and resource consumption over time. Therefore, we are immune to different ways of measuring progress, resources usage and profiles across applications. This trade-off can be measured by comparing percentage variation of progress (%dP) against percentage variation of resource usage (%dR) and establish a ratio between them ($\mathbf{\%dP \div \%dR}$), which represents our *yield*. This is the variable our algorithms will use to decide resource transfer. As decisions also affect the system, we restrict resources to take slots of 5% (of currently allocated), possibly repeated while the application yield does not change significantly and is ranked high by the algorithm.

This simple model allows the scheduling to make decisions, over a given time frame, of where resources should primarily be diverted to and from. It is very adaptable and

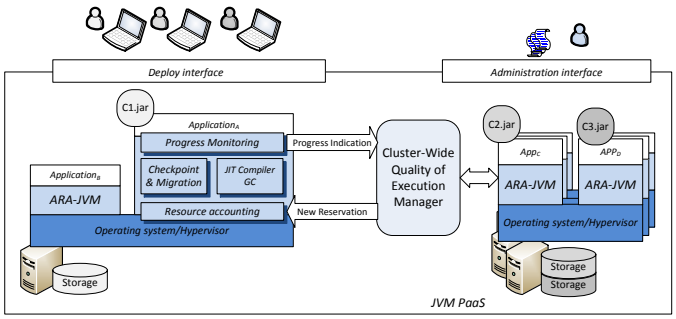


Figure 1: Overview of ARA-JVM system architecture

flexible as it is driven by differential, incremental measurements to detect and determine effectiveness of resource usage and make quick decisions on resource allocation transfer, restriction (in the extreme of application checkpointing or migration).

Empirically, applications experience several phases where they are more or less eager of resources, and others where they stabilize, or could even drop some with little impact (e.g. caching data elements no longer accessed). The key point is to approximately identify when this happens and transfer resources, when they are scarce, accordingly. The remaining of the paper deals with the architectural, algorithmic and implementation-related issues to bring these principles into practice, with efficiency and scalability, in a cloud computing infrastructure.

3.1 An Enabling Architecture

Figure 1 depicts the architecture of our proposal for a Platform-as-a-Service (PaaS) Java . There are two main elements in our approach. The first is ARA-JVM , which acts as the runtime execution environment. The second is a cluster-wide resource scheduler, which aims to allocate resources based on each application effective use and not on classic fairness criteria [8].

We believe that transparency of the whole monitoring and resource adaptation process is an important requisite. This is so because target developers (i.e. scientists using Java applications or writing their own using existing Java frameworks) will most likely be skeptic to the perspective of learning new programming interfaces to transfer their applications/frameworks to an environment such as the ARA-JVM PaaS. In order to comply with this requisite, each instance of ARA-JVM is enhanced with services that allow for: i) monitor the application progress, ii) account resource consumption, iii) reconfigure internal parameters and/or mechanisms, and iv) checkpoint, restore and migrate the whole application. In [12] we focus on the last point which regards checkpointing and restore. Our current design and implementation effort mainly concerns the progress monitoring module and the resource scheduler.

The monitoring information is periodically collected and reported to a cluster-wide resource scheduler. There are different types of progress metrics which we detailed in the next section.

4. PROGRESS MONITORING

There is a tradeoff between the *quality* of the monitoring information and the interference to the application in order to collect that information. We understand monitoring *quality* as a measure of how close the information is regarding the application effective progress. Generic mechanism (e.g. the OS API to monitor CPU usage) will be further away from the application semantics, while information collected by the runtime execution environment (e.g. number of calls to progress relevant methods) will be closer to the application semantics, and so, to the measurement of effective progress. While the former provides *low quality* information, the latter provides monitoring information with *high quality*. Our system takes the second approach. Nevertheless, information from different levels of the system could eventually be merged for processing which would allow for operations with different levels of transparency.

Some techniques are fully transparent to the application but also have low *quality*, for example, performance counters. On the other hand, others are more intrusive to the application development or deploying which makes them less transparent to the application but have higher *quality*, for example, enhancing the application bytecode to track method invocations. The following is a list of progress measuring techniques. The techniques are presented with an increasing level of monitoring quality but with a decreasing level of transparency from the application point of view.

Performance counters. Modern CPUs have built-in support to report how application progresses using *performance counters*. This information regards the lowest level of abstraction and also the one with less semantics regarding the application state or phase.

Operating system memory management. If a program can keep its working set in main memory it will progress faster unlike a program that generates a great amount of page faults. Allocation stalls, an event that occurs when a new page is requested and the OS must evict another's process page, can also contribute to a diminished progress.

Mutator utilization. Managed runtimes have to track running mutators to compare their execution time with time spent in garbage collection activities. This information can be used, for example, to determine how the heap size will grow or shrink. Bigger percentages of time spent in mutators means that the application can be making more progress.

Number of requests processed. This metric is typically associated with batch application like Bag-of-Tasks, or request-driven interactive applications, such as Web applications.

Code: instrumented or annotated. If information is available about the application high level structure, instrumentation can be used to dynamically insert probes at runtime, so that the system can measure progress using a metric that is semantically more relevant to the application.

For this work we analyze how annotated managed code can provide the relevant information to estimate the application progress.

5. IMPLEMENTATION

In this section we focus on implementation issues regarding how progress can be measured (Section 5.1), resources usage can be regulated inside a given runtime (Section 5.2) and how resources can be reallocated among runtimes (Section 5.3).

5.1 Progress monitoring framework

Annotations like the one presented in Figure 2.a are placed

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD,
         ElementType.FIELD,
         ElementType.PARAMETER})
public @interface Progress {
  double relevance() default 1.0;
}
```

(a)

```
public class AClass {
  @Progress(relevance=0.8)
  public void m1() { ... }
  public void m2(@Progress(relevance=0.2) p) {
    for (int i=p; i<limit; ++i) ...
  }
}
```

(b)

Figure 2: (a) The *progress* annotation (b) Usage of the *progress* annotation

by the programmer into strategic places of the code, where the application is known to make effective progress. This includes methods, fields and method parameters. The annotation is characterized by the relative contribution of the method to the overall application progress. Figure 2.b presents a code snippet with two usage examples. All annotations are used to insert progress measurement code at load time which will either count method calls or updates regarding fields and local variables dependent on parameters.

Regarding method calls, the progress measuring code updates a ARA-JVM structure with information regarding the *overall call rate* (OCR) which represents the call frequency since the application start. Periodically, ARA-JVM also calculates a *window call rate* (WCR) which is the call frequency in the last observation window (e.g. number of calls in the last 5 seconds). This helps to determine approximate derivatives of these values and to detect phase changes in application execution.

To process annotations at start time and insert the measuring code, ARA-JVM augmented the load process of a JVM, using an instrumentation Java agent, to look for annotations in the classes metadata as they are loaded into the VM. The agent transverses each class metadata and inserts code where necessary.

5.2 Resource Usage Monitoring and Enforcement

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 virtual machine can be either specific of the runtime (e.g. number of threads, number of objects), which we call *intrinsic resources*, or be strongly dependent on the underlying operating system (e.g. CPU usage), which we call *extrinsic resources*. To unify the management of such disparate types of resources, we have implemented JSR 284 - The Resource Management API [2, 4] - in the context of Jikes RVM.

Applications (directly or through a vendor library) can register for notification and setup new constraints using the appropriate classes available in our augmented GNU classpath. Given the specifications of the resource management framework, applications cannot widen the scope of the policy. For example, if a VM is configured to prevent the consumption of a certain number of file descriptors, the application can only further restrict this number. The VM can be configured during startup or in operation with a description of the resource management policy. The policy, composed by several rules, can be conveniently described using a XML-based file.

We consider resources to be any measurable computational asset which applications consume to make progress.

```
REPORT-NODE-INFO(nodeId)
 1  for each resource r ∈ R
 2       load_r = GET-RESOURCE-USAGE(r)
 3       add load_r to L[r]
 4       if VAR(L[R]) > 10%
 5            MULTICAST(nodeId, load_r)
 6  for each class c ∈ C
 7       for each application a ∈ Apps[c]
 8            progress_a = GET-APPLICATION-PROGRESS(A)
 9            resources_a = GET-APPLICATION-RESOURCES(A)
10            Y[c][a] = progress_a / resources_a
11  for each c ∈ C sort Y[c] in decreasing order
12  MULTICAST(Y)
```

(a)

```
DETERMINE-INFRACTING-APPLICATIONS(P, SLA, δ)
1  infracting = ∅
2  for each class c ∈ C
3       for each application a ∈ Apps[c]
4            if P[a].performance < SLA[a].performance + δ
5                 add a to infracting
6  return infracting
```

(b)

```
YIELD-BASED-RESOURCE-ALLOCATION(Infracting, Candidates)
1  sort Candidates in decreasing yield order
2  for each application a ∈ Infracting
3       c = top(Candidates)
4       remove set R of resources from c
5       donate R to a
```

(c)

Figure 3: (a) Report at each node the load and greatest *yields* (b) Determine applications in risk of violating their SLA (c) Determine cluster-wide resource transfer between applications.

Resources can be classified as either *explicit* or *implicit*, regarding the way they are consumed. *Explicit* resources are the ones that applications request during execution, such as, number of allocated objects, number of network connections, number of opened files. *Implicit* resources are consumed as the result of executing the application, but are not explicitly requested through a given library interface. Examples include, the heap size, the number of cores, network transfer rate. From our work point of view, *implicit* resources are the most relevant ones to control given that they have the ability to throttle the application progress, without leading to an abrupt stop (e.g. an exception because the number of files reqeuested are not allowed).

5.3 Cluster-wide resource management

The resource management strategy of ARA-JVM consists

of a number of algorithms that concur for efficient resource management. Two of them, whose performance and responsiveness is critical, can be executed in a fully distributed, decentralized and concurrent manner. They include two components for: i) dynamic decentralized and scalable load-balancing for quick response to resource outage, wasting, and QoS drop, and ii) resource monitoring and accounting. An extra algorithm, possibly centralized, can perform global cumulative calculation of overall utility, revenue, and penalties for QoS violations that can be ulteriorly fed to an optimization framework, e.g., linear programming or machine learning.

Globally, the system takes into account, to simulate an organization's structure and workload, a lattice C with a set of classes $C = \{Guest, Standard, Premium\}$ and a partial order $\leq$, that is, $C = \{Guest \leq Standard \leq Premium\}$. Classes are assigned to users according to their rank and/or payment level, and they are used to prioritize enforcing resource quotas and performance targets. We present next the three algorithms employed.

Figure 3.a shows the algorithm carried out by each node to report about its resource usage (e.g. when their variation exceeds 10%) and the yields of each application. Yields are reported in decreasing order, organized by each class in C.

The algorithm in Figure 3.b determines which applications must receive more resources in order to comply with the previously established service level agreement (SLA).

The algorithm in Figure 3.c is used to reallocate resources from candidate applications to application in risk of violating their SLAs. The list of candidates results from a yield based sorting, either merged from information received from each node, or calculated by a central component performing global sorting and returning updated candidate sets.

6. EVALUATION

In this section we evaluate our progress framework using both synthetic and real e-Science related workloads. We used machines in a cluster, with Intel(R) Core(TM) i7 Quad core processors (with eight cores each) and 16GB of RAM. Each machine was running Linux Ubuntu 12.04. The evaluation is divided in three parts: First, we evaluate the overhead of instrumenting classes at load time. Second, we show the overhead of monitoring progress during the application execution. Finally we shows results regarding the reallocation of resources.

To evaluate the load time overhead we used the SunFlow render system.[1] SunFlow uses ray tracing and projection techniques which are common in other applications for scientific visualization of data such as biological sequences (e.g. molecules, genes). For this test case, SunFlow is used without any source code modification. The Java instrumentation agent adds $105ms$ to the total time of the application. For the example file used, the rendering process had to load 137 classes which corresponds to an average overhead of $0.76ms$ for each class. This is a very small startup cost, even more so for such a long running application.

To assess the overhead of measuring progress during the application runtime we have have setup two experiences. The first one is a synthetic application in which there is a method performing 1000 write operations on a field. If only the method is annotated the total execution time is $1ms$ which compares with $45ms$ when the field is annotated, and so, each write operation is instrumented. In the second experience we made a single line modification to the SunFlow render system, adding an annotation to a method called for each *bucket* that finish rendering. When running the original code base, the rendering of a simple scene with a resolution of $1920x1080$ took $213s$. When running SunFlow with ARA-JVM and using the instrumentation described above, it took $214s$, which corresponds to an overhead of less than 0.5%. Because the instrumentation code is always the same regardless of the application, the larger the workload the smaller the percentage overhead added to the rendering process.

To evaluate how the resource allocation scheduling influences workloads we used three Java open source e-Science related applications/libraries: SunFlow, Xalan and Lucene. SunFlow, a photo-realistic rendering system, with a ray tracing core, was already presented. Xalan is an XSLT processor which transforms XML documents into other formats such as HTML or XMLs with different schemes. It implements the XPath W3C standard for addressing parts of an XML document. Frequently, scientists use well known information tools, such as ontologies, to model interactions between their elements of study, which are representable in XML documents. Transformations and queries on these documents can be made using Xalan or similar libraries. Lucene library provides a set of classes for documents indexing (*luindex*) and high-speed search (*luseach*). This is an important feature in a biologic sequences database. Sequences of genetic material are usually represented in a text/readable format (e.g. FASTA format is a text-based format for encoding DNA or protein sequences sequences), that often need to be updated with new samples and searched.

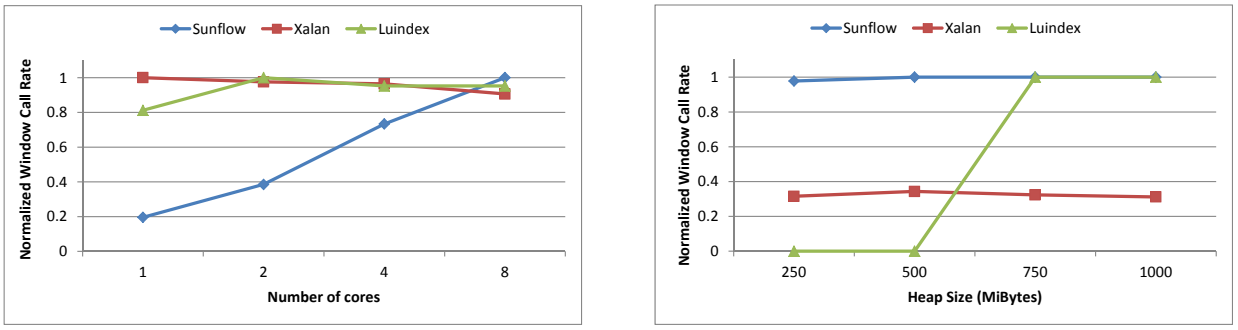
To understand how progress evolves with the allocation of different resources, SunFlow was used to render a 1920x1080 image, using 2040 buckets. Xalan converted a XML file using a complex XSL with 53 transformations. Lucene was used to generate index text files containing the ancestral sequences the Pongo abelii (an orangutan). Figure 4.a shows the average call rate (within windows of 5 seconds) of the three workloads varies using a different number of cores, while Figure 4.b shows the results for the same metric but with different heap sizes.

We can conclude that with call rates interpreted as measuring different progress by applications, they are inline with our predictions. While obviously all applications welcome having more resources, they not always take effective advantage from them, more so if we analyze the ratio between additional progress achieved derived from the additional resources. For instance, Lucene mostly always improves performance with any more resources, while SunFlow is more affected by CPU. Xalan progress is almost immune to extra resources, effectively wasting them, and those could be handed over to other applications.

7. CONCLUSIONS

Allocation of resources in multitenant infrastructures such as Grids and Clouds is mostly guided by service level agreements. Although physical resources, such as memory and CPU, are abundant is such environments, they are not illimitable. Therefore it is necessary to improve execution environments with mechanisms to measure the application progress to continuously exchange (more fine-grained) resource slices among virtual machines, awarding resources to

[1] http://sunflow.sourceforge.net/



(a) (b)

Figure 4: (a) Average window call rate for periods of 5 seconds and using a different number of cores (b) Average window call rate for periods of 5 seconds and using a different heap sizes.

those tenants requiring or entitled to more, while being able to determine where the resource reduction will be more economically effective, i.e., will contribute in lesser extent to performance degradation.

Managed runtimes are being used in large extent to develop new applications and libraries, as they have proven to be efficient, safe and easy to maintain. In e-Science environments this trend also applies. So, in this paper, after discussing some progress metrics and their rationale inspired in economics, we show how the enabling mechanisms can be applied to a managed runtime operating in a shared environment.

We presented the details of the ongoing work on ARA-JVM , including the progress monitoring and adaptation mechanisms in each VM. Furthermore, we experimentally show that not all applications take effective advantage from having more resources, and we are able to successfully transfer them to other applications that currently promise to make better use (progress) of them. In the future, we want to enrich the model with learning and profiling techniques as well as develop a library of typical progress monitoring patterns.

Acknowledgements: This work was supported by national funds through FCT - Fundação para a Ciência e a Tecnologia, under projects PTDC/EIA-EIA/108963/2008, PTDC/EIA-EIA/113613/2009, PEst-OE/EEI/LA0021/2011.

8. REFERENCES

[1] R. Cushing, S. Koulouzis, A. S. Z. Belloum, and M. Bubak. Prediction-based auto-scaling of scientific workflows. In *Proceedings of the 9th International Workshop on Middleware for Grids, Clouds and e-Science*, MGC '11, pages 1:1–1:6, New York, NY, USA, 2011. ACM.

[2] G. Czajkowski and T. von Eicken. Jres: a resource accounting interface for java. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 21–35, New York, NY, USA, 1998. ACM.

[3] J. Du, N. Sehrawat, and W. Zwaenepoel. Performance Profiling of Virtual Machines. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '11, pages 3–14, 2011.

[4] G. C. et al. Java Specification Request 284 - Resource Consumption Management API, http://jcp.org/en/jsr/detail?id=284, 2009.

[5] M. Hauswirth, P. F. Sweeney, and A. Diwan. Temporal vertical profiling. *Software Practice and Experience*, 40(8):627–654, July 2010.

[6] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *Proceedings of the 7th international conference on Autonomic computing*, ICAC '10, pages 79–88, 2010.

[7] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 199–212, 2011.

[8] J. Kay and P. Lauder. A fair share scheduler. *Commun. ACM*, 31:44–55, January 1988.

[9] G. Mc Evoy and B. Schulze. Understanding scheduling implications for scientific applications in clouds. In *Proceedings of the 9th International Workshop on Middleware for Grids, Clouds and e-Science*, MGC '11, pages 3:1–3:6, New York, NY, USA, 2011. ACM.

[10] J. Silva, L. Veiga, and P. Ferreira. Heuristic for Resources Allocation on Utility Computing Infrastructures. In *Proceedings of the 6th international workshop on Middleware for grid computing*, MGC '08, pages 9:1–9:6, New York, NY, USA, 2008. ACM.

[11] J. N. Silva, L. Veiga, and P. Ferreira. A^2ha - automatic and adaptive host allocation in utility computing for bag-of-tasks. *J. Internet Services and Applications*, 2(2):171–185, 2011.

[12] J. Simão, T. Garrochinho, and L. Veiga. A checkpointing-enabled and resource-aware Java Virtual Machine for efficient and robust e-Science applications in grid environments. *Concurrency and Computation: Practice and Experience*, 24(13):1421–1442, 2012.

[13] J. Simão, J. Lemos, and L. Veiga. A^2-VM a cooperative Java VM with support for resource-awareness and cluster-wide thread scheduling. In *Proceedings of the Confederated international conference on On the move to meaningful internet systems*, OTM'11, pages 302–320. Springer-Verlag, 2011.

[14] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of Java applications. In *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3*, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.