Resource-Aware Scaling of Multi-threaded Java Applications in Multi-tenancy Scenarios

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

Navaneeth Rameshan
Universitat Politècnica de Catalunya
rameshan@ac.upc.edu

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

***Abstract*—Cloud platforms are becoming more prevalent in every computational domain, particularly in e-Science. A typical scientific workload will have a long execution time or be data intensive. Providing an execution environment for these applications, which belong to different tenants, has to deal with the horizontal scaling of execution flows (i.e. threads) and an effective allocation of resources that takes into account the effective progress made by each tenant. While this is trivial for Bag-of-Tasks and embarrassingly parallel jobs, it is hard for HPC single-process multi-threaded applications because they cannot be scaled up automatically just by adding more virtual machines to execute the workload. In this paper we present *MengTian*[1], a distributed execution environment or platform capable of addressing the issues above. It encompasses several extensions to the Java execution environment, ranging from middleware to the virtual machine code and libraries. Our Java-based platform provides a Single System Image abstraction supported by a Partially Global Address Space to transparently spawn threads across a cluster of machines. It monitors progress with different levels-of-detail and accounts and restricts resource consumption. The overall goal is to redistribute resources among different JVM instances, increasing the unitary outcome of the progress *vs.* resource usage ratio over time.**

***Index Terms*—Resource scheduling, Progress monitoring, Managed runtimes**

I. Introduction

The Infrastructure-as-a-Service (IaaS) model is now widely adopted by the industry and the information community in general, either through public, private or hybrid clouds. In this service model, tenants (users that share a common infrastructure) are eased from managing the physical hardware and the hardships associated. They get a system *virtual machine* (VM), that runs full-fledged guest OS instance to execute their applications. Furthermore, they can change their needs over time, asking for more or less VMs, using provider-specific interfaces. Nevertheless, the IaaS model does not handle application horizontal scaling adequately in the case of single-process multi-threaded applications. These cannot be scaled up transparently just by launching or allocating more virtual machines to run the application. This may increase throughput (more instances running simultaneously) but it will not make any single instance execute faster.

[1]Meng Tian was a prominent general of the Qin dynasty. During this dynasty, a massive terra cotta army was build to honor the emperor.

Providing an execution environment for these applications, which belong to different tenants, has to deal with the horizontal scaling of execution flows (i.e. threads) and an effective allocation of resources that takes into account the effective progress made by each tenant. While this is trivial for Bag-of-Tasks and embarrassingly parallel jobs, it is hard for HPC single-process multi-threaded applications because they cannot be scaled up automatically just by adding more virtual machines to execute the workload.

Developing a mechanism to transparently scale multi-threaded applications and effectively allocating resources is a challenging task. To transparently scale applications, the execution environment has to deal with two main tasks: distribution of execution flows and share object graphs between distributed nodes. The algorithm to distribute execution flows must naturally take into account the available nodes and their load but also the correlation among different threads. Regarding shared memory, non thread-local modifications must be visible to all threads. The system must either automatically identify this shared state or ask the programmer to do so. Furthermore, the program lock semantics in concurrent accesses to shared memory must be enforced. Regarding the effective use of resources, the execution environment needs mechanisms to infer each workload's progress and to regulate resource consumption, in a workload specific way.

Our work crosses two very active research areas: scaling of applications and resource management. Regarding the former, several new programming models and languages [1], [2], [3], [4] have been proposed. Nevertheless they require the program to be bounded to yet another programming interface, which invalidates the use of previous working solutions by non programming-expert users, e.g. some groups of e-science researchers. Others have proposed to scale a virtual machine to the datacenter scale [5] but this would require new hardware architectures and a penalty to applications that do not need this kind of scalability. Differently from these approaches, we show that the widely known semantics of a Java program can indeed be extended to a distributed environment. Finally, these systems also do not account for scenarios of multi-tenancy where no single scheduling algorithm or resource allocation strategy has the best result for every workload.

In this context, we present *MengTian*, a Java-based platform for shared computer clusters, such as cloud environments.

It extends a distributed shared objects (DSO) middleware to provide a partial global address space with a thread spawning mechanism that has: a) two modes of transparency: one fully automatic, where the middleware can identify the shared elements of the program; the other lets the programmer identify them using annotations, for increased expressiveness and performance; b) two scheduling layers, either hybrid or centralized; c) resource-aware and workload-aware scheduling.

Furthermore, a set of extensions to the JVM are introduced to account for resource usage and progress awareness on a per workload basis. On top of these mechanisms we present a simple model that acts on thread spawning and resource accounting mechanisms, aiming to employ available resources where they are more effective.

As a consequence we: a) apply different scheduling heuristics, each targeted to suit a specific class of applications; b) allocate resources to workloads that exhibit superior incremental performance, when more resources are available to them. Evaluation results demonstrate the limited overhead introduced by the enabling mechanisms, while achieving significant improvements in performance, even with full transparency to programmers and users (i.e., neither changes nor access to source code).

The rest of the paper is organized as follows. Section II discusses the architecture of our system, showing the different abstraction layers we extended, and presents the relevant design issues of the thread spawning mechanism, and the extensions to the DSO middleware. Section III introduces the mechanisms *MengTian* provides for resource and progress monitoring and how they are used to determine if more resources will be used efficiently. Section IV presents the set of scheduling algorithms implemented. Section V discusses the results regarding the new mechanisms overhead and the gains obtained when running well-know Java workloads. Section VI presents related work and Section VII concludes the paper.

II. System overview

Figure 1 depicts the general overview of our proposal for a cloud Java execution environment, targeting shared computer clusters or clouds. There are 3 main elements in *MengTian* : i) scheduler capable of applying different strategies to each workload, ranging from thread scheduling to per workload resource allocation; ii) distributed middleware for object heap sharing that is organized as a partitioned global address space, and extended to support thread spawning in remote nodes; and iii) extended JVM capable of monitoring and accounting resource usage (e.g. number of cores, heap size) and exposing several JVM and application performance indicators (e.g. threads and objects correlation).

We have followed a co-designed approach combining middleware with necessary JVM extensions for control, access and raw performance requirements. However, the extended JVM and the enhanced DSO middleware can actually be used separately. The former can be used whenever it is really necessary to have a rich execution environment with several application progress sensors and resource accounting, while the latter does not depend on those extensions to operate. In the following, we give further details regarding each key aspect of *MengTian* , discussing the tradeoffs of the design.

DSO and Distributed Thread Scheduling.: Our thread distribution mechanism is based on the availability of a distributed shared objects (DSO) middleware for Java applications. We assume the basic services offered by the DSO layer (e.g., Terracotta from terracotta.org) include: i) a programming model where selected objects can be elected to be visible, i.e., shared across a cluster of machines; ii) allow concurrent access to shared objects, while upholding the usual consistency model based on Java monitors. This middleware enables the application to propagate changes to shared objects (which we will discuss next) but it is also used by the *thread spawning middleware* (see Figure 1) to support the thread distributed placement and the distributed reasoning for resource-aware scheduling.

The thread spawning middleware uses a master/worker approach. The master is responsible for starting the application and, for each new thread created, it launches remotely on a worker node. The worker exposes an interface for launching threads and provides all the operations supported by the base class library `Thread` class.

Semantics for objects sharing.: The class(es) associated with each thread can either use local objects or access (for read or write) objects that are shared in the cluster. Without any knowledge about the application behavior, the middleware has to use a pessimistic approach and ensure that every write to a field will be visible to the rest of threads. Although fully transparent, in case of false shares, this will result in an unnecessary overhead regarding network communication. To overcome this, there are two possibilities. One is to use code analysis techniques such as static escape analysis to the thread fields. The other is to let the programmer annotate the shared fields. In the former case, these techniques can also give some false shares, and can be less effective with dynamic code loading. The latter slightly breaks transparency. To accommodate these scenarios, *MengTian* provides two operation modes *Auto* and *Guided*.

Application performance indicators.: These rates express what is the dynamic behavior of the application. These indicators can be either collected from inside or outside the JVM. Examples include the number of instructions executed per machine cycle, the number of objects accessed per milliseconds, or the number of frames processed per second.

III. Progress Monitoring for Workload-Aware Scaling

Resource allocation scaling is an important mechanism to support workloads that can take advantage of extra resources. Nevertheless, in a multi-tenant shared infrastructure, it is necessary to compare the *opportunity cost* of having more resources allocated to a workload of a given tenant if workloads from other tenants can also make some progress. Because physical resources will eventually be limited, we must have mechanisms and metrics to measure and decide for which

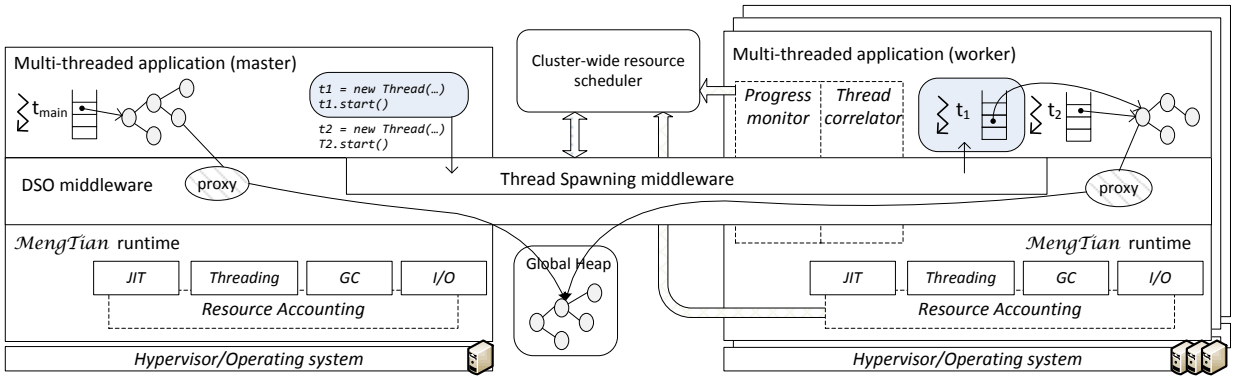

Fig. 1: Overview of *MengTian* system architecture

tenants resources are more effectively employed. To do so, there are three layers that must be in place. First, mechanisms to allocate and control resource usage. Second, mechanisms to determine the quality of progress of different workloads. Third, a control model to act on the allocation mechanisms based on these inputs.

In previous work [6], [7], [8] we have discussed and presented some results regarding the design and implementation of the two first mechanisms. Our current platform includes a Java execution environment that can enforce resource consumption policies and act over critical resources for managed applications, such as the heap growth/shrink policy. It is also aware of progress monitoring annotations that the programmer can use to identify critical functions of a program so that the runtime can handover resources to the workloads that make better use of them.

Regarding the third requirement, we propose in this paper to use an economic metric, *return on investment*, which is a ratio between *net profit* and *investment*, and apply it to our computational environment. As presented in Equation 1, this metric relates *speedup* and the *extra resources* employed to get them for workload w based on two observations j and $j-1$. When considering giving more work units (or resources in general) to entitled tenants, we are able to serve those with expected higher RoIs first, in an eager yet fast approach at maximizing the aggregated RoI.

$$RoI(w,j) = \frac{\frac{Progress_j}{Progress_{j-1}}}{WorkUnits_j - WorkUnits_{j-1}} \quad (1)$$

IV. Resource-Aware Scheduling Algorithms

When an application launches a thread, the master is responsible for making scheduling decisions based on the chosen heuristic. The worker can also make scheduling decisions if a thread spawns multiple threads. The middleware supports two types of scheduling and they are presented below.

A. Centralized Scheduling

In centralized scheduling, the decisions are taken entirely by a single node. Here, the master is responsible for making every scheduling decision. Based on the specified heuristic, the master selects a worker for remotely executing the thread and also maintains state information. The centralized scheduling heuristics supported are the classic *round-robin* and *resource-aware*, where scheduling decisions are made depending on the load of every worker. Regarding *resource-aware* scheduling, *MengTian* provides several variations:

CPU Load: The CPU load of every worker is monitored by the master and the threads are remotely launched on the worker with the least CPU load. The master maintains state information about the CPU load of every worker.

Load Average: Threads are scheduled on nodes with the least CPU utilization until the CPU load gets saturated. The scheduling heuristics then aims at equalizing the load average across the cluster. The values of load average obtained from the command-line top are not instantaneous. They are measured in three ranges as a moving average over one minute, five minute and fifteen minutes and the time taken for updating the moving average is five seconds. If multiple threads are launched instantaneously within a five second window, it is possible that all the threads are launched on the worker with the lowest load average. This problem is circumvented by an estimation of the number of threads to launch based on the processor queue of the worker.

Accelerated Load Average: This heuristic is similar to the scheduling heuristic Load-Average but is not as conservative and takes into account instantaneous changes in load average. It allows for scheduling the minimum number of threads possible while keeping the estimation correct and at the same time aiding in using a recent value of load average. The load information of CPU and load average is updated by the worker in one of the two ways:

- *On-demand* When an application is just about to launch a thread, the master requests all the workers to provide their current load. State information is updated only on demand from the master. This is a blocking update and it incurs an additional overhead of round trip time delay to every worker for every thread launch.
- *Periodic*: The load information of worker nodes are maintained by the master and updated periodically. The period required to perform updates is a configurable

parameter which can be chosen by the user. All updates are performed asynchronously and hence they do not block remote launching of threads. Optimal period for a worker is given by Equation 2:

$$p = \sqrt{\frac{t_l * (2 * t_m + RTT)}{2 * N}} \quad (2)$$

where t_l is the arrival time of the last thread, t_m is the time to monitor load by the worker, RTT is the round trip time to the distributed shared objects server and N is the total number of threads.

Thread Load: The master maintains state information about the number of threads each worker is currently running. This heuristic schedules in a circular fashion just like round robin until at least one thread exits. Once a thread exits, it ceases to behave like round robin and launches threads on nodes with least threads. The state information is updated only when a thread begins or finishes execution.

B. Hybrid Scheduling

Once a thread is scheduled to a worker, depending on the application, the thread itself may launch multiple internal threads. To handle such scenarios, the middleware also supports hybrid scheduling, where local copies of information that help scheduling decisions are maintained. The trade-off between consistency and performance is handled optimally for distributed scheduling.

In this approach, the master asynchronously sends the state information tables (global) to every worker before any thread launch. The workers store a copy of the table locally. Workers use this local table for making scheduling decisions after which they update the local table and then the global table. Once a worker updates its local table, there are inconsistencies regarding the information table among the workers. Nonetheless, they are lazily consistent and the final update on the global table is always the most recent and updated value. We achieve this by considering updates only to entries corresponding to that worker, in both global and local tables. This prevents updates to global table from blocking.

In this context, performance and consistency are inversely proportional to each other and we aim to improve performance by sacrificing a bit on consistency. If a worker has to schedule based on thread load and makes a choice by always selecting the worker with the least loaded node from its local table, then it could result in every worker selecting the same node for remotely spawning an internal thread, eventually overloading the selected node. This happens because the workers do not have a consistent view of the information table for a certain period. To prevent this problem, workers make their choice based on weighted random distribution.

C. Profiling Threads Behavior

Because there is no scheduling strategy that fits all kinds of workloads, we need to profile the application to determine its characteristics and apply the best algorithm. We considered four metrics to characterize an application: i) variations in thread's creation rate, ii) thread's workload imbalance, iii) total memory and iv) CPU usage. Thread's creation rate determines how frequently the application creates new threads. Thread imbalance represents how different is the workload assigned to each thread. Total memory represents all the memory (both the heap and non-heap) consumed for the process running a given JVM. Sections V-C relates the efficiency of the scheduling algorithm with each identified profile. Currently, profiling is done prior to executing the application on the middleware. It is not required to execute the application completely, instead sampling a part of the application is enough to obtain the necessary metrics. However, accuracy of the metrics are directly proportional to th amount of data sampled. Most accurate information about the metrics is obtained by executing applications till they finish.

V. Evaluation

The evaluation of our platform starts in with a micro-benchmark regarding the overheads of several extensions made to the execution environment (Sections V-A). Then we show that, for compute-intensive multi-threaded applications, speedups largely overcome these overheads. Section V-B also presents the *return on investment* for different workloads, showing how some workloads will benefit more than others, resulting from a resource up-scaling. Section V-C compares the impact of different scheduling heuristics. Evaluation was done using up to four 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.

A. Thread Spawning

Executing Java applications on the middleware may incur an additional overhead of increase in the size of bytecode and delay in the launching of each thread. Regarding periodic updates of load information, they are small and asynchronous; this masks any specific overhead incurred. For this reason, and to maintain generality, this section measures two main overheads: size of the bytecode, and delay in spawning a thread. To measure them, we developed a synthetic multi-threaded application, where each thread runs several times the MD5 cryptographic hashing function.

We observed an overhead of 5.04% in bytecode size due to additional instrumentations which provide for transparency, remote launching of threads, monitoring runnable objects and capturing thread execution times. Also, as the number of threads in the MD5 hashing application doubles, the total overhead incurred for launching threads also doubles. The overhead is considerable and indicates that the middleware is mostly suited for applications that are compute-intensive.

B. Execution Speedups

Multi-threaded applications that are compute intensive have the potential to fully exploit the benefits of our extended execution environment. To evaluate such a scenario, we used a micro-benchmarking workload and a complete application.

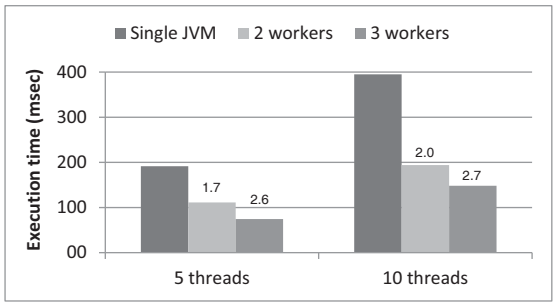

Fig. 2: Parallel MD5 hashing.

Fig. 3: Crawler for 20 and 30 websites.

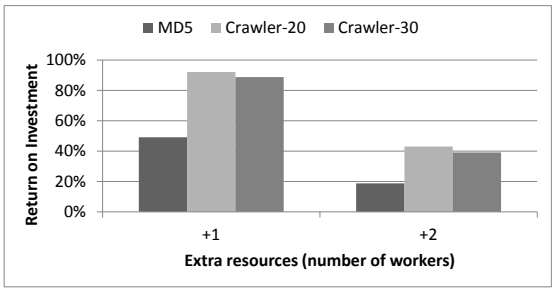

Fig. 4: RoI for MD5 and WebCrawler with increasing number of nodes

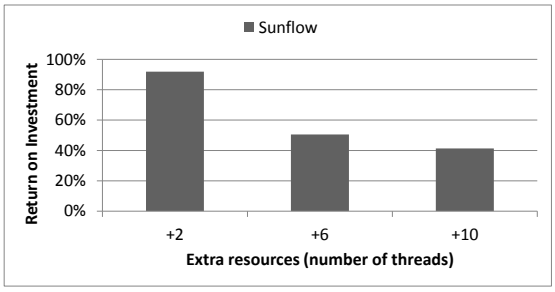

Fig. 5: RoI for Sunflow with increasing number of threads and nodes

The first spawns several threads, each using the MD5 hash algorithm to generate the hash of 500 messages. The second is a *web crawler* that spawns several threads to crawl a set of predefined sites. The third is SunFlow, [2] an Open Source rendering system for photo-realistic image synthesis, which represents a class of multi-threaded applications with coarse-grained interaction between threads.

For the micro-benchmark, the performance is compared by executing the MD5 hashing application on a single machine and using the middleware. Two and three workers are used for the purpose of comparison, and times taken with five and ten threads executing is measured. Figure 2 shows that when the number of workers increase, the time taken to execute the application decreases. There is positive scalability, and roughly linear. Regarding the web crawler, the number of websites crawled was increased for each evaluation. The number of threads for crawling within a single website is maintained as a constant at 3. These 3 threads require synchronization among themselves that may cross VMs. Every website is crawled up to a depth of two. Figure 3 shows the performance results, denoting also positive scalability, albeit sub-linear.

Figures 4 and 5 show how *returns on investment* can help users and administrators to configure thresholds, below which, elasticity is deemed ineffective and no extra resources are awarded (e.g., below 10%). As a law of diminishing marginal returns is expected (e.g., SunFlow from 6 to 10), thresholds can be configured for the RoI-deltas (e.g., less than 5 percent points) to detect this inelasticity earlier and prevent resource sub-efficiency.

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

C. Scheduling Algorithms

In this section, the different scheduling algorithms are evaluated with different application behaviors. All the experiments are done performing the MD5 hashing of multiple messages. To understand how different scheduling algorithms behave with different application characteristics, the following thread characteristics are modified in an sequenced fashion: i) variations in thread creation rate and thread inter arrival time, ii) thread's workload imbalance.

For each of these characteristics two orders of magnitude were considered: i) *small* and ii) *large*. When thread's workload imbalance is *small*, we assume threads to either compute *low* or *high* intensive workloads.

Table 6.a maps the letters used in Figures 6.b, 6.c and 6.d to the corresponding algorithm name. We first make threads perform a similar amount of computations with new threads being created after similar amounts of time. In other words, the thread's imbalance is *small* and variation in the thread's creation rate is also *small* (Fig. 6.b).

Round robin performs better because the threads have equal workloads and are equally spaced regarding their creation time. *CPU load on demand* incurs the overhead of obtaining the load information from every worker before making a decision. The results obtained for low thread workload are similar, except that the time taken for execution is considerably lower. We then make threads perform a similar amount of computations with threads being created after different amounts of time. In other words, the thread's imbalance is *small* and the variation in thread's creation rate is *large*. The results obtained for high and low thread workloads are shown in Figure 6.c.

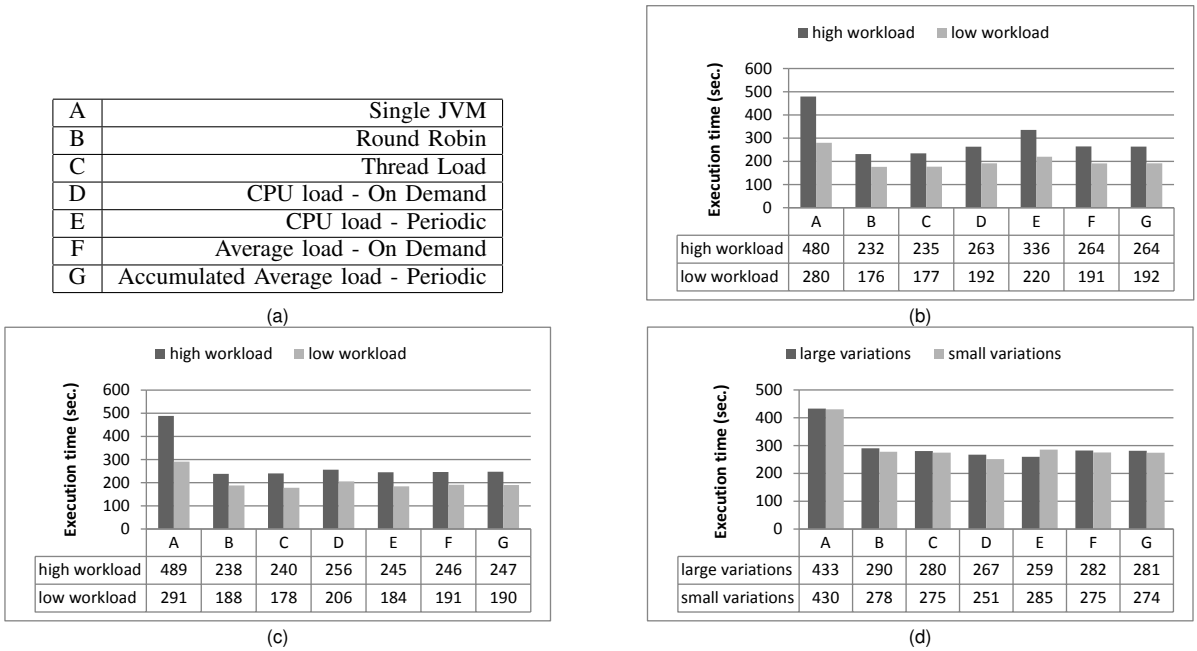

Fig. 6: (a) Legend used in evaluation figures (b) Impact of different workloads sizes for applications with *small* workload imbalance and *small* variations in creation rate (c) Impact of different workloads sizes for applications with *small* workload imbalance and *large* variations in creation rate (d) Impact of variations in thread's creation rate for applications with *large* workload imbalance

For high thread workload *CPU load periodic* finishes faster than CPU load on demand, unlike the previous scenario. The lowest period is enough time to update the state information asynchronously, as opposed to synchronous update for on-demand. For low thread workload, some of the threads finish execution before all threads are even scheduled, and the thread load heuristic is thus able to make a better decision than round robin. The overhead incurred by monitoring the CPU load increases the time taken to schedule threads and, hence *CPU load heuristic* performs worse than *round-robin* and *thread load heuristic*.

Finally, we considered an application with a *large* workload imbalance (i.e. different threads running with low and high intensive workloads). The results obtained for *small* and *large* variations in creation rate are shown in figure 6.d. For high variation in creation rates, the scheduling heuristic *CPU load periodic* consumes the least amount of time to finish execution. Although the scheduling heuristic has no information about the time taken to finish a thread, some threads finish much earlier, before all the threads are scheduled. This information helps the heuristic make a better decision, and spreads out threads of high and low workloads evenly among the different workers.

For low variation in creation rates, *CPU load on demand* performs better than any of the other scheduling heuristics, because the workloads of threads are unknown, and this heuristic aims to greedily equalize the CPU load of different workers, as and when threads are created. *CPU load periodic* takes a much higher time, as the lowest possible period to update the state information is indeed larger than the inter-arrival time between most of the threads. Most of the threads are hence scheduled on the same worker.

VI. Related Work

In this section we frame our work with other contributions regarding single system image and transparent scaling, resource awareness, progress monitoring and adaptability.

Cluster-aware virtual machines are JVMs built with clustering capabilities in order to provide a Single System Image (SSI). In Aridor et al. [9], threads and objects can be distributed without modifying the source or byte code of an application. To synchronize the objects across the cluster a master copy is maintained and updated upon every access which is a major bottleneck. Systems using standard JVMs are built on top of a DSM system to provide a Single System Image for applications. Some of the most popular systems are Java Party [10], Java Symphony[11] and JOrchestra[12]. Nevertheless, these systems either (i) require the programmer to explicitly deal with object management, defeating the advantage of a built-in garbage collection or (ii) lack the mechanisms to support resource and progress monitoring which makes them unsuitable for multi-tenancy scenarios.

Leitner et al. present CloudScale [4], a middleware to offload methods execution from general purpose Java applications to the cloud. Differently from *MengTian* , it relies on a wide-area client-side infrastructure that, based on configuration, transforms local calls into remote ones, and assumes no data sharing between components residing in the cloud. Kächele et al. [13] present an OSGi-inspired component

framework that automatically manages elastic deployment of applications already organized as OSGi-like components. *MengTian* can be used with advantage in applications not bound to the OSGi contract.

Several systems in the literature 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). 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 [14], 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.

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). In [15] two algorithms are proposed which, based on a performance model, can detect tasks with a bottleneck. The identified tasks are then duplicated to increase the throughput of a stream program. Cushing et al. [16] 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.

VII. Conclusions

In this paper we proposed *MengTian* , an improved distributed execution environment, extending the Terracotta middleware and Jikes RVM, with mechanisms to measure the application progress, and fine-grained resource usage, that can drive a metric of elasticity, inspired by the return-of-investment economic notion. This offers illusion of a single system image, with resource elasticity, and transparent access to an elastically scalable object heap and CPU pool, larger than any single physical machine. It is able to scale existing applications with ease and schedule threads efficiently across machines. It supports multiple scheduling heuristics, each best suited for a specific thread behavior in an application. Thread behavior is obtained by a profiling feature supported by the middleware. Based on the results obtained, a cpu-intensive application can be modeled based on the characteristics of the cluster and threads, and the best scheduling chosen.

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. Moreover we want to leverage the profiling in *MengTian* to detect thread competition/coordination/cooperation patterns by intercepting object synchronization in order to decide when to co-locate, with higher or lower density, threads of the same application in a given VM instance. This tradeoff between extra CPU load but lower object synchronization latency will also be subject to monitoring, profiling and optimization.

Acknowledgments: This work was partially supported by national funds through FCT – Fundação para a Ciência e a Tecnologia, projects PTDC/EIA-EIA/113613/2009, PTDC/EIA-EIA/108963/2008, PEst-OE/EEI/LA0021/2013.

References

[1] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," *SIGPLAN Notices*, vol. 40, no. 10, pp. 519–538, Oct. 2005.

[2] J. Su and K. Yelick, "Automatic support for irregular computations in a high-level language," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, ser. IPDPS '05. Washington, DC, USA: IEEE Computer Society, 2005.

[3] E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi, and J. Labarta, "A high-productivity task-based programming model for clusters," *Concurrency and Computation: Practice and Experience*, pp. 2421–2448, 2012.

[4] P. Leitner, B. Satzger, W. Hummer, C. Inzinger, and S. Dustdar, "Cloudscale: a novel middleware for building transparently scaling cloud applications," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, ser. SAC '12. New York, NY, USA: ACM, 2012, pp. 434–440.

[5] Z. Ma, Z. Sheng, L. Gu, L. Wen, and G. Zhang, "Dvm: towards a datacenter-scale virtual machine," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, ser. VEE '12. New York, NY, USA: ACM, 2012, pp. 39–50.

[6] J. Simão and L. Veiga, "QoE-JVM: An adaptive and resource-aware Java runtime for cloud computing," in *Proceedings of OTM Conferences*, ser. Lecture Notes in Computer Science, R. Meersman, H. Panetto, T. S. Dillon, S. Rinderle-Ma, P. Dadam, X. Zhou, S. Pearson, A. Ferscha, S. Bergamaschi, and I. F. Cruz, Eds., vol. 7566. Springer, 2012, pp. 566–583.

[7] ——, "A progress and profile-driven cloud-vm for resource-efficiency and fairness in e-science environments," in *Proceedings of Symposium On Applied Computing*, S. Y. Shin and J. C. Maldonado, Eds. ACM, 2013, pp. 357–362.

[8] L. Veiga and P. Ferreira, "PoliPer: policies for mobile and pervasive environments," in *Proceedings of the 3rd workshop on Adaptive and reflective middleware*, ser. ARM '04. New York, NY, USA: ACM, 2004, pp. 238–243.

[9] Y. Aridor, M. Factor, and A. Teperman, "cJVM: a single system image of a JVM on a cluster," in *Proceedings of the International Conference on Parallel Processing*, 1999, pp. 4–11.

[10] M. Philippsen and M. Zenger, "JavaParty - transparent remote objects in Java," *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1225–1242, 1997.

[11] T. Fahringer, "JavaSymphony: A system for development of locality-oriented distributed and parallel java applications," in *In Proceedings of the IEEE International Conference on Cluster Computing*. IEEE Computer Society, 2000.

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

[13] S. Kächele and F. J. Hauck, "Component-based scalability for cloud applications," in *Proceedings of the 3rd International Workshop on Cloud Data and Platforms*, ser. CloudDP '13. New York, NY, USA: ACM, 2013, pp. 19–24.

[14] 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*, ser. MGC '11. New York, NY, USA: ACM, 2011, pp. 3:1–3:6.

[15] Y. Choi, C.-H. Li, D. D. Silva, A. Bivens, and E. Schenfeld, "Adaptive task duplication using on-line bottleneck detection for streaming applications," in *Proceedings of the 9th conference on Computing Frontiers*, ser. CF '12, 2012, pp. 163–172.

[16] 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*, ser. MGC '11. New York, NY, USA: ACM, 2011, pp. 1:1–1:6.