Thread-level CPU and Memory Usage Control of Custom Code in Multi-tenant SaaS

Majid Makki, Dimitri Van Landuyt, Bert Lagaisse, and Wouter Joosen

imec-DistriNet
Department of Computer Science, KU Leuven
Leuven, Belgium
`firstname.lastname@cs.kuleuven.be`

Abstract. Software-as-a-Service (SaaS) providers commonly support customization of their services to allow them to attract larger tenant bases. The nature of these customizations in practice ranges from anticipated configuration options to sophisticated code extensions. From a SaaS provider viewpoint, the latter category is particularly challenging as it involves executing untrusted tenant custom code in the SaaS production environment. Proper isolation of custom code in turn requires the ability to control the CPU and memory usage of each tenant.

In current practice, OS-level virtualization tools such as hypervisors or containers are predominantly used for this purpose. These techniques, however, constrain the number of tenants that a single node can cost-effectively accommodate.

In this paper, we present a practical solution for thread-level tenant isolation, vis-à-vis CPU and memory usage in presence of tenant-provided custom code. Both Java Runtime Environment (JRE) bytecode and tenant code are instrumented with usage control checkpoints which, based on data gathered using the Java Resource Consumption Management API (JSR-284), ensures that CPU and memory usage of tenants remain within their Service-level Agreements (SLA) limits.

Our experiments show that the tenant accommodation capacity of single node increases 59 times with the proposed solution instead of containers. This scalability improvement comes at the average cost of 0.31 *ns* performance overhead per control checkpoint.

Keywords: SaaS, customization, multi-tenancy, resource management, tenant isolation, JRE.

1 Introduction

Multi-tenant systems share a single run-time environment among multiple customer organizations (tenants) and consequently require isolation mechanisms to avoid any interference or disturbance between tenants. For instance, tenant isolation mechanisms guarantees that tenant Service Level Agreements (SLA) are respected. In common practice, hypervisors and containers are employed which

perform tenant isolation at the level of virtual machines and OS processes respectively.

However, as shown in Figure 1, isolation can also be attained at the higher level of application threads which increases resource utilization. Allocating a separate Virtual Machine (VM) or OS process (cf. [1–4]) to each tenant imposes a constant memory overhead which reduces the number of tenants that a single node can accommodate.

Due to this constant memory overhead, other resources will be underutilized when some tenants are idle or almost inactive. Due to the

Fig. 1: Effects of enforcing tenant isolation at different levels.

lower per-tenant memory overhead of thread-level tenant isolation, the latter allows accommodating more tenants on a single node and increases resource utilization. Accomplishing tenant isolation at the application level in turn allows for a more cost-efficient deployment: more tenants can be provisioned with less resources.

Software-as-a-Service (SaaS) applications commonly allow tenants to customize the application behavior through custom code extensions (e.g. [5, 6]). For instance, in SaaS applications driven by business processes, the code of a certain activity in a workflow may be provided by a tenant instead of the SaaS provider (Business-process-as-a-service, BPaaS). Application platform as a Service (aPaaS) offerings similarly provide application services that can be open to code extension. In contrast to customizations that have been anticipated and developed by the SaaS providers, allowing tenant-provided custom code to be executed creates a number of non-trivial challenges. These code-based customizations are essentially untrusted and tenants may be incentivized to exploit the system, hinder other tenants, etc.

In a SaaS context, existing multi-tenancy approaches [7–10] commonly involve external, black-box measures and techniques to monitor (in terms of metrics such as throughput, requests/second, etc.) and control (e.g. throttling, SLA-aware job scheduling and load balancing) the services delivered to tenants. However, in presence of custom code provided by tenants, these techniques do not suffice. Enforcing tenant isolation at thread-level requires monitoring and controlling the CPU and memory usage of the custom code extensions inside the run-time environment.

In this paper, we present a modified version of the Java Runtime Environment (JRE) which enables running code provided by untrusted tenants in a single JRE instance while CPU and memory usage levels are kept within SLA boundaries. As opposed to experimental Java Virtual Machines (JVM) that target a similar goal [11–15], our JRE modification mechanism can be applied to any JRE used by industrialized JVMs. Our main contribution is, then, providing a portable mechanism for isolating tenants, vis-à-vis CPU and memory usage,

at the level of application threads.

Two challenging obstacles have been addressed. The first obstacle is that Java threads are directly managed by the OS kernel and it is not possible to pause/resume them once started using the JRE [16, 17]. The second obstacle is that no implementation of the Java Resource Consumption Management API (RCM API specified in JSR-284 [18]; cf. [19]) allows imposing limits on CPU or memory usage of threads. Existing implementations such as Oracle HotSpot JVM are only capable of reporting the CPU and memory usage of each thread.¹

Our modified JRE is based on the Oracle HotSpot JVM and is extensively based on Java bytecode instrumentation driven by call graph analysis [20], for two main purposes: (i) ensuring that tenants cannot evade usage control checkpoints which enforce SLA compliance, and (ii) forcing threads to sleep or stop when tenant custom code breaks out of the SLA boundaries.

We have validated our implementation using an illustrative batch-processing application and as such demonstrated that the CPU and memory usage levels can be controlled without using containers or VMs. Furthermore, the evaluation results show that the proposed solution increases tenant accommodation capacity with a factor of 59 compared to process-level solution, and only with a marginal average performance overhead of 0.31 *ns* per usage control checkpoint.

Section 2 formulates the problem at stake in terms of requirements. Section 3 presents the solution. Section 4 reports on the evaluation results. Section 5 discusses practical issues and draws perspectives for future work. Section 6 discusses related work and finally, Section 7 concludes the paper.

2 Problem Statement

This section demarcates the problem addressed by this paper as a set of requirements. The functional requirements are marked by F whereas quality requirements are marked by Q:

- F1** While executing the code provided by multiple tenant organizations, the CPU and memory usage of tenants should be kept below the limits specified in their SLA.
- F2** It is required to allow tenants to use CPU as much as possible within SLA limits.²
- Q1** The number of tenants that can be accommodated on a single node should be significantly increased compared to OS-level virtualization techniques that implement tenant isolation at the process level.
- Q2** It is required to keep the performance overhead imposed by usage control and SLA management within acceptable boundaries.

Section 3 explains how these requirements are fulfilled.

¹ Storage and network resources are not dealt with in this paper because the Java RCM API already provides quite straightforward ways to control IO usage.

² This is specifically required for CPU because fulfilling **F1** for the latter is feasible by suppressing the usage too much.

3 Thread-level Resource Usage Control

In order to address **Q1**, a single JRE instance is used for running code belonging to multiple tenants. This entails executing the code of each tenant in separate threads from each other and the main SaaS service. To enact fine-grained control over these threads, the platform support is introduced for preventing excessive memory usage as well as pause, resume and stop operations to throttle CPU usage (cf. **F1**, **F2**). The proposed solution consists of two main parts: (i) principal components in charge of usage monitoring and control presented in Section 3.1, and (ii) bytecode instrumentation discussed in Section 3.2.

3.1 Principal Components

The Java RCM API [18] provides meters for measuring the usage level of all types of resources, such as network bandwidth and CPU/memory, on a per-thread basis (cf. **F1**). Since the only implementation of this API in Oracle HotSpot JVM is not capable of imposing limits on CPU and memory usage, we have supplemented it to make it suitable for CPU and memory usage control.

The four principal components in charge of resource usage control are shown in Figure 2. The **Monitoring** component, which uses the Java RCM API, provides two meters, namely **CPUmeter** and **MemoryMeter**, counting CPU cycles per second and memory space in terms of bytes respectively. Both of these meters are responsible for notifying a central **UsageStatusRegistry** when a tenant exceeds its SLA limit (cf. **F1**) and also when the tenant usage level goes back to normal (cf. **F2**).

One instance of each meter is initialized for each tenant, holding the SLA *limits*, representing the permitted CPU cycles per second and the permitted number of memory bytes, provided by the **SLA Management** component which reads them from the **SLA Store**. Tenant meters are grouped in a **ResourceContext** and the latter is associated to tenant threads. The **Thread Management** component, which is an extension of the Java **Concurrency API**, is responsible for associating meters to **ControlledThread** instances.

For correct and efficient association of the meters to threads, the **Thread Management** component employs a multi-tenant thread pool. The multi-tenant thread pool instantiates and reserves a separate set of (dormant) threads for each tenant and forbids swapping threads across pools. This ensures that each thread is only used for a single tenant. Consequently, associating meters to threads,

Fig. 2: Usage Control Components

which incurs a considerable performance overhead, is restricted to the creation time of threads (cf. **Q2**).

The **Execution Control** component is responsible for forcing threads to sleep or stop (cf. **F1**) and, in this regard, implements a **control** method which is summarized in Listing 1.1. As a tenant can be recovered from excessive CPU usage by rescheduling threads, threads are forced to sleep in such cases, for a duration determined in the tenant SLA (lines 5-9). In contrast, excessive memory usage cannot be healed by passing of time. Hence, in that case, the thread will be removed from the **UsageStatusRegistry** and will kill itself by throwing an **OutOfMemoryError** (lines 1-4).

```

1  if (memoryExcess) {
2      UsageStatusRegistry.removeThread(this);
3      throw new OutOfMemoryError(); // kills the thread
4  }
5  while (cpuExcess) {
6      awake = false;
7      sleep(duration); // throttles the usage
8      awake = true;
9  }

```

Listing 1.1: Thread Control Behavior

It is possible that all threads of a tenant are eventually put into the sleep mode. In that case, the **CPU Meter** of that specific tenant is not updated anymore and updating the corresponding **cpuExcess** flag cannot be triggered by the meter. Therefore, the **Execution Control** component, specifically

its **IdleTenantRevivingService** which is scheduled to run repeatedly at fixed intervals, scans for tenants with all threads in sleep mode and resets their **cpuExcess** flag. Determining whether all threads of a tenant are in sleep mode requires (re)setting a specific flag before sleeping and after wakeup (lines 6 and 8 in Listing 1.1). Figure 3 summarizes the mechanism described above.

We call every invocation of the **control** method a checkpoint. To ensure that tenants cannot evade the control mechanism (cf. **F1**), at least one checkpoint is enforced in each of the following places: (i) every set of methods which can possibly make a recursion and (ii) every iterative structure such as a for-loop. Despite its exhaustive nature, this approach does not degrade performance considerably (cf. **Q2**) because, rather than measuring usage in every checkpoint, the

Fig. 3: Object Interactions for Usage Control

latter basically involves verifying boolean variables which are *asynchronously* set every time each individual meter recognizes a change of status.

Inserting a checkpoint in the aforementioned places is performed using Java bytecode instrumentation which is discussed in the following.

3.2 Bytecode Instrumentation

Checkpoints are not only required in the tenant custom code but also in the JRE and all cloud service classes accessible for tenants. Inserting usage checkpoints, hence, occurs at two stages by two distinct components shown in Figure 4 namely the **Offline Instrumentor** and the **Tenant Code Instrumentor**. These components are responsible for instrumenting the JRE bytecode alongside the code of the cloud service and tenant custom code respectively.

Since Java does not allow replacing the bytecode of certain JRE classes once they are loaded, instrumenting the JRE bytecode and that of the cloud service takes place *offline*, i.e. even before starting up the shared JRE. The **Tenant Code Instrumentor** kicks in at class loading time because tenant custom code may be uploaded at any moment in time.

Fig. 4: Bytecode Instrumentation Components

The bytecode modifications made by these components are identical. They first perform static code analysis for detecting all recursions. Afterwards, they insert a checkpoint in each detected recursion. In addition, the instrumentors insert a checkpoint in every place that can be used for dynamically building recursions at runtime. Finally, they scan the body of each method for inserting a checkpoint in every iteration.

As shown in Figure 4, the instrumentation components rely on the **Recursion Checkpoints** datastore. This datastore contains all the methods where a checkpoint is required for ensuring that tenants cannot exploit recursions to circumvent the usage control mechanism. To build this datastore, the **Call Graph Analysis** component analyzes a large directed graph, $CG = (M, I)$ where M is the set of all methods and I is the set of all possible invocations as defined in the following:

$$I = \{ \langle m_1, m_2 \rangle \mid m_1, m_2 \in M \ \& \ \text{invoke}(m_1, m_2) \} \quad (1)$$

where $\text{invoke}(m_1, m_2)$ is true *iff* m_1 may invoke m_2 at runtime. The possibility of polymorphic method invocations are taken into account as well. All simple cycles in CG are detected and a minimal subset of M is saved into the **Recursion Checkpoints** datastore such that each detected cycle has at least one member in the datastore.

The instrumentation components insert a checkpoint at the beginning of each method stored in **Recursion Checkpoints**. However, that is not sufficient to cover all possible recursions. In addition to the statically-detectable recursions, it is also possible to build recursions *dynamically* using the Reflection API or lambdas. To protect the system against exploitation of the Reflection API, a checkpoint is inserted in the `invoke` method of the `java.lang.reflect.Method`. A lambda expression, when compiled, yields one or more `invokedynamic` bytecode instruction(s). The target method of an `invokedynamic` instruction is determined at runtime. Therefore, a checkpoint is inserted before every instance of this instruction.

For inserting checkpoints in every iterative programming structure (e.g. a for-loop), the instrumentation components scan the body of each method and look for instructions jumping backward as indicators of iterations. Before every backward jump instruction, one checkpoint is inserted.

Circumventing the usage control mechanism requires exploiting repetitive programming instructions, i.e. either recursions or iterations. Our bytecode instrumentation covers all possible recursions (statically-detectable recursions, recursions made by the Reflection API, and those made by lambdas) and all iterations. Therefore, it is ensured that no tenant can evade the usage control mechanism (cf. **F1**).

4 Validation & Evaluation

This section presents the results of validating the proposed mechanism in terms of **F1** and **F2** and the evaluation results for **Q1** and **Q2**. Section 4.1 details the technical setup of the experiment environment and the subsequent sections present the experiments and analyzes the observations and the obtained results for each of the aforementioned requirements.

4.1 Implementation & Technical Setup

We have implemented the above solution on top of the JRE 1.8 shipped with Oracle HotSpot JVM.³ Our current implementation is limited for use in a single-node setup and evaluated on a Windows 10 (64-bit) machine with Intel Core i7 (3.6 GHz) and 16 GB of memory.

For the experiments, a multi-tenant batch-processing SaaS application is developed that runs batch jobs involving tenant custom code. Software artifacts of each tenant are wrapped in an OSGi bundle [21]. These tenant-provided custom code extensions are instrumented using the mechanism described in Section 3.2.

Before each set of experiments reported in the following section, a warmup task is performed by the multi-tenant application to make sure that the JVM bytecode optimizations are performed before main experiments start. For experiments validating/evaluating CPU usage control, the JVM OS process running

³ The source code can be downloaded via <http://people.cs.kuleuven.be/~majid.makki/icsoc-2019/main.html>.

the prototype SaaS application is given *realtime* priority on *exactly* 4 CPU cores out of 8 to ensure that the impact of other OS processes on the obtained results are kept minimal.

4.2 Validation of F1 for Memory Usage

In this experiment, the code of a tenant, whose SLA specifies a 300 *kB* memory limit, aims at instantiating the `java.math.BigDecimal` class one million times and keeping the instances in a `LinkedList`. The tenant custom code is allowed to instantiate objects and keep them in memory until it exceeds the 300 *kB* limit. This functional test shows that the thread running tenant custom code kills itself at that moment by throwing an `OutOfMemoryError` and the multi-tenant application hosting tenant custom code keeps working and serving requests.

4.3 Validation of F1/F2 for CPU Usage

Two specific tenants, whose CPU-related parameters of their SLA are presented in Table 1, are involved in 9 independent experiments each consisting of 10 concurrent jobs per tenant. Each job performs millions of `ADD` operations in a loop and is engineered to have a fixed number of usage checkpoints executed starting at 10 million checkpoints in the first experiment, increased by 10 million each time up to 90 million checkpoints in the last experiment. The length of jobs increase linearly with the number of executed checkpoints in these specific experiments. Therefore, these experiments verify the validity of **F1** and **F2** independent of the job length.

Four parameters are constantly recorded during all experiments: the start time of each job, its end time, its total

Table 1: CPU Parameters in SLA.

Tenant	Limit	Sleep Duration	Max Threads
Premium	100 MHz	25 ms	4
Standard	50 MHz	50 ms	4

sleep time, and finally the number of CPU cycles recorded by the `CPUMeter`.⁴ The average CPU usage of each tenant in every experiment is calculated using the following equation:

$$usage_{avg} = (1 - \frac{sleep_{avg}}{execution_{avg}}) \times request_{avg} \quad (2)$$

where $sleep_{avg}$, $execution_{avg}$ and $request_{avg}$ are the average sleep time, the average execution time and the average number of CPU cycles requested from the `CPUMeter` for each tenant in a specific experiment.

Figure 5 shows the average CPU usage of each tenant in each of the 9 experiments separately. It clearly shows that the **Standard** tenant is kept below

⁴ The act of constantly recording these parameters has no impact on the obtained results.

Fig. 5: Average CPU Usage

Fig. 6: Sleep Ratio

Fig. 7: Load-independent memory usage compared with containers.

Fig. 8: Tenant accommodation capacity compared with containers.

its 50 MHz limit (depicted as a dotted line in Figure 5) while the **Premium** tenant is allowed to go up to 80 MHz on average, which is below their SLA limit but sufficient for the submitted tasks. Non-zero sleep ratios depicted in Figure 6 show in effect that both tenants had tried to exceed their limits at certain time in some experiments. However, the control mechanism has forced them to sleep and kept the *average* CPU usage below the specified limits.

4.4 Evaluation of Q1

The goal is to compare process-level tenant isolation, representative of containers, with the proposed solution in terms of tenant accommodation capacity. A single-tenant variant of the experimental application is created on top of the standard JRE (i.e. no bytecode instrumentation) and executed separately for each tenant. The multi-tenant application is executed once on top of the modified JRE but without any tenant. The memory usage of each OS process is recorded 5 minutes after startup in absence of any load. For the multi-tenant variant, the OS process memory usage is also recorded 5 minutes after each tenant taken on-board. The above is performed for 5 distinct but identical tenants.

Figure 7 shows the recorded values for both solutions. Not surprisingly, when tenant isolation is enforced at level of OS processes, the load-independent memory overhead increases linearly with the number of tenants whereas the same factor reaches a nearly constant ceiling quite soon in the thread-level solution. Evidently, the memory usage will increase when load is imposed on the system but that would even more constrain the tenant accommodation capacity of the process-level solution.

More importantly, the increase in memory usage incurred by the imposed load does not affect the number of tenants that a single node can accommodate. Figure 8 presents the ratio of tenant accommodation capacity of both

solutions for different amount of memories available on a single node. The ratios are calculated by projecting the recorded memory usage of the first 5 tenants into higher orders (i.e. more number of tenants and more available memory). It can be seen that the ratio approaches 60 when sufficient amount of memory is available (cf. **Q1**).

4.5 Evaluation of Q2

The goal of this evaluation is to measure the runtime performance overhead imposed by the metering and the control mechanisms. It does not reflect the performance overhead of instrumentation which takes place only once for the JRE and once for tenant custom code. The normal circumstances, i.e. absence of any excessive usage, is taken as the baseline for comparisons. This is because, otherwise, the CPU sleep time will be involved in one case but not the other hence yielding incomparable results. To ensure that no usage excess will occur, the CPU and memory usage limits are set extremely high in the experiment.

The experiment involves two groups of 10 independent jobs for a single tenant. The jobs, identical across groups, consist of tenant custom code engineered to have exactly 100 million up to 1 billion checkpoints, increased by the factor of 100 million each time. In the first group of executions, our bytecode instrumentation is applied whereas the second group is performed on the standard JRE without instrumenting neither the cloud service code nor tenant custom code, and without any meters involved.

The differences between the execution time of each pair of identical jobs are presented in Figure 9. Based on the obtained results, the performance overhead of the control mechanism is on average 0.31 *ns* per checkpoint with standard deviation of 0.03 *ns*.

Fig. 9: Performance Overhead.

5 Discussion

This section elaborates on a number of issues concerning further improvement of the proposed solution as well as its real-world application.

Distribution. Currently, the focus has been on increasing the tenant accommodation capacity on a single node. However, coping with load imposed on the system and realistic cloud deployment will require distributed deployment. Dedicated types of orchestration facilities (e.g. SLA-aware load-balancers or thread schedulers) will be required to ensure that the SLA limits are respected across all nodes (cf. [22, 23]).

Supplementary Building Blocks. Beside resource usage control, a thread-level tenant isolation solution requires mechanisms for access control and code

restriction facilities (cf. [24]) to avoid problems with **static** members (cf. [17]) and exploitation of the usage control mechanism by tenants.

Flexible and Adaptive Control. The implemented solution imposes a *fixed* and *strict* limit on tenant usage. In practice, it is possible to allow tenants to slightly surpass their SLA limits if sufficient resources are available. Hence, system health should be monitored and more flexible limit enforcement policies control the tenant usage levels. For instance, a *leniency factor* may be used to determine the extent to which each tenant may go beyond their SLA limits (cf. [24]).

In addition, thread sleeping can be adaptive. Instead of a fixed sleep duration, specified in the tenant SLA, the latter can be adjusted at runtime based on previous records of CPU usage. Furthermore, the number of threads forced to sleep when usage excess is observed can be in function of the transgression rate while in the current implementation, all threads belonging to the aggressive tenant are forced to sleep.

Performance Optimization. Our observations shows that a great majority of checkpoints are inserted in *iterations* rather than recursions. Therefore, the overall performance overhead of usage control mechanism can be further reduced by using a local counter in the iterations for intelligently determining how frequently the checkpoint should actually be involved at runtime instead of the current brute-force machinery.

Hybrid Approaches. The proposed solution is not incompatible to other tenant isolation approaches, and in some cases, it could be beneficial to combine different solutions (cf. [25]). For example, a potential tenant accommodation model could involve clustering different groups of tenants (e.g. premium vs. standard tenants) in separate containers and our thread-level solution could be employed to protect tenants within each container. As shown in the case-driven study of Ochei et al. [26], decisions related to tenant isolation are impacted by many complex trade-offs, and depend on many factors such as the application nature, the trust bestowed upon tenants, the nature of the customizations, etc.

Applicability beyond JRE. Our approach can be implemented in runtime platforms of other languages given that they meet the following conditions: (i) threads or similar structures can be safely paused/resumed, (ii) CPU and memory can be measured on a per-thread basis, (iii) the code of the programming language is compiled into an intermediate language, such as Java bytecode, which can be instrumented before ultimately compiled into machine code.

6 Related Work

This section compares and contrasts our solution with related work.

Performance Isolation. Related work on performance isolation of multi-tenant SaaS applications [7–10, 27–29] focuses on aspects of performance isolation between tenants sharing a single run-time environment. The main distinction however is that these approaches involve *known* and *trusted* service logic,

whereas the specific problem addressed in this paper starts when tenants implement part of the system functionality by providing their own code which is by nature *untrusted*.

The use of tenant custom code that runs in the SaaS application makes the external metrics commonly used in this context (such as job execution time, throughput or response time) unsuited for isolating tenants performance-wise within SaaS applications (cf. [29]). Other elements of related work also rely on internal metrics such as memory and CPU usage [7, 10, 27, 28]. PerfCloud [30] utilizes system-level metrics to proactively detect performance interference between tenant workloads and shows that such approaches succeed in avoiding costly workload profiling and prediction mechanisms without having any interference on application code.

Java Studies. This work is in continuation of existing work that deals with the limitations of running user code in Java threads. Herzog et al. [16] and Rodero-Merino et al. [17] highlight safe termination of threads and resource control as one of the main obstacles. Work aimed at resource consumption control in the JRE [19, 31, 32] has led to the JSR-284 specification [18] also known as the Java RCM API. However, neither the specification nor its implementations (e.g. Oracle HotSpot JVM) currently provide any way for imposing limits on CPU and memory usage. The solution presented here involves complex, holistic and cohesive instrumentation of (i) the JRE, (ii) cloud service provider code, and (iii) tenant-provided code to overcome the above shortcomings.

Multi-tenant JVM. Specialized Java Virtual Machines (JVM) have been built for executing the code provided by multiple tenants in a single JVM instance [11–15]. In contrast, instead of developing specialized JVMs, our approach provides a customized yet backwards-compatible JRE, which can as such be used in industrialized JVMs. In MVM [12, 14], I-JVM [13] and IBM MT JVM [15], tenant custom code is executed as a separate application. This is not desirable in the context of multi-tenant SaaS applications in which tenant custom code runs as an integrated part of the SaaS application, rather than an independent application. In addition, our approach allows sharing JRE and other libraries and consequently increases scalability.

Despite requiring a separate OS process for each tenant, KaffeOS [11] can still be used for SaaS applications that execute tenant custom code, mainly owing to its hierarchical memory management which supports objects being shared between Java processes. This sharing mechanism however sacrifices application portability: every time an object is to be shared or a shared object is required, a specific `Heap` class is *explicitly* used, and this makes it impossible to port the code to another JVM. This becomes even more problematic when an existing application or platform (e.g. a web-server) is required as a multi-tenant cloud service. Everything should be redesigned according to the new programming model which is fundamentally different from standard Java practices whose memory management is entirely hidden from application developers.

Interpretive approach. The Activiti workflow engine allows *Secure Scripting* [33] by means of the instruction count callback mechanism provided by the

Rhino [34] JavaScript engine. Even though resource usage control is limited to memory,⁵ the Java RCM API can be used to support other types of resources. However, this approach is only applicable to interpreted languages because it relies on counting the number of intermediate language instructions before performing a usage control. Our approach, instead, deals with languages whose bytecode are ultimately compiled into native machine code. In that case, there is no interpreter that can count the executed instructions and perform usage control via a callback function. Due to independence of our approach from interpreters, the performance degradation is considerably lower.

7 Conclusion

When it comes to processing service requests, for example in the context of web content, thread-level processing allows for high scalability and resource efficiency. In the context of multi-tenancy however, the limited monitorability and controllability of server threads in terms of their CPU and memory usage impede attaining the desired property of tenant isolation. This is especially problematic in service offerings that can be customized by tenants through custom code. For these reasons, tenant isolation is in practice accomplished by means of containers or virtual machines (VM) in multi-tenant architectures (cf. [35]).

In this paper, which fits into our ongoing research track on application-level tenant isolation in SaaS offerings, we take advantage of Java bytecode instrumentation and the Java RCM API to accomplish CPU and memory usage control at thread-level. By validating our solution in a prototype application, it is shown that the usage level of these two critical resources can be controlled without using containers or VMs. Evaluation results show that the proposed solution increases tenant accommodation capacity with a factor of 59 compared to process-level tenant isolation at the expense of 0.31 *ns* average performance overhead per usage control checkpoint.

Acknowledgment. This research is partially funded by the Research Fund KU Leuven (project GOA/14/003 - ADDIS) and the strategic basic research (SBO) project DeCoMAdS.

References

1. D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat, “Enforcing performance isolation across virtual machines in xen,” in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pp. 342–362, Springer, 2006.
2. G. Somani and S. Chaudhary, “Application performance isolation in virtualization,” in *Cloud Computing, 2009. CLOUD’09. IEEE International Conference on*, pp. 41–48, IEEE, 2009.

⁵ The claimed CPU usage control in fact restricts the response time of the untrusted script rather than its CPU usage.

3. Y. Li, W. Li, and C. Jiang, "A survey of virtual machine system: Current technology and future trends," in *Electronic Commerce and Security (ISECS), 2010 Third International Symposium on*, pp. 332–336, IEEE, 2010.
4. L. M. Vaquero, L. Roderio-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 1, pp. 45–52, 2011.
5. C. D. Weissman and S. Bobrowski, "The design of the force. com multitenant internet application development platform," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pp. 889–896, ACM, 2009.
6. H. Song, F. Chauvel, and A. Solberg, "Deep customization of multi-tenant saas using intrusive microservices," in *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*, pp. 97–100, IEEE, 2018.
7. X. H. Li, T. C. Liu, Y. Li, and Y. Chen, "Spin: Service performance isolation infrastructure in multi-tenancy environment," in *International Conference on Service-Oriented Computing*, pp. 649–663, Springer, 2008.
8. H. Lin, K. Sun, S. Zhao, and Y. Han, "Feedback-control-based performance regulation for multi-tenant applications," in *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pp. 134–141, IEEE, 2009.
9. P. Leitner, B. Wetzstein, F. Rosenberg, A. Michlmayr, S. Dustdar, and F. Leymann, "Runtime prediction of service level agreement violations for composite services," in *Service-oriented computing. ICSOC/ServiceWave 2009 workshops*, pp. 176–186, Springer, 2010.
10. W. Wang, X. Huang, X. Qin, W. Zhang, J. Wei, and H. Zhong, "Application-level cpu consumption estimation: Towards performance isolation of multi-tenancy web applications," in *Cloud computing (cloud), 2012 IEEE 5th international conference on*, pp. 439–446, IEEE, 2012.
11. G. Back, W. C. Hsieh, and J. Lepreau, "Processes in KaffeOS: Isolation, resource management, and sharing in java," in *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, p. 23, USENIX Association, 2000.
12. G. Czajkowski, L. Daynès, and B. L. Titzer, "A multi-user virtual machine.," in *USENIX Annual Technical Conference, General Track*, pp. 85–98, 2003.
13. N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frénot, and B. Folliot, "I-jvm: a java virtual machine for component isolation in osgi," in *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pp. 544–553, IEEE, 2009.
14. G. Czajkowski and L. Daynès, "Multitasking without compromise: a virtual machine evolution," *ACM SIGPLAN Notices*, vol. 47, no. 4a, pp. 60–73, 2012.
15. G. Johnson and M. Dawson, "Introduction to java multitenancy," tech. rep., 2015.
16. A. Herzog and N. Shahmehri, "Problems running untrusted services as java threads," in *Certification and Security in Inter-Organizational E-Service*, pp. 19–32, Springer, 2005.
17. L. Roderio-Merino, L. M. Vaquero, E. Caron, A. Muresan, and F. Desprez, "Building safe paas clouds: A survey on security in multitenant software platforms," *computers & security*, vol. 31, no. 1, pp. 96–108, 2012.
18. JCP, "JSR 284: Resource Consumption Management API." <https://jcp.org/en/jsr/detail?id=284>. Accessed: 2018-12-04.
19. G. Czajkowski, S. Hahn, G. Skinner, P. Soper, and C. Bryce, "A resource management interface for the java platform," *Software: Practice and Experience*, vol. 35, no. 2, pp. 123–157, 2005.

20. D. Grove, G. DeFouw, J. Dean, and C. Chambers, “Call graph construction in object-oriented languages,” *ACM SIGPLAN Notices*, vol. 32, no. 10, pp. 108–124, 1997.
21. O. Alliance, “OSGi specification.” <https://osgi.org/download/r4v43/osgi.core-4.3.0.pdf>, 2012. Accessed: 2017-04-19.
22. 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 *OTM Confederated International Conferences* “On the Move to Meaningful Internet Systems”, pp. 302–320, Springer, 2011.
23. Y. J. Kim, Y. C. Lee, H. Han, and S. Kang, “Hierarchical recursive resource sharing for containerized applications,” in *International Conference on Service-Oriented Computing*, pp. 781–796, Springer, 2018.
24. M. Makki, D. Van Landuyt, and W. Joosen, “Towards paas offering of bpmn 2.0 engines: A proposal for service-level tenant isolation,” in *European Conference on Service-Oriented and Cloud Computing*, pp. 5–19, Springer, 2017.
25. E. Truyen, D. Van Landuyt, V. Reniers, A. Rafique, B. Lagaisse, and W. Joosen, “Towards a container-based architecture for multi-tenant saas applications,” in *Proceedings of the 15th International Workshop on Adaptive and Reflective Middleware*, p. 6, ACM, 2016.
26. L. C. Ochei, J. M. Bass, and A. Petrovski, “Degrees of tenant isolation for cloud-hosted software services: a cross-case analysis,” *Journal of Cloud Computing*, vol. 7, p. 22, Dec 2018.
27. X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, “Cpi 2: Cpu performance isolation for shared compute clusters,” in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 379–391, ACM, 2013.
28. R. Krebs, S. Spinner, N. Ahmed, and S. Kounev, “Resource usage control in multi-tenant applications,” in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pp. 122–131, IEEE, 2014.
29. S. Walraven, W. De Borger, B. Vanbrabant, B. Lagaisse, D. Van Landuyt, and W. Joosen, “Adaptive performance isolation middleware for multi-tenant saas,” in *Utility and Cloud Computing (UCC), 2015 IEEE/ACM 8th International Conference on*, pp. 112–121, IEEE, 2015.
30. P. Lama, S. Wang, X. Zhou, and D. Cheng, “Performance isolation of data-intensive scale-out applications in a multi-tenant cloud,” in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 85–94, IEEE, 2018.
31. W. Binder, J. G. Hulaas, and A. Villazón, “Portable resource control in java,” in *ACM SIGPLAN Notices*, vol. 36, pp. 139–155, ACM, 2001.
32. A. Janik and K. Zieliński, “Transparent resource management with java rm api,” in *International Conference on Computational Science*, pp. 1023–1030, Springer, 2006.
33. “Activiti.” <https://www.activiti.org/>. Accessed: 2018-12-04.
34. “Rhino.” <https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino/>. Accessed: 2018-12-04.
35. M. HoseinyFarahabady, Y. C. Lee, A. Y. Zomaya, and Z. Tari, “A qos-aware resource allocation controller for function as a service (faas) platform,” in *International Conference on Service-Oriented Computing*, pp. 241–255, Springer, 2017.