Cloud platforms and embedded computing - The operating systems of the future

Rellermeyer, Jan S.; Lee, Seong Won; Kistler, Michael

DOI
10.1145/2463209.2488826

Publication date
2013

Document Version
Accepted author manuscript

Published in
Proceedings of the 50th Annual Design Automation Conference, DAC 2013

Cloud Platforms and Embedded Computing – The Operating Systems of the Future

Jan S. Rellermeyer[1] Seong-Won Lee[1,2] Michael Kistler[1]

[1]IBM Austin Research Lab
Future Systems Group
{rellermeyer, mkistler}@us.ibm.com

[2]Seoul National University
School of Electrical Engineering and Computer Science,
swlee@altair.snu.ac.kr

ABSTRACT

The discussion on how to effectively program embedded systems has often in the past revolved around issues like the ideal instruction set architecture (ISA) or the best operating system. Much of this has been motivated by the inherently resource-constrained nature of embedded devices that mandates efficiency as the primary design principle.

In this paper, we advocate a change in the way we see and treat embedded systems. Not only have embedded systems become much more powerful and resources more affordable, we also see a trend towards making embedded devices more consumable, programmable, and customizable by end users. In fact, we see a strong similarity with recent developments in cloud computing.

We outline several challenges and opportunities in turning a language runtime system like the Java Virtual Machine into a cloud platform. We focus in particular on support for running multiple tenants concurrently within the platform. Multi-tenant support is essential for efficient resource utilization in cloud environments but can also improve application performance and overall user experience in embedded environments. We believe that today's modern language runtimes, with extensions to support multi-tenancy, can form the basis for a single continuous platform for emerging embedded applications backed by cloud-based service infrastructures.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Run-time environments*

General Terms

Design, Performance

DAC '13, May 29 - June 07 2013, Austin, TX, USA.

Keywords

Platform as a Service, Embedded Systems, Cloud Computing

1. INTRODUCTION

Traditional embedded systems are fixed-function devices that combine specialized hardware with special-purpose, low-level software and provide very limited or even no user interface elements. The software components of traditional embedded systems are often closely tied to the hardware implementation, in some cases written directly in the assembly language for the processor. Examples of traditional embedded systems are digital watches, MP3 players, and computer modems.

While many of today's embedded systems fit this traditional model, a new class of embedded systems is emerging that offers a much larger set of capabilities, increased flexibility, and a greatly enhanced user interface. This new class of embedded systems is made possible by the continued advance of hardware technologies, particularly in the areas of microprocessor performance, memory density, and reduced power consumption. Touch-screen and voice-recognition technologies have also helped drive many of the new user interface features in these new embedded systems. Common examples of this new type of embedded system are smart phones and tablets, in-car navigation and entertainment systems, and programmable wireless access points (e.g. running OpenWRT [29]).

Traditional cloud computing environments, on the other hand, offer a highly flexible and programmable environment. A typical cloud environment provides clients with *virtual machines* which can be programmed with a choice of operating system, middleware, and applications, all configured according to the client's specifications. The client typically also has a choice of virtual machines with different amounts of processing power, memory size, and storage configuration. This style of cloud environment has come to be known as Infrastructure-as-a-Service, or *IaaS*. Common examples of IaaS cloud environments are Amazon EC2 [12] and IBM's SmartCloud Entry [19].

Cloud environments are also evolving. While the traditional IaaS cloud offers very high flexibility and configurability, this comes with a large system administration burden.

Virtual machines need similar care in terms of maintenance and security patching as physical machines do. Furthermore, many new applications for the cloud are being developed in highly portable languages such as Java and Javascript. As a result, the flexibility and configurability of the IaaS virtual machine offers little benefit to these modern applications but still entails a high cost. These trends have driven a new paradigm in cloud computing referred to as Platform-as-a-Service, or *PaaS*. A PaaS cloud environment provides the client with one or more high-level application runtimes, such as the Java Runtime Environment (JRE), along with other services such as a database. a key-value store, or an authentication service. The PaaS cloud provider manages all of the underlying physical and virtual hardware, operating system images, file systems, and network configuration.

The trends in embedded systems and the trends in cloud computing platforms have striking similarities. In both cases, the platform is evolving towards higher functionality with a focus on application-level capabilities. In fact, many of the technologies between these two spaces are starting to converge. For example, Javascript was once almost exclusively a client-side language used primarily within web browsers, but through packages such as Node.js [27] it is now also being employed in web applications deployed in PaaS cloud environments (e.g., Microsoft's Windows Azure [25]). Similarly, Java, Ruby, Scala, and Python, are now quite commonly used in both embedded and cloud-based applications. Besides languages, other common infrastructures are also emerging, e.g., asynchronous and multi-channel network communication such as WebSockets [13]. This enables new opportunities for shifting parts of the application between the client and the cloud. In short, the embedded and PaaS cloud application runtime environments now have many significant areas of similarity.

What gives this trend special significance is that many emerging embedded applications utilize one or more web applications to provide their functionality, and these web applications are often deployed into PaaS cloud environments. The traditional view of this application design may be *client-server*, but the similarity of the application runtime environments between client and server allows us to view it as a more general *distributed* application. The difference is subtle but powerful, in that it allows a more flexible and dynamic distribution of functionality between embedded application and cloud-based service. Application data can be stored on the device or in the cloud, and the code to manage this can be nearly identical between the two platforms. Data analysis and transformation functions can be performed either on the device or in the cloud, with the decision made dynamically and possibly even involving transfer of the functions' implementation between the device and the cloud.

Differences certainly still remain between the embedded environment and PaaS cloud environments, e.g., the typical total amount of resources, the richness of user input and interaction, or the degree of multi-tenancy. While differences such as these are certainly significant, we believe there is often value in focusing on the similarities of the environments and exploiting these where possible. Having a common platform to program both the embedded device and the cloud helps to create a symbiotic relationship and a continuous user experience.

In the following section, we discuss mobile phones as an example of embedded systems which have already evolved into a more open and consumable platform that includes devices and the cloud. Mobile phones also provide a prime example of language runtime systems serving as a platform for both embedded devices and the cloud. In Section 3, we shed light on the systems issues involved in turning a language runtime system like the Java Virtual Machine [23] into a platform. While some of the architectural changes are motivated by the multi-tenant setup in the cloud, they can also help to reduce the footprint of the platform and make it better suited for embedded devices. Finally, in Section 4 we discuss the implications of using the principles of Platform-as-a-Service on the embedded system to create a continuous user experience between embedded device and the cloud.

2. LESSONS LEARNED FROM MOBILE PHONES

Mobile phones are a prominent example of a class of embedded systems that is already and aggressively moving towards a symbiotic relationship with the cloud. For instance, isolated on-device storage is becoming less important and many platforms already use it more as a cache for cloud-based storage. Furthermore, mobile phones have made the transition from closed devices programmed by experts and in low-level languages to vibrant platforms where a large community of developers creates an ecosystem of apps around the devices. The most popular platforms such as Android are based on high-level, interpreted or just-in-time compiled languages such as Java, C#, or JavaScript to bridge hardware heterogeneity and lower the burden of writing applications. The notable exception is Apple's iOS which uses Objective C, arguably still a low-level language. It has to be taken into account, though, that Apple is also one of the few examples of a mobile platform that is entirely controlled by a single company in that Apple develops both the software and the hardware. Since they do not make their software available to any other hardware platform heterogeneity is severely limited. However, even for the iOS platform there is a trend towards developing content in a platform-independent manner and using native capabilities only through plugins, e.g., as in PhoneGap [32].

Another notable aspect of smart phones is the interaction between the devices and the backend server infrastructure. Smart phone applications tend to be limited in the amount of processing performed on the device. Yet, they have emerged as a more seamless experience for accessing external content that is better integrated into the client platform as opposed to web pages running in a browser. Since mobile apps typically have strong dependencies to their backend infrastructure, developers are increasingly exploring ways of developing both client app and backend logic in a single process and using the same tools. For instance, it has become popular to use languages like JavaScript originally designed for client-side processing on the server side, e.g., with Node.js [27] or Vert.X [35]. This has lead to a new type of Platform as a Service (PaaS) specifically for mobile devices, often referred to as Mobile Backened as a Service (BaaS or MBaaS) which specifically targets the common services that mobile applications require.

While there are many potential advantages to a common platform that could support both embedded and PaaS cloud applications, there are also some significant challenges. One of these is support for *multi-tenancy*, which is the ability

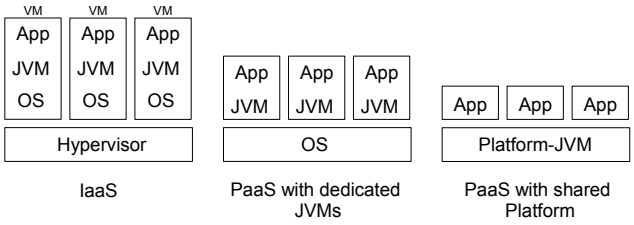


Figure 1: Options for Embedding Runtime Systems into Platforms

to support multiple independent active applications on the platform. Multi-tenancy is clearly a key feature for cloud environments, where the role of the cloud infrastructure is to achieve efficiency through high resource utilization by sharing physical and virtual resources to a potentially large set of active applications. Another dimension of multi-tenant support is the efficient handling of tenants and workloads that continuously come and go over time.

With embedded devices such as mobile phones increasing in processing capability and richness of user interface, support for multi-tenancy is poised to deliver significant value in this space as well, particularly if this support is provided in a consistent, seamless fashion across the embedded and cloud environments. Returning to our mobile phone example, many newer applications include support for push notifications, alerts, alarms, scheduled operations, and other features that generally require the app to remain running in the background. As a result, it is not uncommon today for a mobile phone to have tens of active applications sharing the resources of the device. All indications are that the number of active applications will grow rapidly in future years, so robust and efficient multi-tenancy is also a key requirement for future embedded platforms.

3. FROM RUNTIME SYSTEMS TO PLATFORMS

Language runtime systems such as the Java Virtual Machine [23] have reached a high level of optimization and by now run a significant share of enterprise workloads such as application servers (e.g., WebSphere [20]), analytics (e.g., Hadoop [1]), and search (e.g., Lucene [2]). As a result, many popular Platform as a Service offerings such as OpenShift [28], Google AppEngine [15], Heroku [17], and VMware CloudFoundry [10] include Java support. However, optimal resource utilization is important for cloud stacks in order to be cost effective, a similarity with embedded systems which are resource-constrained by design.

Figure 1 illustrates multiple ways to embed language runtime systems such as the JVM into platforms and make them available to multiple tenants at the same time. The first and most straightforward way is to build the platform atop IaaS and let every tenant get its own virtual machine image containing an operating system, the language runtime (in this case the JVM) and the application that the tenant wants to run (left part of Figure 1). This solution provides nearly perfect isolation between tenants and predictable performance governed by the hypervisor. However, it also has high resource overhead due to the dedicated OS instances and the footprint of the virtual machines. As a result, typical servers can run only a small number of tenants on the same hardware, which limits the density and thereby the effective utilization of hardware resources. Many platforms therefore do not rely on hardware virtualization but instead resort to some form of OS-provided process-based isolation such as chroot jails [8] or Linux Containers (LXC) [24]. In this solution (middle of Figure 1) tenants share the operating system but still run dedicated instances of the language runtime, which imposes overhead. By intuition, it should be most resource-efficient to run all tenants on a single, shared runtime (right part of Figure 1). This certainly raises security concerns due to the lack of proper isolation but it represents a baseline for the performance of multi-tenancy.

We devised a series of experiments to explore the benefits and limitations of a shared JVM platform. All of our experiments use multiple concurrent instances of the DaCapo [6] Batik benchmark as the primary workload, where each benchmark instance represents a tenant. All experiments were performed on a 12 core Intel Xeon E5645 system with HyperThreading running at 2.40 GHz, 12 GiB of main memory, and running Oracle's HotSpot JVM build 1.6.0_33-b03 in server mode.

Figure 2 presents the average per-instance execution time of the DaCapo Batik benchmark when running on either a *dedicated JVM* platform, where each instance of the benchmark runs on a separate JVM, or a *shared JVM* platform, with all benchmark instances running in the same JVM. The results indicate that the dedicated JVM platform can support up to about 85 instances while still providing predictable performance. The candlestick error bars depict both the standard deviation between iterations of the same experiment (sticks), of which we conducted 5 per data point, as well as the average spread between the instances executed in each run (candle). After 85 instances, the system starts to thrash due to resource exhaustion and exhibits highly volatile behavior. Surprisingly, the shared JVM platform does not scale as well as expected. After about 20 concurrent benchmark instances the system exhibits volatile and inferior per-instance performance, significantly worse than running each instance on a dedicated JVM.

This experiment clearly illustrates that the JVM lacks support for efficient execution of multi-tenant workloads. The results suggest that increasing sharing between instances to utilize resources more effectively has actually created a

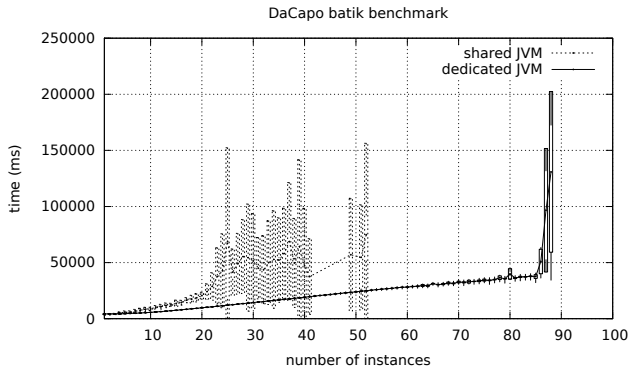
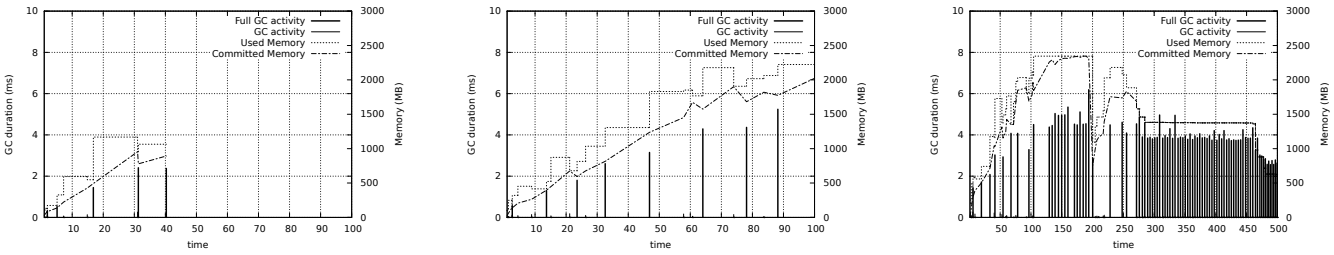


Figure 2: DaCapo Batik Benchmark



(a) 10 Instances (b) 20 Instances (c) 40 Instances

Figure 3: Garbage collection activity for the DaCapo Batik benchmark

high level of accidental sharing within the virtual machine that adversely affects performance. In the following sections we identify some of the key structural changes required for the JVM to support efficient multi-tenancy. By doing so, we can draw new lines in the design space for language runtime systems between tenant isolation where required and resource sharing where permitted. This will allow the JVM to use cloud resources more efficiently, and also to run more client applications within the scarce resources of an embedded system.

3.1 Tenant Isolation and Memory Management

Isolation in JVMs has been previously discussed, e.g., in the context of running multiple (desktop) applications on the same JVM. Sun's Multitasking Virtual Machine (MVM) [11] introduced the notion of *Isolates* which form containers of minimal isolation between tasks whereas the remaining JVM and its internal components are shared. The authors indicate that static fields, class initialization state, and instances of *java.lang.Class* are sufficient to replicate per task to achieve isolation whereas everything else (e.g., constant pool, interpreter, JIT compiler, etc.) can be shared. The isolates in the MVM, however, provide full isolation akin to OS processes which makes communication among isolates costly and sharing of common resources difficult. This is particularly an issue with fine-granular component models like OSGi [30] where inter-module communication is expected to happen frequently. Projects like i-JVM [14] have therefore looked into more lightweight forms of isolation by allowing threads to cross the boundaries of isolates and controlled sharing of objects for inter-isolate communication. The general challenge remains to share where desired, e.g., to avoid duplication of identical resources like the Java classpath, and to isolate where tenants show individual behavior. This is not only a security concern but has important implications on the resource consumption of applications and performance. One example where the performance aspect of isolation shows is communication, as the authors of the i-JVM have pointed out. Another one is memory management.

Garbage collection is a critical component in managed runtime systems and can have significant impact on the runtime performance of applications [5, 18]. In current JVMs, the garbage collector (GC) is unaware of independent tenants with no common object references, so it cannot operate on individual isolates but needs to traverse the entire heap each collection cycle. We performed an experiment to assess the impact of GC on a shared JVM platform. In this experiment, we run multiple instances of the benchmark concurrently in a single shared JVM. Due to lack of isolation in the JVM, all tenants share a single heap which is managed by one instance of the garbage collector. This resembles the situation in i-JVM and the *old generation* on the heap of the MVM. Figure 3 presents time-series graphs that show the behavior of the GC during the execution of 10, 20, and 40 instances of the benchmark. GC activity is plotted as bars whose height indicates the duration of the activity in ms as shown on the left y-axis. Used and committed memory in MB are shown as lines plotted against the right y-axis. All runs use the default JVM heap size of 3GB, and GC statistics are obtained from the JVM through command line flags. When running 10 or 20 concurrent instances as in Figures 3(a) and 3(b), full collection cycles occur infrequently and complete in a short time. The trace for 40 instances (Figures 3(c)) shows a system that is overloaded with garbage collection towards the second half of the run, even though the memory pressure does not appear to be higher than for 20 instances.

This illustrates a scalability problem with larger heaps and underlines the need for better tenant isolation when using the Java Virtual Machine as a platform for multiple tenants. It also points out that the assumption that the majority of JVM components can be shared is sufficient for isolation but likely exhibits scalability and performance issues when running a high number of tenants. Instead, we propose a different JVM architecture where components like garbage collection are offered as a service (but not necessarily as a single instance) to multiple tenants.

3.2 Just-in-time Compilation

Just-in-time compilation is a proven technology for acceleration of high-level interpreted languages. For a single application, the rule of virtue is to concentrate on hot spots of the application with the compilation capacity as large as possible. Various approaches have been developed in research as well as in products for improving the performance of JIT compilation. Among these efforts are a selective compilation with bytecode interpreter [31, 33] and adaptive compilation with multi-level optimizing compilers [3, 34, 9]. Because the decision of when and what to compile has to be determined during runtime, the hot spot detection technique is one of the most important concerns in these optimization frameworks [4, 16, 7, 26].

Figure 4 shows the total JIT time (in seconds) and JIT throughput (in bytecodes processed per second) of the just-in-time compiler during the execution of multiple instances of the DaCapo Batik benchmark on a shared JVM platform. From our previous experiments we know that performance

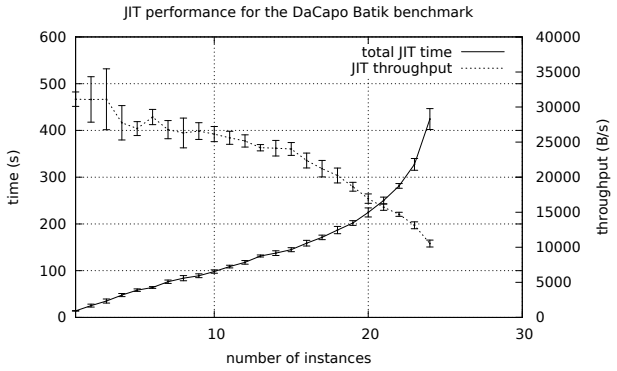


Figure 4: JIT Throughput for the Batik Benchmark

begins to degrade at around 20 instances, and in this graph we see that at this point the total JIT time is increasing exponentially while JIT throughput is significantly degraded. This is a clear indication that the shared JIT compiler becomes a bottleneck and contributes to the limited scalability of the JVM in a multi-tenant setup.

For our proposed cloud platform, we have to address a different strategy because each tenant has a resource and content sharing policy of its own and we are supposed to provide compilation resource in a more flexible manner. A promising design option is having isolated compiler instances, created on demand and assigned job allocation from task pools shared among tenants. Thus, as depicted in Figure 5, no duplicated compilation request for a shared item among tenants exists in the compilation task pool and tenants are served generated native code from the shared code cache or output of JIT compiler directly.

As our multi-tenancy model is structurally similar to a multi-threaded application running on a single JVM from a resource sharing point of view, we can anticipate our JIT compiler operation based on some previous work in such environments. Our cloud platform tries to achieve a goal to maximize resource sharing among tenants and it inevitably can occur situations such that multiple tenants together with a compiler instance are scheduled to be distributed CPU time. Kulkarni et. al. have demonstrated that application performance could be severely degraded as the number of threads per core is increased and a single compilation thread is dispossessed of its resource utilization by the JVM thread scheduler [21]. However, even under this circumstance, it makes the throughput stable sooner to guarantee some amount of CPU time for the compilation thread. Obviously, more than one JIT compiler instance can leverage the throughput of compilation requests especially when there exist multiple tenants starting up over abundant hardware resources. As a matter of fact, it is reported that multiple compilation threads can produce significant performance improvement on a many core environment [22].

3.3 Changes to the JVM

We have identified the need for structural changes in language runtime systems like the JVM in order to be used as multi-tenant platforms. Most importantly, proper support for tenant isolation needs to be added not only for security but also for performance reasons. At the same time, it is desirable to share common infrastructure such as the classpath, class linking and loading, GC, JIT compiler, etc. However, especially the performance-critical components of the JVM do not scale well enough to be used as system singletons. Instead, what we envision is a dynamic pool of these components, each serving a group of tenants. We have illustrated this design approach in more detail for just-in-time compilation but the same principle can also be applied, e.g., to memory management/garbage collection or the classloading subsystem.

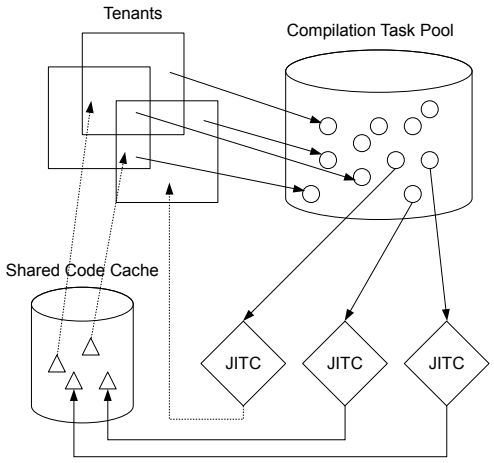


Figure 5: JIT Compiler Operation in Cloud Platform

4. CONCLUSION: PAAS ON EMBEDDED DEVICES

In the previous sections, we have outlined some of our experience in using a traditional language runtime system like the JVM in a multi-tenant setup for cloud computing. The goal was to provide a higher density of tenants on a single server machine and we were able to show that the current JVM is not able to achieve this goal due to adverse effects when running multiple tenants on the same VM. We have discussed architectural changes to the JVM that are likely to improve this situation. In our ongoing work, we envision that language runtime systems like the JVM are not only becoming more scalable and amenable for running multiple tenants in PaaS settings but ultimately also become more lightweight in general. This is an important step towards running the same stack on both the backend server and the embedded device and provide a continuous platform experience between device and cloud. Here it will have even greater benefits because embedded resources are typically much more static and constrained in comparison to the cloud. If platforms will eventually be able to cover both the client devices and the cloud backend, thereby raising the level of abstraction significantly, they will become what the next generation of users and programmers will see as the operating system of the future for both embedded systems and the cloud.

5. REFERENCES

[1] Apache Hadoop. `hadoop.apache.org`.

[2] Apache Lucene. `lucene.apache.org`.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeno JVM. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pages 47–65, New York, NY, USA, 2000. ACM.

[4] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 168–179, New York, NY, USA, 2001. ACM.

[5] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: the performance impact of garbage collection. *SIGMETRICS Perform. Eval. Rev.*, 32(1):25–36, June 2004.

[6] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[7] D. Buytaert, A. Georges, M. Hind, M. Arnold, L. Eeckhout, and K. De Bosschere. Using hpm-sampling to drive dynamic compilation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 553–568, New York, NY, USA, 2007. ACM.

[8] chroot. `http://www.freebsd.org/chroot/2`.

[9] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. The Open Runtime Platform: a flexible high-performance managed runtime environment: Research Articles. *Concurr. Comput. : Pract. Exper.*, 17(5-6):617–637, Apr. 2005.

[10] Cloud Foundry. `http://www.cloudfoundry.com`.

[11] G. Czajkowski and L. Daynés. Multitasking without comprimise: a virtual machine evolution. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 125–138, New York, NY, USA, 2001. ACM.

[12] Amazon Elastic Compute Cloud (EC2). `http://aws.amazon.com/ec2`.

[13] I. Fette and A. Melnikov. The WebSocket Protocol. *RFC 6455*, December 2011.

[14] 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 *International Conference on Dependable Systems and Networks (DSN 2009)*, Estoril, Portugal, June 2009. IEEE Computer Society.

[15] Google AppEngine. `https://developers.google.com/appengine`.

[16] D. Gu and C. Verbrugge. Phase-based adaptive recompilation in a JVM. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 24–34, New York, NY, USA, 2008. ACM.

[17] Heroku Cloud Application Platform. `http://www.heroku.com`.

[18] M. Hertz and E. D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 313–326, New York, NY, USA, 2005. ACM.

[19] IBM Smart Cloud Entry. `http://www.ibm.com/cloud`.

[20] IBM WebSphere. `www.ibm.com/software/websphere`.

[21] P. Kulkarni, M. Arnold, and M. Hind. Dynamic compilation: the benefits of early investing. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 94–104, New York, NY, USA, 2007. ACM.

[22] P. A. Kulkarni. JIT compilation policy for modern machines. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 773–788, New York, NY, USA, 2011. ACM.

[23] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[24] LXC - Linux Containers. `http://lxc.sourceforge.net`.

[25] Microsoft Windows Azure Node.js Developer Center. `http://www.windowsazure.com/en-us/develop/nodejs`.

[26] M. A. Namjoshi and P. A. Kulkarni. Novel online profiling for virtual machines. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '10, pages 133–144, New York, NY, USA, 2010. ACM.

[27] Node.js. `http://nodejs.org`.

[28] OpenShift by RedHat. `https://openshift.redhat.com`.

[29] OpenWRT. `https://openwrt.org`.

[30] OSGi Alliance. *OSGi Core Release 5*, 2012.

[31] M. Paleczny, C. Vick, and C. Click. The Java HotSpot Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, Berkeley, CA, USA, 2001. USENIX Association.

[32] PhoneGap. `http://phonegap.com`.

[33] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. Design and Evaluation of Dynamic Optimizations for a Java Just-In-Time Compiler. *ACM Trans. Program. Lang. Syst.*, 27(4):732–785, July 2005.

[34] V. Sundaresan, D. Maier, P. Ramarao, and M. Stoodley. Experiences with Multi-threading and Dynamic Class Loading in a Java Just-In-Time Compiler. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 87–97, Washington, DC, USA, 2006. IEEE Computer Society.

[35] Vert.x. `http://vertx.io`.