Return of the Runtimes: Rethinking the Language Runtime System for the Cloud 3.0 Era

Martin Maas
University of California, Berkeley
maas@eecs.berkeley.edu

Krste Asanović
University of California, Berkeley
krste@eecs.berkeley.edu

John Kubiatowicz
University of California, Berkeley
kubitron@eecs.berkeley.edu

ABSTRACT

The public cloud is moving to a Platform-as-a-Service model where services such as data management, machine learning or image classification are provided by the cloud operator while applications are written in high-level languages and leverage these services.

Managed languages such as Java, Python or Scala are widely used in this setting. However, while these languages can increase productivity, they are often associated with problems such as unpredictable garbage collection pauses or warm-up overheads.

We argue that the reason for these problems is that current language runtime systems were not initially designed for the cloud setting. To address this, we propose seven *tenets* for designing future language runtime systems for cloud data centers. We then outline the design of a general substrate for building such runtime systems, based on these seven tenets.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Runtime environments**; Object oriented languages; • **Hardware** → Hardware accelerators;

KEYWORDS

Managed Language Runtime Systems, Data Centers, Cloud 3.0, Platform-as-a-Service, Serverless Computing, Resource Disaggregation, FPGAs, Garbage Collection, JIT Compilation

ACM Reference format:
Martin Maas, Krste Asanović, and John Kubiatowicz. 2017. Return of the Runtimes: Rethinking the Language Runtime System for the Cloud 3.0 Era. In *Proceedings of HotOS '17, Whistler, BC, Canada, May 08-10, 2017,* 6 pages.
https://doi.org/10.1145/3102980.3103003

1 INTRODUCTION

The public cloud's deployment model is shifting from machine VMs in an Infrastructure-as-a-Service (IaaS) setting to a Platform-as-a-Service (PaaS) model where the cloud operator provides services such as databases, machine learning frameworks or speech engines, and customers access these services through APIs. This is sometimes called *Cloud 3.0* [44], reflecting the shift from Software-as-a-Service (SaaS) to IaaS to PaaS.

HotOS '17, May 08-10, 2017, Whistler, BC, Canada

ACM ISBN 978-1-4503-5068-6/17/05.
https://doi.org/10.1145/3102980.3103003

We are currently seeing this model adopted at companies such as Amazon [48], Google [43] and Microsoft [4]. This trend decouples the application from the underlying infrastructure and brings a unique opportunity for cloud operators to replace any part of the stack, including hardware, OS and language runtime system.

Emerging data center designs are already taking advantage of this, through custom hardware [9, 40] and a shift to resource disaggregation in server racks [16]. One example is TensorFlow [1]: While users can buy TensorFlow resources from Google and program against a high-level Python API, computation can run on CPUs, GPUs or custom Tensor Processing Units [25] (TPUs).

In the Cloud 3.0 setting, application developers increasingly focus on high-level functionality such as application logic, processing pipelines and statistical models, while performance-critical components such as machine learning infrastructure or data stores are managed by the cloud provider. This is reflected in the fact that application workloads are now typically written in managed *productivity languages* such as Java, JavaScript, Python or Scala, while the underlying services are run by the cloud operator and implemented in any language (including C/C++, Go or Rust), on any platform, or in hardware. This model abandons the traditional server abstraction, and *serverless* frameworks such as Amazon Lambda [49] can even deploy functions written in high-level languages directly.

This shift changes the primary role of the OS and language runtime system: they are now responsible for managing and composing a fleet of services spread across software and accelerators. While new OS concepts such as Multikernels [5] and data plane operating systems [6, 42] may help support this setting, the language runtime system has changed very little. At the same time, problems have been reported in connection with using managed languages in the cloud setting, ranging from overheads and unpredictability from GC [32] to memory bloat [37] and long startup times [29].

Several projects have tried to work around these problems using solutions such as region-based memory management [18], managing memory off-heap [13], recycling JVMs [29], or coordinating GC [32]. While these approaches fix the symptoms, they do not address the fundamental problem: most language runtimes that are used in the cloud were originally designed for different scenarios, which is reflected in the trade-offs that they make.

In this paper, we argue that we should rethink how language runtimes are designed for the Cloud 3.0 era. We do this by laying out *seven tenets* of building language runtimes for the next generation of cloud data centers. We then distill these tenets into a proposal for a shared substrate to underpin these future runtimes.

2 DATA CENTERS IN THE CLOUD 3.0 ERA

While there are competing views on how future data centers will be designed and programmed, several common trends have emerged:

2.1 Resource Disaggregation

Facebook [26], HPE [15], Huawei [10], Intel [23] and UC Berkeley [3] have proposed rack-scale system designs where resources are *disaggregated*. Instead of deploying individual servers with a certain amount of compute, memory and storage, all resources in a rack (i.e., storage, memory, accelerators and compute SoCs with a small amount of stacked high-bandwidth memory) are managed in separate pools and connected through a high-bandwidth, low-latency backplane such as PCIe or Infiniband (Figure 1a).

Compared to a traditional deployment, this reduces the number of different system configurations: Instead of managing several types of nodes with varying combinations of resources to fit the requirements of different workloads, a disaggregated system can allocate exactly the right resource mix to each application. Disaggregation also makes it possible to scale resources independently and does not require keeping idle nodes powered on to retain access to their memory or storage.

2.2 Hardware Accelerators

Recent work has shown that hardware accelerators can substantially improve certain cloud workloads, and major companies including Amazon [47], Baidu [40], Google [1] and Microsoft [9] are currently adopting them in production. These accelerators come in two flavors: custom ASICs with limited programmability (such as Google's TPU [25] or ICT's DianNao-line of chips [11]) or fully programmable FPGAs (such as Microsoft's Catapult [9]).

The deployment model also differs: while accelerators can be managed as peripherals (similar to GPUs) or pooled as a disaggregated resource, Microsoft recently proposed connecting FPGAs directly to the network (Figure 1b), which allows the FPGA to handle network requests and dispatch work to the CPU [9].

2.3 Serverless Deployment Model

Data center applications are often designed as micro-services that communicate with each other through service-level APIs. The services can be stateful or stateless and are often backed by infrastructure provided by the cloud operator, such as data stores or distributed computation frameworks.

Traditionally, these services were deployed as long-running server instances running within virtual machines or containers. However, there has been a recent shift towards a serverless deployment model, where customers implement their services as high-level functions and the cloud operator provides an orchestration framework that transparently scales and schedules these services as they see fit (e.g., Google AppEngine [20] or AWS Lambda [49]). Container-based orchestration frameworks such as Kubernetes [8] and library operating systems such as Mirage [33] have made it easier to deploy these services in a lightweight manner.

We believe that these trends will elevate the role of the language runtime system. It now becomes the component that connects applications to the services they are using, is responsible for reliably executing a large number of potentially latency-sensitive services on disaggregated hardware, and is mapping service-level APIs to a range of custom hardware accelerators.

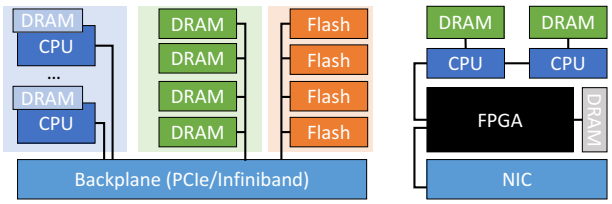

Figure 1: Cloud 3.0 Data Center Architectures

3 RETHINKING THE LANGUAGE RUNTIME

We now discuss how these changes affect language runtime systems in data centers and how they should be designed in the future. We phrase our views as seven guiding principles, or *tenets*. Each tenet represents an area where we believe that many widely-used runtime systems make the wrong trade-offs and should instead adopt different techniques – some of which exist already and some of which should be the subject of future research.

Tenet #1: Stop the JITing

Many managed language runtime systems (including the HotSpot JVM and Microsoft's CLR) use a multi-tier compilation system where bytecode is initially interpreted and "hot" regions are at runtime compiled by a Just-In-Time (JIT) compiler. Typically, these systems collect performance profiles as the code executes and continously recompile bytecode using these profiles. This enables dynamic optimizations such as trace-based inlining.

JIT compilation was the right choice when managed runtimes primarily ran short-lived applications that needed to be portable and only ran once (such as Java applets). However, cloud workloads are different: the target platform is typically known at compile time and the same application often runs many times (potentially on different machines). In this scenario, a JIT compiler primarily introduces performance unpredictability and wastes resources by re-JITing code multiple times (some data center applications spend up to 33% of time in JIT warm-up [29]).

The main argument for using a JIT in modern runtime systems is dynamic optimization, which is crucial to achieving high performance in dynamic languages such as JavaScript or (to a lesser extent) Java. However, dynamic optimization is not predicated on using a JIT compiler. Instead, the language runtime should provide a code cache that is pre-populated at installation time and survives executions. As the system runs, profiles are taken and the code is continuously recompiled *out-of-band*, allowing the same dynamic optimizations that a JIT provides, without the warm-up.

The compilation process remains largely the same (except that the JIT now has to produce relocatable code). This approach has been used by Android's ART Runtime [45], the CLR [35] and an implementation for the HotSpot JVM will be available in JDK 9 [38]. In the data center setting, the code cache could even be made available across different nodes in the cluster, to avoid redundant compilation work and allow the optimizer to take advantage of a larger set of performance profiles.

Tenet #2: Provide a Unified Data Model

Modern cloud applications run complex pipelines that can comprise multiple frameworks operating on the same data [19]. These frameworks are often written in different languages, and crossing process boundaries between them is expensive and can cause overheads. While support for sharing a single runtime system between multiple applications exists [24, 34], this approach only works if all components are written in the same language family.

Recent work proposed an alternative approach by providing a runtime-level IR and data model that allows frameworks written in different languages to target the same runtime system, which can then co-optimize across the entire application [41]. The reported speed-ups have been significant – up to 30× in some cases.

While this work looked at big data frameworks, we believe that the idea generalizes. Although languages represent objects in different ways, libraries such as Apache Arrow [2] enable sharing of data across languages, and frameworks such as MMTk [7] can support multiple object layouts and policies in the same managed environment. At the same time, frameworks like Truffle [51] allow different languages to target the same language runtime and compiler. Taken together, these tools may enable running and co-optimizing services operating on the same data.

Future runtime systems should adopt these ideas and enable frameworks written in different languages to operate on the same data, as well as enabling the compiler to optimize across them.

Tenet #3: Enable Efficient Composition

Service composition is a defining feature of cloud workloads, but managed runtimes often pay a substantial penalty when communicating across the runtime boundary. There are two different cases: calling into native code and calling into services running in other runtimes (either on the same machine or across the network).

Calling into native code requires copying and tracking objects across language boundaries (primarily for GC). It also interferes with safepoints and prevents dynamic optimizations. However, while the resulting overheads can be problematic for fine-grained communication, they are usually not a problem if a sufficient amount of work is performed per call (e.g., when calling into *numpy*). In contrast, communicating between different runtime systems additionally requires copying and serializing/deserializing arguments, which is expensive and can account for a substantial fraction of CPU cycles in communication-heavy frameworks such as Spark [39].

In a traditional client setting, this is not a problem, but in cloud data centers, connecting services that run in different runtime systems is very common, and one of the key responsibilities of the language runtime. Language runtime systems should therefore be designed with fast inter-process communication in mind, similar to processes in a traditional OS (or *Application Domains* [34] in .NET, a mechanism for data sharing between managed applications).

One way to achieve this could be by using an object layout that can be shared between multiple processes through traditional OS communication mechanisms (e.g., shared pages) and over the network (e.g., through RDMA). This could be implemented by designating parts of the address space for communication (similar to Apache Arrow), using location-independent pointers in these regions and applying a restrictive GC policy within them (to avoid expensive book-keeping). Cloud operators could then co-locate frequently communicating services that belong to the same application, such that they can use this fast communication mechanism.

Tenet #4: Go Light On the Objects

A common complaint about managed languages is that they are heavyweight, create memory bloat and spend a long time performing GC. The typical approach to these problems has been to improve or replace the GC, but we believe that this is addressing the problems from the wrong perspective. Many of these problems are the result of having too many objects, often due to unsuitable abstractions. Specifically, languages such as Java and Scala allocate a very large number of objects, by boxing basic types such as integers. This causes memory overhead from object meta-data and results in a large number of objects to trace during GC.

This is avoidable through language design. For example, providing value types [14] as part of the language can reduce the number of objects substantially (C# already supports value types, and Java support for them is expected in future versions [14]). In addition, the runtime system can provide lightweight abstractions and specializations for common data structures in the runtime library (such as linked lists), further reducing the number of objects.

In the absence of such language features, strategies such as region-based memory management [18] or decoupling control and data path [36, 37] have been successfully used to reduce the number of objects without changing the language itself.

To allow the runtime system to support these strategies, the system should divide memory into regions with different policies that can be used to reduce the number of objects. By safely providing applications and frameworks with a selection of options for managing different parts of memory, many of the cases that require a large amount of objects could likely be avoided.

Tenet #5: Manage Disaggregated Resources

One new challenge of the Cloud 3.0 data center is how to handle resource disaggregation (Section 2.1). With resources disaggregated at the rack or data-center level, application-level knowledge is required to decide how to move data between different pools of memory, including high-bandwidth memory on chip and remote memory elsewhere in the rack. While these decisions could be directly exposed to the application, the programmer may not have the information to make the best decision, and the resulting code may not be portable. Instead, previous work has explored page-based migration mechanisms [16], but those work at a coarse granularity and cannot take application-level knowledge into account.

Improving over this approach is challenging in native languages, as moving data at a finer granularity than a page requires rewriting pointers. However, managed runtime systems are perfectly suited to handle disaggregated memory for the application, as managed runtimes already have a mechanism to relocate objects and redirect pointers as necessary. This could enable them to transparently migrate different parts of the heap. Managed runtimes also have mechanisms to measure performance profiles (such as access frequencies), which makes it possible for them to dynamically decide where to place data. As such, they may be able to do a better job than the programmer, and do so transparently.

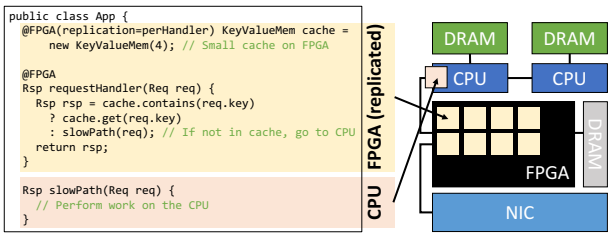

Figure 2: Programming FPGAs with high-level code

Tenet #6: Embrace Hardware Accelerators

Another challenge faced in future Cloud 3.0 data centers is how to program hardware accelerators (Section 2.2). Current language runtime systems struggle with this, as they were designed at a time when accelerators were not a common occurrence. Hence, most language runtimes neither have abstractions to expose accelerators to the application, nor to communicate with them efficiently.

Future runtime systems need to address this problem. One particularly interesting class of accelerators are FPGAs such as those available in Microsoft's Catapult system [9]. Currently, these accelerators work in tandem with the application but are programmed manually in a hardware-description language. This makes it challenging for software developers to program them effectively, and makes interfacing with the software error-prone.

Future runtime systems provide an opportunity to target CPUs, FPGAs and other accelerators simultaneously, allowing developers to program application and accelerator in the same environment. This could reduce the degree of hardware knowledge that is required, and simplify the interfacing of hardware and software components. By using dynamic optimization, such a system could even automatically schedule work across hardware accelerators and CPUs, based on dynamic performance profiles. There have been several examples of programming FPGAs in high-level languages. The closest is Dandelion [12, 46], which maps .NET LINQ queries to FPGAs. Another project, Lime [21], enabled programming of FPGAs using a Java dialect. More recently, DHDL [27] automatically generated FPGA accelerators from high-level code.

Incorporating such mechanisms into the runtime system would, for example, allow a developer to specify fast-path request handlers that run on an FPGA and automatically pass execution to the CPU if the request cannot be handled on the FPGA (Figure 2). The runtime system can then synthesize a dynamic number of instances of this event handler, enabling the FPGA to process a large number of requests in parallel (reminiscent of LINQits [12]).

An equivalent approach could be used to target the increasing amount of software-defined hardware in data centers. For example, applications for SDN-enabled switches could be synthesized from high-level applications and deployed by the runtime system.

Tenet #7: Use Concurrent Garbage Collectors

One of the most prominent problems of managed runtime systems have traditionally been the pauses introduced by garbage collection. Cloud workloads suffer from unpredictable GC pauses in different ways [18, 32], and unpredictability is often worse than lower throughout (particularly for services with a high fan-out, as stragglers introduced by GC can propagate). Different workarounds have been proposed [18, 31, 36, 37], but the only way to consistently avoid the problem of unpredictable GC pauses is to completely eliminate them by employing a pause-free collector such as C4 [50].

Runtime systems should therefore universally employ such collectors. While this lowers the overall application throughput, this effect can be alleviated by reducing GC pressure (see Tenet #4). We also believe that future hardware support may be able to perform concurrent GC with much lower overheads than software [30].

4 A NEW LANGUAGE RUNTIME SYSTEM

Based on these seven tenets, we propose a shared substrate to underpin language runtime systems for future cloud data centers. Our vision is a generic managed runtime framework that can be targeted by different language *frontends* (e.g., for Java, C#, Python or JavaScript) and supports *backends* for different CPU instruction sets, FPGAs, GPUs and other accelerators.

We believe that such a shared substrate is necessary to achieve efficient composition of services in the cloud. A common foundation allows services to communicate efficiently without incurring serialization/deserialization overheads and translating between different formats. Furthermore, as projects such as LLVM [28] and Truffle/Graal [51] have shown, building different languages around a single framework can lead to high performance and maintainability, since work can be leveraged by all these different languages.

Such a runtime system will need to build on existing technologies to find adoption. Specifically, it will have to support existing languages, applications and platforms out of the box. A clear candidate for the compiler portion of this work is LLVM [28]. It is a widely supported compiler framework, and already provides a toolkit for building managed runtime systems based on it [17]. LLVM is also being adopted in the context of a commercial JVM, underpinning Azul's Falcon compiler [22]. Finally, LLVM already supports a wide range of backends and has strong tool support.

Figure 3 shows a high-level overview of our proposed substrate. The system contains the components that all language runtimes have in common (compiler, garbage collector, etc.) and accepts frontends for different languages. When code needs to be compiled, it is handed to the frontend of the corresponding language, which will transform it into LLVM IR (as well as stack maps, meta-data for deoptimization, etc.) to pass to the compiler. We now outline how this design incorporates our seven tenets:

#1 Shared Ahead-of-time Compilation: Prior to running an application for the first time, the framework compiles its bytecode and stores the result in a code cache that is accessible by all runtimes of the user (potentially across machines). Execution profiles are continuously collected and the compiler uses them for dynamic optimization and updating the code cache. The cache can contain multiple copies of each method, and is indexed by profile.

#2 Object Layout Adapters: To support interactions between multiple languages, we propose a mechanism where frontends register *Object Layout Adapters* with the system, which are callbacks that tell the runtime how to interpret the fields of an object. This allows the runtime to optimize across code from different languages, by letting them operate on the same data (similar to Apache Arrow [2]).

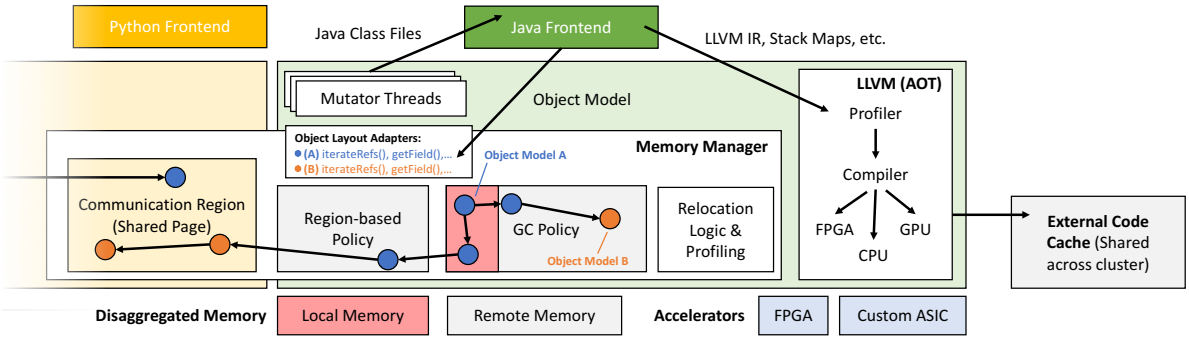

Figure 3: High-level overview of our proposed managed-language platform

#3 Communication Regions: Zero-copy communication between different applications is enabled by allocating special pages on the heap that are not subject to GC and are considered volatile. These pages can be accessed from multiple language runtime systems by mapping them into their heap. Pointers within these pages cannot point to a non-communication page and are relative to the page's base address. Since the runtime systems understand each others' object formats, this avoids serialization or copying overheads.

#4 Heterogeneous Memory Management Policies: The heap is divided into different parts, each managed by its own policy (e.g., GC or region-based). Application developers can then choose the policy that is most appropriate for a specific type of data. For example, per-iteration data in frameworks such as Apache Spark or Naiad lends itself to region-based memory management [18].

#5 Disaggregated Memory Management: Accesses to heap data are constantly profiled and commonly used data is relocated to local memory (see Tenet #5). This can reuse many of the same mechanisms as the garbage collector, and allows to manage disaggregated memory at a much finer granularity than page-based schemes.

#6 Hardware Accelerator & FPGA Targets: Specialized hardware should be supported through backends for the LLVM compiler that can target FPGAs and other accelerators. This could integrate ideas from frameworks such as DHDL or Dandelion.

#7 Fully Concurrent GC: Parts of the heap under GC policy should be managed by a fully concurrent collector, potentially supported by hardware acceleration [30].

5 DISCUSSION

Realizing our proposed substrate requires considering several questions and trade-offs. We briefly discuss some of them here.

5.1 Development Approach

Building a new runtime system from scratch is a large undertaking, and previous examples (such as HotSpot, the CLR, HHVM or JikesRVM) suggest that it can take tens to hundreds of developer years before a runtime system is competitive in terms of performance. It is therefore important to build on existing technology as much as possible. We believe that a combination of MMTk [7] and LLVM [28] could be a good foundation (it has been shown that a well-performing runtime system can be built around these two components within the scope of an academic project [17]).

5.2 Programming Model

Several of our proposed features – particularly support for hardware accelerators and heterogeneous memory management – require application changes in order to take advantage of the system. An important question is therefore how this would be integrated into existing languages. There are three fundamental options: (1) changing the language itself, (2) reusing existing language constructs but changing the compiler, or (3) a fully library-based approach.

We believe the latter to be the most promising approach, as it is the only option that can be easily generalized to a wide range of different languages. In this case, the runtime system would run conventional applications but provide a special library with annotations or API calls to support features such as region-based memory management or targeting hardware accelerators.

5.3 Real-world Adoption

A key question for any new framework is how to achieve adoption. We believe that the trend towards serverless computing and the Cloud 3.0 may facilitates this. One path to adoption could be to integrate our managed-language substrate into serverless frameworks (such as *Fission* for Kubernetes). These frameworks could then take advantage of the runtime transparently to the application.

6 CONCLUSION

As the public cloud is moving to Platform-as-a-Service, managed language runtimes are becoming ever more central to the software stack. We believe that future language runtimes in the cloud will need to target accelerators, manage disaggregated memory and compose large numbers of (potentially serverless) services efficiently. Current language runtimes are not ideally suited for this task, and future runtimes should be built on a common substrate specifically designed for the workloads of the Cloud 3.0 era.

Acknowledgements: Research was partially funded by DOE grant #DE-AC02-05CH11231, the STARnet Center for Future Architecture Research (C-FAR), and ASPIRE Lab sponsors and affiliates Intel, Google, HPE, Huawei, and NVIDIA.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*.

[2] Apache Arrow. 2017. Powering Columnar In-Memory Analytics. (2017). https://arrow.apache.org/

[3] Krste Asanovic and D Patterson. 2014. Firebox: A hardware building block for 2020 warehouse-scale computers. In *USENIX FAST*, Vol. 13.

[4] Microsoft Azure. 2017. Machine Learning. (2017). https://azure.microsoft.com/en-us/services/machine-learning/

[5] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *22nd Symposium on Operating Systems Principles (SOSP '09)*.

[6] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*.

[7] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *26th International Conference on Software Engineering (ICSE '04)*.

[8] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* 59, 5 (April 2016), 50–57.

[9] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. 2016. A Cloud-Scale Acceleration Architecture. In *49th Annual IEEE/ACM International Symposium on Microarchitecture*.

[10] Huawei Press Center. 2017. Huawei proposed DC 3.0 architecture of future data center to meet the requirement of real-time data processing in big data era. (2017). http://pr.huawei.com/en/news/hw-423134-3.0.htm

[11] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2016. DianNao Family: Energy-efficient Hardware Accelerators for Machine Learning. *Commun. ACM* 59, 11 (Oct. 2016), 105–112.

[12] Eric S. Chung, John D. Davis, and Jaewon Lee. 2013. LINQits: Big Data on Little Clients. In *40th International Symposium on Computer Architecture (ISCA '13)*.

[13] Databricks. 2015. Project Tungsten: Bringing Spark Closer to Bare Metal. (2015). https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html

[14] Jesper de Jong. 2015. Project Valhalla – Value Types. (2015). http://www.jesperdj.com/2015/10/04/project-valhalla-value-types/

[15] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. 2015. Beyond Processor-centric Operating Systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*.

[16] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network Requirements for Resource Disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*.

[17] Nicolas Geoffray, Gaël Thomas, Julia Lawall, Gilles Muller, and Bertil Folliot. VMKit: A Substrate for Managed Runtime Environments. In *6th ACM International Conference on Virtual Execution Environments (VEE '10)*.

[18] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Viswani, Dimitrios Vytiniotis, Ganesan Ramalingan, Manuel Costa, Derek Murray, Steven Hand, and Michael Isard. 2015. Broom: sweeping out Garbage Collection from Big Data systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*.

[19] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P. Grosvenor, Allen Clement, and Steven Hand. 2015. Musketeer: All for One, One for All in Data Processing Systems. In *EuroSys '15*.

[20] Google. 2017. Google App Engine: Platform as a Service. (2017). https://developers.google.com/appengine

[21] Shan Shan Huang, Amir Hormati, David F. Bacon, and Rodric Rabbah. 2008. Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In *ECOOP '08*.

[22] InfoQ. 2017. Azul Systems Launches Falcon, a New Just-in-Time Compiler for Java, Based on LLVM. (2017). https://www.infoq.com/news/2017/05/azul-falcon

[23] Intel. 2017. Intel® Rack Scale Design. (2017). http://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html

[24] Mick Jordan, Laurent Daynès, Grzegorz Czajkowski, Marcin Jarzab, and Ciarán Bryce. 2004. *Scaling J2EE Application Servers with the Multi-tasking Virtual Machine*. Technical Report. Sun Microsystems, Inc., Mountain View, CA, USA.

[25] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, et al. 2017, arXiv:1704.04760. In-Datacenter Performance Analysis of a Tensor Processing Unit. (2017, arXiv:1704.04760).

[26] Data Center Knowledge. 2013. Meet the Future of Data Center Rack Technologies. (Feb. 2013). http://www.datacenterknowledge.com/archives/2013/02/20/meet-the-future-of-data-center-rack-technologies/

[27] David Koeplinger, Christina Delimitrou, Raghu Prabhakar, Christos Kozyrakis, Yaqi Zhang, and Kunle Olukotun. 2016. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. In *43rd International Symposium on Computer Architecture (ISCA '16)*.

[28] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*.

[29] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. 2016. Don't Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*.

[30] Martin Maas, Krste Asanovic, and John Kubiatowicz. Grail Quest: A New Proposal for Hardware-assisted Garbage Collection. In *6th Workshop on Architectures and Systems for Big Data (ASBD '16)*.

[31] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. 2016. Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*.

[32] Martin Maas, Tim Harris, Krste Asanovic, and John Kubiatowicz. Trash Day: Coordinating Garbage Collection in Distributed Systems . In *5th Workshop on Hot Topics in Operating Systems (HotOS XV)*.

[33] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*.

[34] Microsoft Developer Network. 2017. Application Domains. (2017). https://msdn.microsoft.com/en-us/library/2bh4z9hs(v=vs.110).aspx

[35] Microsoft Developer Network. 2017. Ngen.exe (Native Image Generator). (2017). https://msdn.microsoft.com/en-us/library/6t9t5wcf(v=vs.110).aspx

[36] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A High-Performance Big-Data-Friendly Garbage Collector. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*.

[37] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. 2015. FACADE: A Compiler and Runtime for (Almost) Object-Bounded Big Data Applications. In *20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*.

[38] OpenJDK. 2017. JEP 295: Ahead-of-Time Compilation. (2017). http://openjdk.java.net/jeps/295

[39] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *12th Symposium on Networked Systems Design and Implementation (NSDI '15)*.

[40] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, Song Jiang, undefined, undefined, undefined, and undefined. 2014. SDA: Software-defined accelerator for large-scale DNN systems. *2014 IEEE Hot Chips 26 Symposium* (2014).

[41] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. Weld: A Common Runtime for High Performance Data Analytics. In *8th biennial Conference on Innovative Data Systems Research (CIDR '17)*.

[42] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*.

[43] Google Cloud Platform. 2017. Cloud Dataflow - Batch & Stream Data Processing. (2017). https://cloud.google.com/dataflow/

[44] Tom's IT Pro. 2016. Cloud 3.0 And 'Building Scale' At Interop. (May 2016). http://www.tomsitpro.com/articles/interop-cloud-3-building-scale,1-3277.html

[45] Android Open Source Project. 2017. Implementing ART Just-In-Time (JIT) Compiler. (2017). https://source.android.com/devices/tech/dalvik/jit-compiler.html

[46] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *24th ACM Symposium on Operating Systems Principles (SOSP '13)*.

[47] Amazon Web Services. 2017. Amazon EC2 F1 Instances. (2017). http://aws.amazon.com/ec2/instance-types/f1/

[48] Amazon Web Services. 2017. Amazon Machine Learning - Predictive Analytics with AWS. (2017). http://aws.amazon.com/machine-learning/

[49] Amazon Web Services. 2017. AWS Lambda - Serverless Compute. (2017). http://aws.amazon.com/lambda/

[50] Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: The Continuously Concurrent Compacting Collector. In *ISMM '11*.

[51] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-optimizing Runtime System. In *3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12)*.