The Case for the Holistic Language Runtime System

Martin Maas* Krste Asanović* Tim Harris† John Kubiatowicz*

* University of California, Berkeley † Oracle Labs

ABSTRACT

We anticipate that, by 2020, the basic unit of warehouse-scale cloud computing will be a rack-sized machine instead of an individual server. At the same time, we expect a shift from commodity hardware to custom SoCs that are specifically designed for the use in warehouse-scale computing. In this paper, we make the case that the software for such custom rack-scale machines should move away from the model of running managed language workloads in separate language runtimes on top of a traditional operating system but instead run a distributed language runtime system capable of handling different target languages and frameworks. All applications will execute within this runtime, which performs most traditional OS and cluster manager functionality such as resource management, scheduling and isolation.

1. INTRODUCTION

We introduce *Holistic Language Runtime Systems* as a way to program future cloud platforms. We anticipate that over the next years, cloud providers will shift towards rack-scale machines of custom designed SoCs, replacing the commodity parts in current data centers. At the same time, we expect a shift towards interactive workloads developed by third-party, non-systems programmers. This leads to a programming crisis as *productivity programmers* have to develop for increasingly complex and opaque hardware (Section 2).

We believe that, as a result, the cloud will be exclusively programmed through high-level languages and frameworks. However, today's software stack is ill-suited for such a scenario (Section 3): the conventional approach of running each application in a separate runtime system leads to inefficiency and unpredictable latencies due to interference between multiple runtime instances, and prevents optimizations.

We propose to solve this problem through running all applications in a rack-wide distributed runtime system. This *holistic* runtime (Section 4) consists of a per-node runtime that executes all workloads within that node, and a distributed inter-node runtime that coordinates activities between them (Figure 1). The per-node runtime takes over the functionality of a traditional OS (such as resource management, scheduling and isolation), while the distributed layer coordinates activities between the different nodes (similar to a cluster manager, but with low-level control of the system). This enables to e.g. coordinate garbage collection between nodes to not interfere with low-latency RPCs, share JIT results, or transparently migrate execution and data.

We discuss the merits of this model and show challenges that need to be addressed by such a system (Section 5).

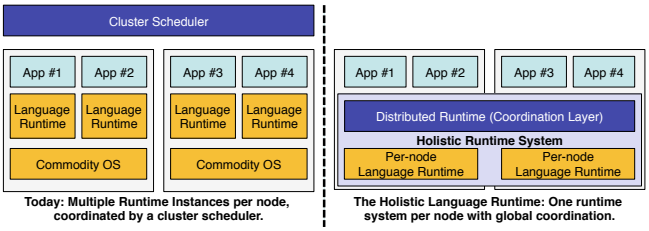

Figure 1: Comparing today's software stack and holistic runtimes

2. CLOUD COMPUTING IN 2020

Warehouse-scale computers in the cloud are becoming the backbone for a growing portion of applications. We expect this trend to continue: in this section, we identify a set of developments that, we believe, will lead to significant changes to cloud infrastructure over the next 5-10 years.

2.1 Different Cloud Hardware

As the market for IaaS and PaaS cloud services is growing, and an increasing number of organizations and developers are adopting the cloud, we expect providers to shift away from using commodity parts. While this kind of hardware has been the most economical solution for the last decade, the overheads of replicated and often unused components (such as unnecessary interfaces, motherboards or peripherals) lead to significant inefficiency in energy consumption, hardware costs and resource utilization. We expect that due to the increasing scale of warehouse-scale computing, it will become more economical to design custom SoCs for data centers (either as a third-party offering, or designed from stock IP by the cloud provider itself). This has been facilitated by the availability of high-quality IP (such as ARM server processors) and would enable to produce SoCs with 10s of cores, accelerators, on-chip NICs integrated into the cache hierarchy (such as [38]) and DRAM controllers that enable remote accesses without going through a CPU.

We envision 100 to 1,000 of these SoCs to be connected in a rack-scale environment with a high-bandwidth/low-latency interconnect. Such a rack of SoCs will be the new basic unit of the data center. While the different SoCs are tightly coupled and may provide a global address space, remote caching may be limited. At the same time, I/O and bulk storage are moved to the periphery of the rack and accessed over the network. Compared with today's clusters, advantages of such a system include better energy efficiency and better predictability due to flatter interconnects enabled by the rack-scale form factor (such as a rack-level high-radix switch) and being able to integrate NICs into the SoC.

The rack-scale setup bears similarity to systems such as the HP Moonshot [4] or the AMD SeaMicro SM10000 [41], but with custom SoCs. This trend towards rack-scale systems extends to the research community: for example, an increasing amount of work is looking at scale-out architectures for memory [38] and CPUs [35]. Storage systems are changing as well, with DRAM becoming the main base for storage [45] and new low-latency non-volatile memory technologies on the horizon [19]. Research is also looking at accelerators to make use of available chip area and tackle the power wall [34, 48], and custom SoCs enable inclusion of accelerators for common cloud workloads, such as databases [34]. However, this increases the degree of heterogeneity.

→ **The cloud is becoming more opaque**: It is likely that details of the SoC design will be proprietary and programmers will have less knowledge about the hardware they run on. Even today it is difficult (and not desirable for portability) to fine-tune programs to the underlying hardware [31]. However, as cloud providers move to custom SoCs, it will become infeasible. This creates a programmability crisis: programmers will have to program a platform they know little about, while the complexity of the system is growing.

2.2 Different Mix of Cloud Workloads

Today, the majority of workloads running in cloud providers' data centers are developed in-house, such as Google Search or Microsoft Hotmail. Developers of these workloads have access to deep knowledge about the underlying infrastructure and can fine-tune them to the platform if necessary. These workloads are often interleaved with IaaS jobs from external customers, through platforms such as AWS [1], AppEngine [3] or Azure [6]. Some cloud providers (such as Rackspace) even cater exclusively to external customers.

By 2020, we envision that these third-party workloads will make up the majority of data center workloads, just like third-party apps today make up the majority of apps on any smartphone or PC. We see a shift towards ubiquitous computing, with large numbers of apps for smartphones, watches or TVs that interact with a growing number of sensors and use the cloud as backend. At the same time, new computation and data intensive *interactive* applications emerge, such as augmented reality, remote gaming or live translation.

This significantly changes the mix of data center workloads. Today, latency-sensitive workloads are relatively rare, and can be run on their own servers, with an ample supply of batch workloads sharing the other machines. However, by 2020, interactive workloads will be in the majority, requiring low latency response times while operating on large amounts of data. Further, there will be a much larger and more heterogeneous set of workloads running in the cloud, and the majority of them will be supplied by external developers.

Like today, these workloads will require performance guarantees, tail-tolerance [22], failure tolerance (e.g. through replication) and security guarantees (such as privacy through encryption [8] and isolation between applications [43]). Furthermore, there is increasing adoption of service-oriented architectures (SOAs) to increase productivity and reduce risk. We expect that this adoption will continue, requiring cloud platforms to handle such service interfaces efficiently.

→ **Radical over-provisioning will cease to be cost-effective**: Cloud providers today have to radically over-provision resources. This is happening for two reasons:

First, the high degree of complexity in the system can lead to high tail-latencies. This is unacceptable for interactive jobs, which hence require over-provisioning and other techniques to be tail-tolerant [22]. This approach works today, since interactive jobs make up a relatively small part of the mix. However, as the portion of interactive jobs is growing, more resources are wasted for over-provisioning and replication, which quickly becomes economically infeasible.

Second, load on the warehouse-scale system varies greatly between different times of the day, week and year, and cloud providers need to provision for peak times. Shutting off individual machines is difficult, since workloads and storage is spread across many machines and the overhead of consolidating them before shutting off may be too large.

Due to the rising cost of over-provisioning, we expect that much of it will have to be replaced by fine-grained sharing of resources – both to reduce tail-latency and to make it possible to consolidate workloads on a smaller set of nodes (and switch off unused parts of a running system). This implies a need to reduce detrimental interference between jobs on a highly loaded machine, which is a challenging problem [27].

2.3 Different Group of Cloud Programmers

An increasing portion of cloud workloads is programmed by *productivity programmers* [20], similar to PHP webapps in 2000 and iOS apps today. A large portion of smartphone apps already use the cloud as backend and high-level frameworks and languages are widely used when programming for the cloud – examples include the AppEngine stack [3], Hadoop [51], DryadLinq [55] and Spark [56]. The programming languages of choice include Java, C# and Python, which are all high-level managed languages.

The reason these frameworks and languages are popular is that except for an expert programmer deeply familiar with the underlying system, it is prohibitively difficult to handle its complexity (e.g. distribution, replication, synchronization and parallelism) while maintaining productivity.

We expect that in 2020, the majority of data center workloads will be written by productivity programmers. Expert programmers deeply familiar with the system will exist, but will be creating a smaller portion of the workloads themselves. They will instead provide frameworks for productivity programmers to use and program workloads for companies large enough to run their own data centers.

→ **The cloud will be exclusively programmed using high-level languages and frameworks**: While high-level languages and frameworks are already widely used to program the cloud, they will become the only way to develop for future cloud platforms, due to the nascent programmability gap of productivity programmers having to program complex opaque systems. High-level managed languages bridge the gap between software and hardware, as they abstract away hardware details and have more flexibility in adapting code to the underlying platform. This makes the problem of programming increasingly complex cloud architectures tractable and facilitates moving between cloud providers.

We therefore expect that cloud applications in 2020 will be almost exclusively written in high-level managed languages and use high-level frameworks for storage, parallelism, distribution and failure/tail tolerance. Instead of providing customers with machine VMs, cloud providers will expose high-level APIs that give access to frameworks which the provider implements however it chooses (a PaaS model).

3. LANGUAGE RUNTIMES TODAY

We are already seeing the start of some of the trends identified in Section 2. In particular, high-level managed languages and frameworks are already a main way to program the cloud (Section 2.3). Arguably, the current source of this popularity are their good productivity and safety properties. However, they also give the runtime system additional freedom in tuning workloads to the underlying hardware. This property becomes increasingly important on platforms that cannot be hand-tuned for, such as the rack-scale machines we are envisioning. Specifically, the advantages of managed language workloads include the following:

- *They add a level of abstraction*: Managed languages hide the complexity of the underlying architecture. This is often considered a disadvantage (since it prevents some low-level tuning), but in the cloud scenario, it is necessary as hand-tuning will not be possible.
- *They operate on Bytecode*: This allows transparent recompilation and auto-tuning to a particular architecture based on runtime performance counters (using a JIT). There are also high-level frameworks such as SEJITS [20] or Dandelion [44] that can help to program accelerators in a high-level language by using introspection into compiled programs.
- *They provide automatic memory management*: Explicit memory management is prone to errors, and bugs such as memory leaks or buffer overruns can be particularly damaging in a cloud scenario of long-running, security-sensitive workloads. At the same time, there is little benefit of explicit management in an opaque system – automatic memory management and garbage collection (GC) are therefore a significant advantage.
- *They operate on references instead of pointers*: This allows transparent migration of data and execution between different nodes and automatic forwarding. It also gives more flexibility for memory management, as it allows data to be relocated.

Unfortunately, the way managed languages are integrated into today's software stack has inefficiencies and is a bad fit for future cloud data centers, with large (rack-scale) machines, high load and a requirement for fine-grained resource sharing. In today's deployments, applications run in isolated containers (depending on the platform, these are VMs, resource containers [14] or processes), and each container runs a separate instance of the language runtime for that application (Figure 2). This causes four problems:

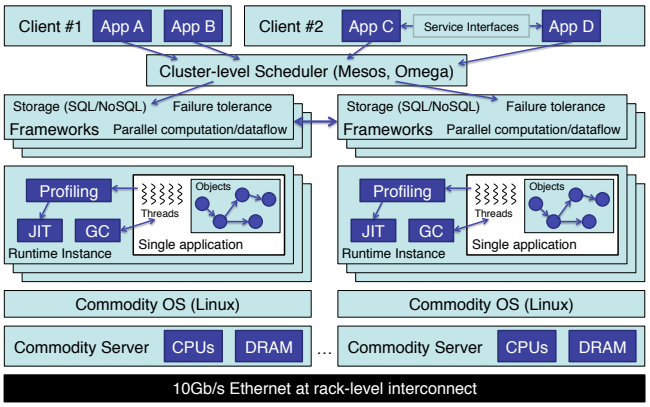

Figure 2: The current software stack

The Interference Problem: As cloud workloads exhibit an increasing spectrum of resource requirements, it becomes increasingly important to be able to run a large number of different applications on the same machine (to consolidate workloads and for load balancing reasons). For parallel workloads, co-scheduling of applications on the same machine can lead to detrimental interference through scheduling [27, 39] (in addition to other sources of interference such as memory bandwidth or caches). This applies to managed language runtimes as well: for example, GC threads can get descheduled at inopportune times (and stall the application) or application servers receiving an RPC may not be scheduled when the RPC arrives. The OS cannot schedule around this because the required high-level information about the threads is not available to the OS scheduler.

In addition, when there are more runtime instances than hardware threads, it can be challenging to dispatch a request to the right instance, and the overhead of waking up that process adds latency. This makes RPC unsuitable for fine-grained communication with remote calls on the order of microseconds. However, in a service-oriented architecture, there will be a very large number of service calls with low latency requirements. Work on the hardware level tackled this for RDMA [38], but does not generalize to RPCs yet.

The Redundancy Problem: When running multiple language runtimes, a large amount of code in shared libraries is loaded and JITed multiple times (for example, OpenJDK 7 loads 379 classes for a minimal "Hello World" program). Since the JITed code is tied into the logical data structures of the runtime, page sharing between processes cannot avoid this problem. Additionally, each runtime instance runs its own service daemons such as JIT compiler or profiler, adding further overheads. Finally, VM images often contain an entire OS with a large amount of unused libraries and code, even if only a small subset is used (which can lead to three orders of magnitude overhead in binary size [36]).

The Composability Problem: For productivity and maintainability, applications are often broken into a large number of services – e.g. page requests to Amazon typically require over 150 service calls [23]. However, when services are shared between different runtime instances, a service call requires crossing the boundary between two processes. This prevents optimizations such as code inlining or service fusion, and adds overheads from context switches and scheduling. At the same time, simply putting all services into the same process would cause problems in current systems, such as losing failure isolation and not being able to dynamically migrate service instances between nodes.

The Elasticity Problem: Adding servers in response to increased load involves launching new VMs, which have a significant startup latency and take time to warm up (e.g. JITing the code). This makes it harder to react to varying workload demands at a fine granularity.

Taken together, these problems show that the current approach of running each application in its own runtime system appears infeasible for the cloud in 2020 (and already causes inefficiency today). We aim to solve these problems through the introduction of a *Holistic Language Runtime System*.

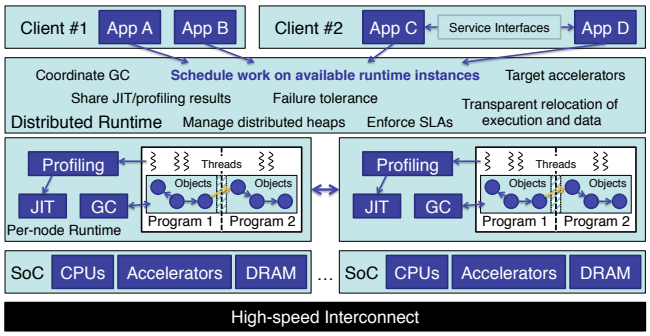

Figure 3: High-level Overview of a Holistic Language Runtime

4. HOLISTIC LANGUAGE RUNTIMES

There is a sense that the changing cloud computing landscape provides an opportunity for a new approach to the software stack. Some work proposes new OS structures that span multiple nodes/cores and enable inter-node coordination such as gang-scheduling – examples include Akaros [42], Barrelfish [15] and DiOS [46]. The service-oriented model has also been adopted by several research OSs (such as fos [50] or Tessellation [21]). At the same time, library OSs on hypervisors (like Mirage [36] or OSv [7]) tackle the redundancy problem by compiling application and OS together and removing layers of indirection.

We are exploring a different approach: we believe that the problems from Section 3 can be tackled by having all jobs logically share a distributed language runtime that manages all applications in a rack-scale machine. In this model, all threads, heaps and applications are exposed to the same runtime system and live in the same address space (assuming a global, but not necessarily cache-coherent, address space). Applications are supplied as a set of (PaaS-style) services[1] that are restricted to a single coherence domain and can be replicated. The runtime system consists of two parts: a per-node runtime on each SoC that runs all workloads within that node, and a distributed runtime that spans all nodes in a distributed fashion, enabling fine-grained coordination between SoCs. We call this system the "Holistic Language Runtime System", for its holistic view of the rack (Figure 3).

Compared to the current model (Figure 2), the per-node runtime and the OS merge. Services become the unit of virtualization and the runtime coordinates resource allocation and device access (in particular the NIC) between them. Storage and drivers are implemented as libraries, similar to the Exokernel [24]. This proposal is supported by the fact that many traditional OS services have already been pushed out of the OS: for example, many scheduling tasks have been moved into the application level (e.g. through cluster schedulers such as Mesos [28] or Omega [47]) and many virtualization tasks have been pushed into hardware (e.g. Infiniband NICs). Within each per-node runtime, applications are protected through software-based isolation, with controlled references between sub-heaps (and potentially hardware support in the SoC, such as Mondrian-style protection [53]).

In contrast, the distributed runtime coordinates the different per-node runtime instances. This includes tasks traditionally performed by the cluster manager (such as load balancing), but also coordinated decisions such as when to run GC on a part of the heap, how to relocate or resize applications and sharing profiling/optimization results. While the distributed runtime can make use of the fact that it can access the entire shared address space (e.g. for zero-copy transfers), each SoC is treated as a separate fault domain and the distributed runtime has to operate using traditional agreement protocols (similar to a Multikernel [15]).

This approach has a number of advantages and solves the four problems from Section 3. We believe that a holistic runtime system will enable the following features:

- Transparent migration of execution and data between nodes of the system while preserving a PaaS view. This can exploit locality in workloads, and interacts with other mechanisms such as performance guarantees.
- Solving the redundancy problem through reusing JIT results or profiling data across multiple nodes and consolidating runtime services within the rack.
- Fine-grained coordination and interleaving of parallel jobs [27], alleviating the interference problem.
- Coordinating GC between nodes (i.e. scheduling GC to not interrupt critical operations, and avoid descheduling threads in certain GC phases). A holistic runtime can also track and collect references across nodes.
- Low-latency RPCs ($\sim$ 1us per RPC call) in the language runtime (to solve the composability problem). This is not currently possible under load and requires scheduling handlers from different applications very quickly, without the overhead of a context switch.
- Service fusion and other dynamic optimizations that become possible by having high-level information available (similar to those performed in Optimus [33]) and sharing a JIT compiler between jobs. This can be guided by higher-level measurements and semantics as in Jockey [25] or Pacora [17], and enables auto-tuning. This also targets the composability problem.
- A unified mechanism for failure handling instead of each framework having its own. This would allow to coordinate failure handling between applications (such as replicating to the same racks so that everything can be migrated at once in the case of correlated failures).
- Adding resources at a much finer granularity (additional threads instead of VMs). This allows scaling much more smoothly to solve the elasticity problem.

Most of these advantages also apply to conventional rack-scale clusters with RDMA, but benefit from tight coupling of nodes, low latencies and hardware support – the custom rack-scale machines we envision therefore pose an opportunity to exploit these advantages even better.

Holistic runtimes bear some resemblance to Singularity [30] and MVM [32] (and the debate about the precise split between OS and runtime is, in fact, much older [29]). However, in contrast to this work, we target rack-scale machines, which bring some unique challenges (Section 5). The holistic runtime is also related to past work on distributed JVMs [12, 13]. While these JVMs aim to provide a single-system view at the level of the JVM interface, we aim to provide a single-system view at the framework and service level.

[1]Here, a service is a monolithic unit that communicates only through a service interface. Services access bulk DRAM through RDMA-style transfers and communicate through RPCs (more complex patterns can be provided by libraries).

5. CHALLENGES & FUTURE WORK

There are many challenges that need to be addressed to implement a holistic runtime and make it work in a real deployment. Many of these challenges apply to conventional systems as well, but can benefit from the coordination abilities and high-level information available to a holistic runtime.

Communication: The communication model between services is crucial, since it has to enable low-latency communication within and between nodes, but also strong isolation between applications. Singularity provided this isolation through channel contracts, but incurred performance hits due to the lack of pipelining, context switch overheads and not being able to express certain asynchronous communication patterns. This problem may be solved through a more expressive contract language and hardware support for ultra low-latency RPCs (e.g. inspired by work on active messages [49] and LRPC [16]) – low-latency access to the NIC from a managed runtime is another important challenge. At the same time, it is important to keep the programmer-facing model simple, to not harm productivity.

Garbage Collection: Objects can be spread across multiple nodes and have references between them. This may require approaches for cross-node garbage collection, a largely unexplored area. Approaches to divide the heap into isolated portions will be crucial, as a rack-wide heap would be infeasible to garbage-collect, and collection of tera- or petabyte sized heaps is an unsolved problem. However, research has looked at garbage collection for NUMA systems [26] and DRAM-based storage systems [45]. Some of these solutions may apply to the rack-scale scenario as well.

Fault isolation and life-cycle support: It is essential that failures in a single application, a hardware component (e.g. an SoC) or a runtime instance do not bring down the entire rack-scale machine. In addition, it must be possible to update parts of the system without shutting down the entire rack (a rack is a much larger unit than an individual server, and shutting it down incurs a higher cost). These problems are already being recognized in Java, were summarized in JSR-121 [5] and implemented in the Multi-tasking VM project [32] and commercial products [9]. They could be addressed by the holistic runtime through e.g. isolated per-application data structures and language-level containers.

Security and isolation: A holistic runtime needs to provide security guarantees. While applications live in the same address space, they need to be logically isolated, potentially with hardware support to harden software-based security. This challenge already exists in data centers today: just like today's cloud servers, the rack-level machine will be shared between multiple entities, and it is important to avoid side-channels between tenants and encrypt all data in memory. Performance interference is important as well, to prevent applications from impacting another's performance (either maliciously – such as a DoS attack – or through failure).

Extensibility: Expert programmers should be able to implement new optimizations and auto-tuners, potentially using some knowledge of the hardware. This is required to allow the runtime to make optimal use of the platform it is running on. Projects like Graal [54] can help with this by exposing low-level runtime functionality to frameworks. As projects such as the Jikes RVM [10] show, writing low-level systems frameworks is possible in high-level, safe languages.

Backward-compatibility: It must be possible to run existing frameworks (such as Naiad [37] or Cassandra [2]) and applications on top of the system, with minimum changes. The compatibility layer is the language runtime (e.g. Java Bytecode and Class library). However, how to best structure this interface is an open question, as is integrating other (e.g. per-tenant) runtimes, similar to the goal of Mesos [28].

For backward-compatibility, it is necessary to emulate a vast amount of existing functionality in class libraries, which uses syscalls and OS functionality. One approach is to have an instance of a legacy OS running in a container, to handle functionality not handled by the holistic language runtime (this is the approach taken in Libra [11]). An alternative is to handle them with a library OS such as Drawbridge [40].

Support for different languages: Since the holistic language runtime will run most workloads on the rack-scale machine, it must support all languages that people will use in their applications. Precedents for this type of system exist: Microsoft already uses a cross-language approach with their CLR and projects like Truffle [52] can help to support additional languages in a runtime. Frameworks like MMTk [18] can support GC in a language-independent way.

Tail-tolerance: Dean and Barroso define tail-tolerance as "[creating] a predictably responsive whole out of less predictable parts" [22]. Tail-tolerance is crucial for cloud workloads, particularly interactive ones. A holistic runtime needs to be able to support techniques for providing tail tolerance, such as hedging requests or selective replication.

Performance Guarantees: The system needs to be able to provide probabilistic performance guarantees to services. However, providing such guarantees for managed language workloads (in a way that is compositional) is difficult, in particular in the presence of a large portion of interactive workloads. One question is how guarantees are to be expressed. For example, they could be provided as a service agreement when an application enters the system (similar to the Tessellation OS [21]) or specified as high-level application goals as in Pacora [17]. There are many aspects to enforcing these guarantees, such as handling interference between applications while taking GC, JIT compilation and other hard-to-predict aspects into account. There also needs to be a way to respond to unpredicted interference (e.g. in the cache hierarchy), such as dynamically relocating jobs [57].

Resource management: The runtime needs to make decisions when and where to migrate data or execution. This is similar to the classic resource allocation problem, as well as cluster-level scheduling [47]. However, trade-offs are different in a rack-scale machine, where migration to a different SoC may be cheaper than in a traditional cluster. The runtime also has to colocate data and computation to reduce communication. Another challenge is how to handle applications that span more than a single rack – these cases still require the equivalent for cluster-level scheduler that needs to interact with the holistic runtimes within each rack.

In conclusion, we make the case that future warehouse-scale cloud computing will be performed on rack-scale machines and that the traditional software stack will be unsuitable. We instead propose to run cloud workloads within a holistic language runtime. We are investigating such runtimes on both software and hardware level and are planning to implement a JVM-based prototype of a holistic runtime system.

Acknowledgements

We would like to thank Scott Beamer, Juan Colmenares, Kim Keeton, Margo Seltzer and the anonymous reviewers for their feedback on earlier drafts of this paper. We would also like to thank Philip Reames and Mario Wolczko for discussions about managed language runtime systems. Additional thanks goes to the FireBox group at UC Berkeley for discussion and input on the future of rack-scale platforms.

References

[1] "Amazon web services." [Online]. Available: http://aws.amazon.com/

[2] "The apache cassandra project." [Online]. Available: http://cassandra.apache.org/

[3] "Google app engine: Platform as a service."

[4] "HP moonshot system." [Online]. Available: www.hp.com/go/moonshot

[5] "JSR-000121 application isolation API specification." [Online]. Available: https://jcp.org/aboutJava/communityprocess/final/jsr121/

[6] "Microsoft windows azure." [Online]. Available: http://www.windowsazure.com/

[7] "OSv: designed for the cloud." [Online]. Available: http://osv.io/

[8] *Software Guard Extensions Programming Reference*. [Online]. Available: http://software.intel.com/sites/default/files/329298-001.pdf

[9] "Waratek - cloud virtualization and java specialists." [Online]. Available: http://www.waratek.com/

[10] B. Alpern, S. Augart, S. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp, "The jikes research virtual machine project: Building an open-source research community," *IBM Systems Journal*, vol. 44, no. 2, 2005.

[11] G. Ammons, J. Appavoo, M. Butrico, D. Da Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. Van Hensbergen, and R. W. Wisniewski, "Libra: A library operating system for a jvm in a virtualized execution environment," in *Proceedings of the 3rd International Conference on Virtual Execution Environments*, ser. VEE '07, 2007.

[12] J. Andersson, S. Weber, E. Cecchet, C. Jensen, and V. Cahill, "Kaffemik - a distributed JVM featuring a single address space architecture," in *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, ser. JVM'01, 2001.

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

[14] G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: A new facility for resource management in server systems," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99, 1999.

[15] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: A new OS architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09, 2009.

[16] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy, "Lightweight remote procedure call," *ACM Trans. Comput. Syst.*, vol. 8, no. 1, Feb. 1990.

[17] S. L. Bird and B. J. Smith, "PACORA: performance aware convex optimization for resource allocation," in *Proceedings of the 3rd USENIX Workshop on Hot Topics in Parallelism (HotPar: Posters)*, 2011.

[18] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Oil and water? high performance garbage collection in java with MMTk," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04, 2004.

[19] G. Burr, B. Kurdi, J. Scott, C. Lam, K. Gopalakrishnan, and R. Shenoy, "Overview of candidate device technologies for storage-class memory," *IBM Journal of Research and Development*, vol. 52, no. 4.5, Jul. 2008.

[20] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox, "SEJITS: getting productivity and performance with selective embedded JIT specialization," *Programming Models for Emerging Architectures*, 2009.

[21] J. A. Colmenares, G. Eads, S. Hofmeyr, S. Bird, M. Moretó, D. Chou, B. Gluzman, E. Roman, D. B. Bartolini, N. Mor, K. Asanović, and J. D. Kubiatowicz, "Tessellation: Refactoring the OS around explicit resource containers with continuous adaptation," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13, 2013.

[22] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, Feb. 2013.

[23] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07, 2007.

[24] D. R. Engler, M. F. Kaashoek, and J. O'Toole,Jr., "Exokernel: An operating system architecture for application-level resource management," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95, 1995.

[25] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: guaranteed job latency in data parallel clusters," in *Proceedings of the 7th ACM european conference on Computer Systems*, ser. EuroSys '12, 2012.

[26] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro, "A study of the scalability of stop-the-world garbage collectors on multicores," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13, 2013.

[27] T. Harris, M. Maas, and V. J. Marathe, "Callisto: co-scheduling parallel runtime systems," in *Proceedings of the 9th ACM European Conference on Computer Systems*, ser. EuroSys '14, 2014.

[28] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: a platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, ser. NSDI'11, 2011.

[29] J. Howell and M. Montague, "Hey, you got your language in my operating system!" Dartmouth College, Tech. Rep., 1998.

[30] G. C. Hunt and J. R. Larus, "Singularity: rethinking the software stack," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, Apr. 2007.

[31] K. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. Wright, "Performance analysis of high performance computing applications on the amazon web services cloud," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, Nov. 2010.

[32] M. Jordan, L. Daynès, G. Czajkowski, M. Jarzab, and C. Bryce, "Scaling J2EE application servers with the multitasking virtual machine," Sun Microsystems, Inc., Tech. Rep., 2004.

[33] Q. Ke, M. Isard, and Y. Yu, "Optimus: A dynamic rewriting framework for data-parallel execution plans," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13, 2013.

[34] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.

[35] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi, "Scale-out processors," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12, 2012.

[36] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13, 2013.

[37] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, 2013.

[38] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[39] H. Pan, B. Hindman, and K. Asanović, "Composing parallel software efficiently with lithe," in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10, 2010.

[40] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library OS from the top down," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, 2011.

[41] A. Rao, "System overview for the SM10000 family," Jul. 2011. [Online]. Available: http://www.seamicro.com/sites/default/files/TO2_SM10000_System_Overview.pdf

[42] B. Rhoden, K. Klues, D. Zhu, and E. Brewer, "Improving per-node efficiency in the datacenter with new OS abstractions," in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, ser. SOCC '11, 2011.

[43] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09, 2009.

[44] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly, "Dandelion: A compiler and runtime for heterogeneous systems," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, 2013.

[45] S. M. Rumble, A. Kejriwal, and J. Ousterhout, "Log-structured memory for DRAM-based storage," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, 2014.

[46] M. Schwarzkopf, M. P. Grosvenor, and S. Hand, "New wine in old skins: The case for distributed operating systems in the data center," in *Proceedings of the 4th Asia-Pacific Workshop on Systems*, ser. APSys '13, 2013.

[47] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13, 2013.

[48] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV, 2010.

[49] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: A mechanism for integrated communication and computation," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ser. ISCA '92, 1992.

[50] D. Wentzlaff, C. Gruenwald,III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal, "An operating system for multicore and clouds: Mechanisms and implementation," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10, 2010.

[51] T. White, *Hadoop: The Definitive Guide: The Definitive Guide*, 2009.

[52] C. Wimmer and T. Würthinger, "Truffle: A self-optimizing runtime system," in *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, ser. SPLASH '12, 2012.

[53] E. Witchel, J. Cates, and K. Asanović, "Mondrian memory protection," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X, 2002.

[54] T. Würthinger, "Extending the graal compiler to optimize libraries (demonstration)," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. SPLASH '11, 2011, p. 41–42.

[55] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, 2008.

[56] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10, 2010.

[57] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "CPI2: CPU performance isolation for shared compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13, 2013.