A down-to-earth look at the cloud host OS

Malte Schwarzkopf Steven Hand
University of Cambridge Computer Laboratory
firstname.lastname@cl.cam.ac.uk

Abstract

Current cloud programming models have opened up new opportunities, but the platforms they run on are still rooted in the legacy of single machine-centric computing. This leads to inefficiency that both costs money and offends scientific sensibilities. In this position paper, we make a passionate and necessarily opinionated argument for a research agenda that challenges fundamental assumptions about operating systems and "cloud" application software. We present a set of ideas for possible directions, and hope to elicit fruitful discussion within the community.

1. Introduction

The IT industry, and, to some degree, systems researchers, have been obsessed by "the cloud" in recent years, and have sung the gospel of infinite scalability, big data, energy efficiency, *-as-a-service, OpEx or CapEx reduction (chose several).

Commercially, the cloud paradigm has built up momentum, as the success of Amazon's cloud offerings and the Hadoop framework shows. Hence it seems clear that in the future, we will increasingly be processing data on remote and distributed systems. However, as good researchers, we have to continuously ask ourselves if we have already found the best answers to the correct fundamental questions, or if we are just taking the cheap way out.

In this position paper, we argue that it is time to revisit some of the fundamental assumptions that much of current "cloud" research is making: does the cloud node host OS really have to look like a traditional UNIX-based system? Is it right to train a generation of programmers to use tightly restricted programming paradigms such as MapReduce? Should we not be putting more effort into re-building our entire systems stack in such a way that it is designed for distributed operation, rather than retro-fitting high-level frameworks onto 30 years of cruft?

In this paper, we hope to shed some light on these questions. We first motivate our questioning of the status quo by highlighting inefficiency issues with the current cloud environments (§2), and next discuss a lowest common denominator in requirements that cloud applications have, representing the minimal feature set we must support (§3). We then debate the implications of this minimal feature set for construction of a "cloud node OS", including the removal of unnecessary legacy constructs that bear no importance in this cloud environment, and may hurt performance (§4). To illustrate the power of this new, slimlined OS structure, we consider a common paradigm in cloud frameworks – task-parallel computation – describe how it maps onto the minimal abstractions, and how it helps to establish a simple, efficient cloud platform, including support for non-determinism and memoisation (§5). Finally, we present the vision of a "cloud compiler", which transforms high-level language code into efficient, statically linked task binaries exploiting the advantages of the platform without programmers having to manually break an application up into tasks (§6). Finally, we relate the concepts presented to existing work exploring similar approaches (§7), and conclude (§8).

2. Why bother?

It is reasonable to ask why we should bother revisiting assumptions about the way things work, and why what we already have in "the cloud" is insufficient – after all, it appears to work well. That is true, and the reluctance to throw legacy infrastructure – or, more euphemistically put, "proven technology" – out of the window is understandable. However, as it stands, software running on the cloud is largely less efficient than it could be. Its performance is subject to high variation, and the fixed overhead in doing anything at all is high. Some of this is due to inherent properties of distributed systems, or the laws of physics, but much of it is human-made – in the sense that we built and designed both the software and the hardware used.

Of course, one can compensate for inefficiency by throwing more resources at the problem. That, however, might motivate us to take a step back and put our optimization hat

HotCDP 2012 – 1st International Workshop on Hot Topics in Cloud Data Processing
April 10, 2012, Bern, Switzerland.

on: with the prevalent charging models in the cloud, albeit coarse, we pay for time, and inefficiency costs time, thus real, hard money.[1]

One might argue that the impact of the inefficiency is negligble compared to the problem size in the "big data" world, especially as constant overheads are considered. While this may sometimes be true, it is not always the case. To give an example, we previously discovered that Hadoop's record I/O interface can severely harm the performance of algorithms executing on many small data, such as k-means clustering, compared to an array-based implementation possible in less restrictive or lower-level frameworks [7] – exemplified, respectively, by our CIEL [8] and MPI (see Table 1).

# 100-dim. vectors	*Hadoop rec. I/O*	CIEL [7] *rec. I/O*	CIEL [7] *array*	*MPI array*
1.6M	263.9s	149.1s	8.1s	4.57s
8M	888.0s	777.2s	38.0s	18.0s
16M	1664.5s	1536.9s	75.7s	34.9s
32M	3,194.8s	3,050s	150s	73.9s

Table 1. k-means performance for different frameworks' implementations; reproduced with permission [7].

While using an simple and fairly general framework like Hadoop MapReduce might be easier for the programmer, and thus save expensive engineering time, the key question is whether we can achieve better performance while maintaining the engineering simplicity. Indeed, others have followed the same tracks: Spark [11] uses the abstraction of resilient distributed datasets (RDDs) as a flexible alternative, while integrating into the Scala language. Similarly, the OpenStack project[2] aims to provide a general-purpose cloud stack. We believe, however that both of these approaches suffer from inherent inefficiency due to deep layering and their high-level, user-space nature. Zaharia *et al.* have made the call for a data centre operating system [12], but again, they explicitly position their vision and some of its implementation at a higher level than the host OS [3, 11]. The closest approach to that taken in this paper is that of our previously described Mirage ecosystem [6], which compiles OCaml applications to virtualized exokernels. Mirage derives its simplicity from restricting itself to a virtualized environment. Virtualization, though, always has an overhead, and it is instructive to note that large cloud data centre operators such as Google, Facebook and Microsoft do not use it.

In this paper, we advocate a more radical approach: to break with legacy assumptions on operating system and application structure. Instead, we advocate providing a minimal, high performance OS interface for data access and program execution, and including framework-specific abstractions as libraries in the applications. This deliberately low-level approach aims to avoid the risk of building ever more complicated and deeply layered systems, eventually being unable to see the bottom from the top and vice versa. To some extent, this is already the case: who can, these days, when debugging a job running on "the cloud", tell the exact nature of a performance glitch, and at which layer it originated?

In the following, we take a step back, and first consider the essential functionality required by cloud applications, and thus the fundamental set of requirements our proposed approach must satisfy.

3. What do we really need?

Cloud workloads come in all shapes and sizes, but they broadly fall into two categories: *batch data processing*, which is usually not especially time-critical, but must be scalable to large data sets, and *serving*, which must both deliver high-performance, low latency responses and scale according to demand, while performing the appropriate back-end actions. Some workloads also live in a gray zone between these: semi-time-critical data processing in response to a request or incremental processing, for example.

A shared characteristic of all of these is sensitivity towards contention: depending on whether workloads are CPU-bound, memory-bound or I/O-bound, sharing these resources with other computations will degrade performance. For this reason, performance isolation is important. Isolation is also crucial for security reasons, since different jobs are not necessarily mututally trusted, especially when multiple users are sharing cloud infrastructure. Historically, multiplexing resources and isolating processes from each other has been the responsibility of the operating system, but it is nowadays also increasingly enforced at hypervisor level.

Similarly, it is also important to be able to claim new resources as workloads grow, and to be able to discipline rogue computations if necessary. This execution control has also traditionally, in a multi-process, time-shared OS, been enforced by the kernel, using pre-emption and access controls. In current cloud frameworks, the responsibility is commonly split between the host OS – which often has a myopic view of the host machine, due to virtualization or containerization – and a global, framework-specific "master" component.

Computations, also need some means of accessing data. In traditional OSes, temporary and persistent storage are usually separately managed and accessed via independent APIs. In the cloud, data is often – even temporarily – stored distributedly for performance and durability. Access to it is largely managed by user-space distributed file systems, which rely on host OS mechanisms to access the actual data on the machine holding it.

These observations already hint at a minimal set of features that we must support; in the following, we additionally consider specific requirements for the two workload categories.

[1] The HPC community, where supercomputer time is precious and expensive, has been aware of this for a long time.

[2] `http://openstack.org`

3.1 Batch data processing

Batch processing workloads – such as a MapReduce job – are often implemented in a task-parallel way: many individual instances of the same code each process a part of the data. The requirements for doing so are mostly straightforward: the task consists of a piece of *code* (compiled or not) that sequentially processes some *input data*, producing some *output data*. This is remarkably similar to the most primitive notion of a "program": a program counter is pointed to the first instruction in a sequence and runs along it, mutating some state on the way.

While running, a task consumes resources – disk head time, memory, CPU time – although the precise amounts can be dynamic, which makes advance reservation of the exact resources required hard. Unlike serving jobs or interactive desktop applications, however, tasks in batch jobs rarely have idle periods in their execution, assuming they only start when all inputs are available.

3.2 Serving workloads

Serving remote requests (HTTP traffic, RPCs) is a common "cloud-side" workload in today's networked and web applications, and also a prime example of a workload driven by fluctuating demand, requiring high scalability. Frequently, the work is I/O-bound, although this does not have to be the case – deep backend pipelines (in-memory caches, key-value stores, databases, content generation, etc.) may add further computation to the simple request-response scheme. Scalability in these workloads is usually achieved by running additional instances of the serving code, and multiplexing requests to them, balancing the load.

Contemporary serving workloads are generally constructed by composing traditional long-running processes, which communicate using standard communication channels such as TCP.

4. Dumbifying the OS

Cloud platforms today utilize customized versions of existing mainstream operating systems as host OSes – usually based on a UNIXoid kernel and the GNU software stack, or some version of Windows. Frequently, these operating system images run inside VMs in a virtualized infrastructure, or in thin isolation containers in a non-virtualized one.

However, modern operating systems are highly complex. They consist of many layers of multiplexing, and what is seen by an application program as a processor that sequentially executes instructions in the program code and accesses memory is really only an illusion. Not only is the hardware seen often virtual, the execution is also interrupted by context switches due to scheduler pre-emptions, interrupt handling or virtual machine monitor decisions. Memory accesses may be translated, sometimes multiple times, and syscalls may cause unexpected actions to take place. On top of this, there may be further layers of indirection, such as a language runtime and user-space threads.

If we recapitulate the essential needs identified in the previous section – whether for batch job or a serving job – they do not mandate the complexity of a multi-process, time-shared operating system with dynamically linked libraries, drivers and kernel modules. Really, the operating system is only required to provide four kinds of functionality: *execution control*, *resource management*, *isolation* and *data access*. It should be feasible to build a simple "cloud OS" supporting these, without requiring a deep stack (like existing high-level platforms), or virtualized operation (like Mirage). In fact, the more of the mechanisms for data access and execution control currently baked into high-level frameworks we can standardize and push into a common OS layer, the higher we should expect our application efficiency and the environment's comprehensibility to be.

4.1 Execution control

Such a simplified multi-process OS, which provides nothing apart from the above four features to non-preemptively scheduled processes without communication primitives, is a model like that adopted by some embedded operating systems, such as TinyOS [5]. Unlike most embedded systems, however, we can make the assumption that the OS is running on a multi-core host machine, and thus need not to worry about interrupting processes to handle events, as we can reserve one or more cores for OS event processing. For the same reason, livelock is not a problem: if the OS retains control of one or more cores, it can always kill processes on others, meaning that a non-preemptive scheduler suffices. Hence, a process, once scheduled, will not be descheduled or its execution interrupted, unless it explicitly yields or blocks, and effectively has dedicated access to a core. We expect that the lack of context switches and interrupts, combined with exclusive access to caches and memory local to the core will result in a noticeable improvement in execution performance for application code.

Binaries in this environment should come with all their necessary libraries statically linked, such that, assuming a matching machine architecture, they can execute without any external dependencies apart from the narrow OS interface, similar to a library OS approach [9]. This may lead to very large binaries, but we believe that contemporary and future data centre networks are not likely to be troubled by a 100 MB executable, and that the binaries will remain tiny compared to many of the input data sets processed. The great advantage that this restriction provides is that we can now treat binaries as entirely self-contained: they have no dependencies on anything other than their inputs.

4.2 Resource management and isolation

In this model, as with the classical OS model, performance isolation and fair sharing of comparatively few shared I/O resources are key challenges. The OS might, however, make

this easier for itself by restricting memory access and I/O such that it can only occur to and from kernel-provided buffers. In practice, this means that applications must explicitly request and commit I/O buffers via a kernel API, including disk accesses and heap allocation. This limitation means that the OS can throttle any process to a fair share of I/O resources by not supplying any more buffers to it while it waits for recently commited ones to be drained.

Since executing processes receive exclusive access to a CPU, the OS can also ensure that buffers granted to them are, as far as possible, optimally chosen – for example, heap memory can be granted on the closest NUMA node.

4.3 Data access

The same new kernel API should also provide access to persistent data, allowing the OS to transparently provide processes with local or remote data by supplying appropriate buffers.

Specifically, we advocate that there should exist a syscall interface to allow a process to request a particular, globally unique named *data object*, which will then be resolved locally or remotely, and once accessible, mapped into a buffer that is granted to the process. This is effectively replacing the common distributed file system abstraction with a kernel-provided lookup functionality that may refer the request to remote machines or an index server.

Since binaries are also data objects in this global namespace, executing a process amounts to asking the kernel to load the data object into memory and start running it, assuming the data object and sufficient local resources are available. Alternatively, the kernel may refer the execution request to a remote machine, instantiating the process there. This is possible because binaries have no dependencies apart from their inputs (see §4.1), but, in combination with the non-preemptive scheduling, also necessitates that any IPC must go through the kernel buffer API.

4.4 Summary

In the vision outlined above, the OS is degraded to a local execution and resource multiplexing coordinator.

Of course, dropping support for applications requiring a shell, standard binaries, a filesystem, multi-threading, locking, concurrency primitives, pre-emption, or more than a rather small set of basic drivers may seem extreme. As we show in the following, though, the basic primitives described are completely sufficient for task-parallel computation, and in fact offer particularly appealing benefits to this widely used cloud application paradigm.

5. It's task-parallel, Jim!

Task-parallel computation has emerged as a very successful design pattern for distributed computing, and many popular cloud data processing systems embrace it. Unlike in parallel programming using threads and shared-memory, the code for each individual task is entirely sequential, and all synchronization is implicit in the programming model, taking place when tasks access inputs and produce outputs.

We see the simplicity and tasks' clearly defined interaction behaviour as an ideal match for a simplified "cloud OS". Our particular model considered here takes significant inspiration from our prior work on univeral data-flow programming using CIEL [8], but similar models are imaginable.

Just like a binary in a traditional OS is an executable file, a task can be treated as just an executable data object from the OS perspective (see §4.3). Other data objects (which may themselves be tasks) constitute the inputs and outputs of a task, and buffers for accessing them can be requested from the kernel. In order to be able to locate data objects in the global namespace, they must be named by UUIDs. If these names are deterministically generated from components describing their generating task, freshness and provenance, running a task-parallel computation becomes easy. For example, the name of an output might be composed from the generating task's UUID, the output sequence number, a version number and a set of input UUIDs used to generate the output,[3] i.e.

$$\textit{output UUID} = \{\textit{task UUID}, \textit{seq. no.}, \textit{ver.}, [\textit{input UUIDs}]\}.$$

In this scheme, names are capabilities, such that in order to access a data object, a task needs to know either its name, or the components necessary to generate the name. Since the OS does not supply a file system or any data object listing primitive, tasks must explicitly communicate the names between them.[4] Second, the scheme can support both relaxed and strong consistency on data: when requesting a data object, the kernel can either be instructed to perform an expensive check on the version number (e.g. by querying a distributed meta-data index), or omit this check and use a potentially stale version of the data object (cached locally or fetched from a remote). The version number could also be set to a special value indicating a non-deterministically generated data object, which must be re-generated every time on every access by executing its generating task.

If data objects are cached locally when used (executed or accessed), frequently executed task binaries (e.g. mapper or reducer tasks) are likely to be cached locally once one instance has started running. Similarly, it is possible to reuse deterministically generated data objects cached from previous computations, such as intermediate results from a MapReduce job.

6. Compiling to the cloud

Assuming the simple cloud OS model works well with task-parallel computations, as dominant in today's batch processing frameworks, two key questions remain: first, how the

[3] Usually, but not necessarily, the inputs to the generating task.

[4] Although it is of course possible for an application to build its own directory or index abstraction on top of the data objects.

task binaries are generated, and second, how other types of applications can be supported in an OS providing little in the way of traditional APIs.

Ideally, a programmer would write a cloud application in a higher-level language, and a compiler would automatically translate the application into a binary suitable for running on the simple cloud OS. In practice, the best way to achieve this might be to translate the application into a set of independent tasks, given that we have shown task-parallel computations to map well onto a simple OS API. This is a very hard problem if mutable global state exists, as its presence makes it tricky to precisely identify what the inputs and outputs of a task are. Without mutable global state, however, a classic data-flow analysis might be able to give us an idea how data is passed between different parts of an application. This approach differs from existing similar approaches – such as CIEL's Skywriting language [8], or Cilk-NOW's `spawn` primitive [1] – in that task boundaries are inferred, rather than explicitly specified.

Indeed, it might be possible to express even serving workloads using the task-parallel paradigm, if non-determinism is supported. Ultimately, as a blue-sky vision, it might be conceivable for a compiler to be written in a task-oriented way, such that it generates task binaries from source code stored in data objects, all running over the cloud infrastructure.

Implementing such a compilation infrastructure is a large and ambitious project, but we believe that recent advances in modular compiler design, such as the LLVM project [4], make it much easier to build the necessary modules, while still generating very efficient machine code.

7. Nothing new under the sun?

As with many positions in systems research, our ideas and concepts are not entirely novel, and variants have been proposed before.

In 2008, Thibault *et al.* considered the potential of running HPC applications on top of Xen by linking the application to a small "stub domain" OS [10]. More recently, Zaharia *et al.* made the case for a coherent data centre OS instead of ad-hoc solutions [12], although their position is more pragmatic, and explicitly not concerned with the host OS in cloud data centres or questions of efficiency, but with cluster-wide coordination and framework interoperability issues. They do, however, identify the similar roles for a data centre OS as we have discussed as properties of a host OS in this paper: fair resource sharing, performance isolation and data sharing between jobs. With the Mesos cluster manager, which manages different applications' resource allocations and enables different frameworks to share a cluster without detrimental effects on performance [3], the same authors have delivered a part of their vision.

We have already discussed the Mirage OS, coming from the same motivation we advocate, but designed for a virtualized environment and less oriented towards task-parallel computation [6]. The CACHE kernel [2] supports memory-based messaging for IPC, and the Drawbridge OS is the a recent library OS that bundles libraries with applications in a similar way to our statically linked binaries [9].

8. Let's find out!

We do not know if any aspect of the approach we have proposed in this paper will actually exhibit the envisioned benefits over the state of the art – but we do firmly believe that we should try to find out. While we do not plan to implement all of the ideas discussed in this paper in the short term, we are planning to investigate some of the concepts we proposed. We encourage the community to think outside the box, and to be willing to challenge widely accepted "truths" about OS design when thinking about the cloud.

References

[1] BLUMOFE, R., AND LISIECKI, P. Adaptive and reliable parallel computing on networks of workstations. In *Proceedings of USENIX ATC* (1997), p. 10.

[2] CHERITON, D. R., AND DUDA, K. J. A caching model of operating system kernel functionality. In *Proceedings of OSDI* (1994).

[3] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of NSDI* (2011).

[4] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of CGO* (2004), no. c, pp. 75–86.

[5] LEVIS, P., MADDEN, S., POLASTRE, J., SZEWCZYK, R., WHITEHOUSE, K., ET AL. TinyOS: An operating system for sensor networks. *Ambient intelligence 35* (2005).

[6] MADHAVAPEDDY, A., MORTIER, R., AND SOHAN, R. Turning down the LAMP: software specialisation for the cloud. In *Proceedings of HotCloud* (2010).

[7] MURRAY, D. G. *A distributed execution engine supporting data-dependent control flow*. PhD thesis, University of Cambridge, 2011.

[8] MURRAY, D. G., SCHWARZKOPF, M., SMOWTON, C., SMITH, S., MADHAVAPEDDY, A., AND HAND, S. CIEL: a universal execution engine for distributed data-flow computing. In *Proceedings of NSDI* (2011).

[9] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the library OS from the top down. *Proceedings of ASPLOS* (2011), 291.

[10] THIBAULT, S., AND DEEGAN, T. Improving performance by embedding HPC applications in lightweight Xen domains. In *Proceedings of HPCVirt* (2008), pp. 9–15.

[11] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *Proceedings of HotCloud* (2010), p. 10.

[12] ZAHARIA, M., HINDMAN, B., KONWINSKI, A., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. The Datacenter Needs an Operating System. In *Proceedings of HotCloud* (2011).