Data centers are microkernels done accidentally: lessons for building a million-core distributed OS

Malte Schwarzkopf [†] Matthew P. Grosvenor[†] David Chisnall[†] Ionel Gog[†] Natacha Crooks[†]
Robert N. M. Watson[†] Anil Madhavapeddy[†] Frank Bellosa[‡†] Steven Hand[†]
[†]*University of Cambridge Computer Laboratory* [‡]*Karlsruhe Institute of Technology*

Abstract

Data centers have accidentally fulfilled the microkernel vision: they move critical operating system components into user-space middleware. Consequentially, it is time to move to a different view of the OS in the data center—treating the node OS merely as a local enforcement agent of global decisions made by user-space systems software. Based on this observation, we present a novel distributed OS design for treating the data center as a "warehouse-scale computer", and explain why this has attractive benefits for common distributed applications.

1 Introduction

Large-scale data centers have become indispensable over the last decade. While the 1990s still saw single mainframes power many large-scale operations, economies of scale and increased per-processor performance have made "warehouse-scale computers" (WSCs) from commodity parts a popular choice in data center design. A data center with 20,000 servers of 48 CPU cores each—in other words, a million cores—is a plausible configuration for a WSC today [44]. However, size is not the only distinguishing factor: WSCs, unlike traditional data centers, are operated by a single organization, use homogeneous hardware and "much of the application, middleware, and system software is built in-house" [5, 25]. Operating system design, however, has failed to keep up with this trend: shifting from time-shared mainframes to general-purpose desktop computing in the 1990s, it was taken aback by the reversal of direction. Consequently, modern WSCs run many instances of a commodity microcomputer OS linked together in an ad-hoc fashion by user-space software [50].

At the same time, research into microkernel OS construction has been an obsession dear to systems research, mostly due to its architectural elegance [21]: touting advantages such as a communication-centric design by breaking the OS into independent user-space components [2, 37], fast IPC [22, 34], extensibility via the same mechanisms [9, 46], exposure of abstraction-less low-level interfaces to user-space programs [17, 32], and small trusted computing base and formal verifiability [30, 47], microkernels were always the cool kids' LEGO. Unfortunately, they fail to be deployed in large-scale systems, with some arguing that they are inferior to the pragmatic sandcastle of virtual machine monitors in practice [21].

Surprisingly though, microkernels conceptually have come full circle in today's WSCs: while microkernels originally adopted a communication-centric design aiming to reduce kernel complexity to allow flexible implementation of systems-level functionality in user-space, WSCs have adopted a similar design for expedience and convenience. Typical WSC infrastructure middleware is a distributed system in user-space: file systems [18, 39, 45], structured storage [4, 13, 14], data processing engines [15, 28], synchronization and locking services [11, 27], schedulers [29, 54] implement functionality duplicating traditional OS services. In the following, we describe how this surprising coincidence came about (§2), discuss the implications for operating systems on WSCs (§3), and present a clean-slate distributed OS design for this setting (§4-5). Finally, we discuss its advantages and limitations, and related existing work (§6).

2 The accidental microkernel

While actual microkernels have enjoyed some popularity in the embedded systems community due to their small resource footprint [23], they were never (to our knowledge) considered for data centers. Why is it that, nonetheless, WSC operators have adopted a *de facto* microkernel way of running their systems?

We believe that there are two main reasons: pragmatically, the need for *rapid evolution* and, architecturally, the fact that WSC applications are *distributed by default* to attain scalability and resilience unavailable on a single machine.

Rapid evolution. WSC infrastructure—due to a business need for systems to scale to millions of users—has had to evolve rapidly over the last decade. For example, Google has deployed five major storage systems over the last eight years: GFS [18] and its successor Colossus, MegaStore [4], BigTable [13] and Spanner [14]. This pace of innovation is well beyond even the most optimistic development cycles for OS features. In fact, it may only be feasible as a result of an active choice to forgo OS integration inspite of its potential performance benefits: user-space software development and debugging is easier, cheaper and can make use of a breadth of existing libraries, unlike an in-kernel implementation.

Distributed operation. Large scale computation is most cost-effective when fault tolerance features are implemented in software [6]. WSCs are constructed from inexpensive commodity hardware and simple Ethernet interconnects. Fault tolerant software running on such a platform necessarily takes the form of a message-based distributed system. Commodity OSes do not embrace distribution, as popular variants have their roots in the age of time-shared SMP machines, and even recent OS research efforts stop at the chassis boundary [7]. Fully distributed OSes, on the other hand, have fallen out of fashion after attempts to join networks of workstations into a single computing environment [31, 37, 40] failed to take off.

Custom distributed OSes are used on some high-performance computing (HPC) systems [20], but these are tailored to very specific problem domains orthogonal to the WSC use case. Specifically, such approaches focus on homogenous, compute-heavy, usually numerical workloads. WSC workloads are far more diverse: they include latency-sensitive user-facing operations as well as long-running data processing tasks. As HPC OSes emphasize reducing jitter and OS impact on synchronized processes, they do not support co-scheduling heterogeneous workloads [3]. Hence, WSC operators were left with no choice but to use commodity OSes. These, however, have traditionally shied away from moving essential OS components, such as file systems or schedulers, to user-space in order to efficiently support interactive desktop computing. Since distributed versions of such components are required, WSC operations reimplement them in user-space. They only use the kernel to effect network and block storage I/O, to bootstrap the node and to partition it into resource allocations, as well as to maintain isolation between the systems running in user-space. Data is managed and policy decisions are made by distributed infrastructure in user-space. This exactly fulfills the original microkernel vision: the kernel only provides minimal fundamentals, while the majority of OS functionality is implemented in user-space.

3 Implications

The incentives of commodity, general-purpose OSes and of WSCs are not well aligned: while the former must support a variety of use cases in general-purpose computing and are reluctant to specialize, WSCs benefit from specialization. Indeed, WSC operators are known to optimize at microscopic levels in order to harness potential for savings at scale—both in hardware [1] and in software [36], and at significant expense.

It is thus conceivable—and indeed known practice [50]—for WSC operators to heavily modify the commodity OS they run. While pragmatic in the short term, and admittedly beneficial in running a platform compatible with the "outside world", we believe that this approach misses a number of crucial opportunities, while being hampered by redundant baggage.

Awkward fit of domain abstractions. WSCs run a very particular breed of applications, coming with their own abstractions from the distributed systems world: for example, task-parallel data-flow programming is common [38], and message-based algorithms for replication and consensus abound [12]. Mapping these abstractions onto classic OS primitives such as shared-memory synchronization or sockets is often far from straightforward. Furthermore, the existing abstractions for remote communication are designed for a pessimal general-purpose case, rather than the tight, low latency interconnect of a WSC: for example, the socket interface necessitates a data copy, based on the assumption that copying memory is a much cheaper operation than network communication. An OS designed to fit the distributed computing bill of WSCs' particular niche could do better here—improving performance and reducing both complexity and development effort.

Static local/remote boundary. The abstractions used for effecting local and remote actions (most importantly, inter-process communication and I/O) are separate in modern commodity OSes: replacing a shared memory connection with sockets (or vice versa) is a significant refactoring effort. However, as the number of cores in a machine scales and WSCs grow in size, the optimal decomposition into local and remote actions is non-obvious, and may change frequently. Unifying these abstractions is an opportunity that WSC software could greatly benefit from, but which is unlikely to be met with much enthusiasm in commodity OSes.

Unused code and complexity. Running a commodity OS, such as Linux, but using it in a microkernel fashion as described above necessarily ends up utilizing only a fraction of the considerable code base. Is that necessarily an issue? Not *per se*, but unused code introduces complexity, which makes the system harder to understand

and evolve. Complete tracing of a distributed job running on a WSC is a hard problem [10, 41], in part due to additional complexity being introduced by an already complex OS underneath the distributed infrastructure.

Cost of maintaining specializations. Despite the benefits that being "hooked in" with a commodity OS's development process brings, it is also a burden: customizations must be adapted to suit upstream changes, and this constitutes a significant effort. Google, for example, in 2009 were maintaining 1,280 proprietary patches adding ~300,000 lines to the Linux kernel [50]. Due to the incentive misalignment, upstream changes that benefit e.g. interactive desktop computing are often detrimental to the performance of WSC applications: Google found the CFS scheduler in the 2.6 kernel to be unhelpful, and forward-ported an old 2.4-era scheduler [50].

4 Requirements

What if, instead of using a commodity OS, a WSC operator invested the same resources into development of a custom, special-purpose OS? This approach worked for physical infrastructure[1] and we believe there is value in considering what a clean-slate OS redesign for a WSC might look like. But what are the precise requirements that such an OS must meet?

Infrastructure "middleware". Consider the typical WSC use case: infrastructure systems—such as MapReduce [15], BigTable [13] or a cluster scheduler like Quincy [29]—serve higher-level applications, typically called *jobs*, which are composed of many parallel *tasks*. The infrastructure systems run as long-running tasks and are akin to privileged user-space servers, such as an external pager, in a traditional microkernel: they are trusted to access all data in the system, and their correct operation is necessary for the system to work and to make progress. Typically, they are implemented by the most expert programmers, and changes are vetted thoroughly.

User jobs. Jobs running on top of this infrastructure software are data analytics work for business logic or serving work for user-facing front ends. Usually, they will link against a client library that facilitates communication with the infrastructure middleware via system-specific APIs. Their tasks are typically scheduled centrally, based on multi-dimensional resource requirements [19]. The data they deal with—for example, users' emails, map tile images for a mapping application, pages in a search index, or nodes and edges in a social graph—are numerous, and replicated for availability and resilience.

I/O, partitioning and fault-tolerance. The OS-level requirements of such data-intensive workloads are first and foremost high performance disk and network I/O (since they are commonly I/O-bound), but also the support for a variety of consistency levels in I/O, currently usually implemented in the infrastructure systems. Since the WSC is a shared system, it is recursively partitioned into resource shares: between organizational units, individual users, jobs and eventually tasks, which must also be accounted for [16]. The composition of many nodes built from commodity hardware also necessitates strong tolerance towards individual components' failures, which occur frequently at the scale of a WSC, and fast, reliable messaging between the nodes.

[1]E.g., in OpenCompute: `http://opencompute.org`

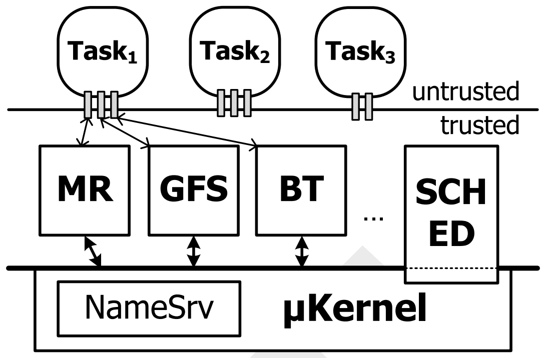

Figure 1: Three untrusted tasks using MapReduce, GFS, BigTable and a cluster scheduler running atop DIOS.

5 A distributed WSC OS design

We have argued in §3 that the *status quo* of commodity OSes being used as *de facto* microkernels is unsatisfactory. We now state the principles for our DIOS design, and sketch a possible syscall API (Table 1).

Fundamentally, an operating system is concerned with *(i)* locating targets for I/O, *(ii)* allocating required resources and scheduling computation, and *(iii)* effecting I/O. This is reflected in the DIOS design, although we allow all higher-level policy decisions to be made by user-space software. The primitives supplied by the OS kernel, however, embrace distributed operation.

While functionally equivalent to a microkernel (Fig. 1), DIOS, unlike classic microkernels, implements a network stack and a block storage driver, as well as a distributed name service, in the kernel. Like a classic microkernel, DIOS has privileged *trusted* and unprivileged *untrusted* user-space applications.

Global names and local references. Naming is a key problem in distributed systems (as well as in classic OSes), and we support it using two primitives in DIOS: every object (an unstructured set of bytes, a running task, or a device) is named by a globally unique and valid identifier, its *name* ($\mathcal{N}$). Global name resolution to reachable objects is only available to infrastructure tasks running

Syscall	Function
$\langle\mathcal{P}$,`size`$\rangle$ `begin_read(`$\mathcal{R}_o$`)`	Informs kernel of read on object referenced by $\mathcal{R}_o$; returns read buffer pointer $\mathcal{P}$.
`bool end_read(`$\mathcal{R}_o$`)`	Checks consistency of completed read on $\mathcal{R}_o$; returns validity indication.
$\mathcal{P}$ `begin_write(`$\mathcal{R}_o$`, size)`	Informs kernel of write of `size` on $\mathcal{R}_o$ or upgrades to write; returns pointer $\mathcal{P}$.
`bool end_write(`$\mathcal{R}_o$`)`	Checks consistency of completed write on $\mathcal{R}_o$; returns validity indication.
$\mathcal{R}_t$ `run(`$\mathcal{R}_o$`)`	†Runs object referenced by $\mathcal{R}_o$; returns reference to running task.
$\langle\mathcal{R}_o^0$`, ..., `$\mathcal{R}_o^n\rangle$ `lookup(`$\mathcal{N}$`)`	†Attempts to find all reachable instances of the object named by $\mathcal{N}$.
`bool pause(`$\mathcal{R}_t$`)`	Pauses the running task referenced by $\mathcal{R}_t$; returns success indication.
$\langle\mathcal{N}$`, `$\mathcal{R}_o\rangle$ `create()`	Creates a new object and returns a name $\mathcal{N}$ for and reference $\mathcal{R}_o$ to it.
`bool copy(`$\langle\mathcal{R}_o^0$`, ..., `$\mathcal{R}_o^n\rangle$`, `$\mathcal{R}_t$`)`	Copies references $\mathcal{R}_o^i$ into the environment of task t referenced by $\mathcal{R}_t$.
`bool delete(`$\mathcal{R}_o$`)`	Removes the object instance referenced by $\mathcal{R}_o$.
$\mathcal{R}_o^i$ `select(`$\langle\mathcal{R}_o^0$`, ..., `$\mathcal{R}_o^n\rangle$`)`	Returns the first readable reference $\mathcal{R}_o^i$ out of the n references in $\langle\mathcal{R}_o^0,\ldots,\mathcal{R}_o^n\rangle$.

Table 1: DIOS *syscall API; all syscalls additionally take a set of flags $\mathcal{F}$. † trusted infrastructure tasks only.*

in trusted mode. In order to describe objects throughout the system, locally scoped references ($\mathcal{R}$) are used. As objects may be replicated at various points in the WSC, multiple references can refer to different instances of the same object. The node kernel mints references as part of name lookup or name creation. The former interrogates the global naming service, while the latter produces a newly generated, unshared name and reference. The reference is stored in a task-specific reference table and carries context information that only makes sense in the local environment. For example, it may hold information on whether the target is local or remote (exposing its access cost), specify the the maximum write size (buffer size), or specify information on whether an object instance is in persistent memory. References can be explicitly shared between tasks by invoking the `copy` operation, which will store them in the target task's reference table and adapt them to its environment.

Syscall API. Table 1 shows a straw-man syscall API for the DIOS design, which we refer to for illustration in the following. Note that all syscalls take a set of flags, $\mathcal{F}$, in addition to the parameters specified. These flags specify expectations on behavior: for example, flags to the I/O syscalls determine whether the object should be accessed under a weak or strong consistency regime, while flags to `copy` can be used to restrict (but not widen) access to a reference. In addition, all syscalls are blocking.

Tasks are provided with an initial set of references at startup. These may be references to data (e.g. arguments), resources such as an OS timer, IPC endpoints, or IRQ registration points. Asynchronous abstractions are implemented using `select` and a timer reference.

Only trusted tasks can invoke `run`, since it has the ability to start a task anywhere in the WSC (resources permitting). In practice, it is largely a single trusted task that makes use of the task running facility: the user-space scheduler. Untrusted jobs may spawn additional tasks by communicating with this scheduler using its API; it is the scheduler who then decides on the resources and invokes `run`. The scheduler is bootstrapped by checking at node bootup if any scheduler task is running in the WSC; if not, the node starts a scheduler task.

Recursive abstractions. In operating systems, it is often handy to allow the same mechanisms to be used at different scales. This is particularly pertinent in a WSC, where dynamic partitioning of resources into administrative, job and task domains is necessary. Indeed, such ideas are commonly found in mainframe literature—for example, the Cambridge CAP Computer [51] had a recursive process abstraction, Popek and Goldberg [42] defined the requirements for recursive virtualization in 1974, and in the VM/370 architecture, levels of nested virtual machines up to five deep were reported [48].

A recent resurgence of interest in library OSes [35, 43] testifies to the timeliness of this observation: Drawbridge, for example, defines a deliberately narrow ABI of 36 calls to ease resource virtualization towards the library OS [43]. Similarly, by providing a simple, easily virtualizable, set of kernel interfaces, we make it easy for users of the system to easily create partitions of an arbitrary granularity. Interposing the trusted syscalls (`lookup`, `run`) using a dynamically-linked library that forwards them to an IPC endpoint allows us to nest DIOS instances arbitrarily, supporting this goal.

Unified I/O and communication. DIOS is explicitly designed to use zero-copy unified primitives for communication and persistent I/O. Typically, the application obtains a reference by either creating an object, by looking up a name, or by being supplied via `copy`. If the object is another task, I/O to the reference results in IPC (local or over the WSC interconnect); if it is stored data (in a local shared-memory area, remote memory, or on disk), I/O results in access to said data with the specified consistency regime. Unlike previous distributed OS designs [24], in which every memory access could potentially refer to a remote location, and thus have a high cost and latency, in DIOS, explicit primitives need only be invoked for access to shared data (references that have been passed by `copy`) or other objects, though—an application may use its own virtual memory in whatever way it pleases.

As we trust the infrastructure tasks, and because kernel crossings are a major source of overhead on I/O syscalls which get invoked frequently in typical data-intensive applications,[2] such tasks may perform I/O to their references entirely in userspace.

The previously described concept of references addresses the typical problem of distributed microkernels with unified I/O primitives: that in hiding whether communication is local or remote, they make it impossible to reason about the performance or reliability of seemingly simple code. With references, we make it easy to write code that can determine whether an action is remote or local and can recover in the non-local case.

6 Discussion

Security. Due to the sensitive nature of data processed on WSCs, it is paramount that DIOS is able to prevent accidental data disclosure to the furthest possible extent. The security model we described is two-fold: access to name resolution is limited to trusted infrastructure tasks, and references can be passed and restricted like capabilities. This model is conceptually equivalent to the one realized by Capsicum [49] for BSD, and sufficiently expressive to allow user-space infrastructure to enforce its own authentication and access control abstractions for untrusted jobs. Trusting the infrastructure software is not a problem—it is the *status quo* in today's WSCs.

Benefits. The distributed OS design described has several benefits over current solutions: as user-space middleware dictates OS policy (e.g. in scheduling), conflicting goals between distributed infrastructure and node OS can be avoided. A key example of this is the lack of interaction between local OS schedulers and cluster schedulers in current setups—bridging this gap might enable more efficient operation. Furthermore, due to its simplicity and small size, DIOS is likely to be easy to maintain, extend and customize. The reference abstraction is handy in distributed systems construction, and as a side-effect provides OS-level information flow data useful for tracing, debugging and provable compliance.

Limitations. DIOS does not support userspace multi-threading, and instead uses task based parallelism with explicit memory sharing. This is intentional: it is central to our design philosophy that no two control flows communicate without explicit use of the I/O primitives. This facilitates flexible task decomposition into local and remote parts, benefiting migration and simplifying debugging and traceability. Furthermore, interactive desktop computing paradigms, such as interactive shells, are a poor fit for DIOS. While feasible under the API, these would be cumbersome and of little practical value in a WSC.

Messaging performance. Remote operations in DIOS frequently necessitate messages to be sent to another node. As with traditional distributed microkernels [33], extending messaging beyond a single machine requires strong latency and reliability guarantees in order to be efficient. We thus make the assumption that message delivery inside a WSC interconnect is fast—which is reasonable in the tightly coupled environment of a data center—and assume that, while we cannot guarantee reliability, delivery failures can be detected. This can either be achieved by aggressive timeouts (trading additional failures for performance), or by support in the interconnect.[3]

Rollout. DIOS can be deployed incrementally: while some machines in the WSC continue to run a commodity OS, DIOS is deployed to others. Infrastructure systems can then migrate separately—since their API with the user jobs does not need to change, the rollout can be transparent to most programmers. Interoperability between systems at different stages of being ported to DIOS can be provided by a shim library implementing the DIOS API deployed on top of a commodity OS.

Related work. The utility of distributed systems primitives for building scalable operating systems for multi-core machines has been noted [8, 26], and led to the conception of the OS as communicating multi-kernels [7]. This work is complementary to ours: DIOS could well be realized using per-core multi-kernel instances. Indeed, others have previously noted that "the data center needs an operating system" [53], but stopped short of replacing the commodity node OS. Most directly related to our ideas are Amoeba [37] and Hydra [33], both distributed microkernel OSes. However, our design is not as purist: to specialize DIOS to the WSC environment, we include network and disk I/O in the kernel, and do not mandate strict use of capabilities (references) inside the kernel.

Micro-kernels are the original sin of OS research, and we have been tempted too: we are currently working on D¢OS, an implementation of the DIOS concept.

[2] We found that 35-50% of Linux syscalls made by typical WSC applications such as Hadoop MapReduce, Redis or ZooKeeper are `read` or `write`.

[3] Switch support is conceivable [52]; however, in related but separate work, we are building a novel switchless, bufferless data center interconnect with this property.

References

[1] Abts, Dennis and Marty, Michael R. and Wells, Philip M. and Klausler, Peter and Liu, Hong. Energy proportional datacenter networks. In *Proceedings of ISCA* (2010), pp. 338–347.

[2] Accetta, M., Baron, R., Bolosky, W., and Golub, D. Mach: A new kernel foundation for UNIX development. In *Proceedings of Summer USENIX Conference* (1986).

[3] Adiga, N., Almasi, G., Almasi, G., Aridor, Y., Barik, R., Beece, D., Bellofatto, R., et al. An overview of the bluegene/l supercomputer. In *Supercomputing, ACM/IEEE 2002 Conference* (nov. 2002), p. 60.

[4] Baker, J., Bond, C., Corbett, J., Furman, J., Khorlin, A., Larson, J., Léon, J., Li, Y., Lloyd, A., and Yushprakh, V. Megastore: providing scalable, highly available storage for interactive services. In *Proceedings of CIDR 2011* (2011).

[5] Barroso, L. A. Warehouse-scale computing: Entering the teenage decade. *SIGARCH Comput. Archit. News 39*, 3 (June 2011).

[6] Barroso, L.A. and Dean, J. and Holzle, U. Web search for a planet: The Google cluster architecture. *Micro, IEEE 23*, 2 (march-april 2003), 22–28.

[7] Baumann, A., Barham, P., Dagand, P., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of SOSP* (2009), pp. 29–44.

[8] Baumann, A., Peter, S., Schüpbach, A., Singhania, A., Roscoe, T., Barham, P., and Isaacs, R. Your computer is already a distributed system. Why isn't your OS? In *Proceedings of HotOS* (2009), pp. 12–16.

[9] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S. Extensibility safety and performance in the SPIN operating system. In *Proceedings of SOSP* (1995).

[10] Bligh, Matt and Desnoyers, Mathieu and Schultz, Rebecca. Linux Kernel Debugging on Google-sized Clusters. In *Proceedings of the Ottawa Linux Symposium* (2007).

[11] Burrows, M. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of OSDI* (2006), pp. 335–350.

[12] Chandra, T. and Griesemer, R. and Redstone, J. Paxos made live – an engineering perspective (invited talk). In *Proceedings of PODC* (2007), vol. 7.

[13] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of OSDI* (2006), pp. 205–218.

[14] Corbett, J.C. and Dean, J. and Epstein, M. and Fikes, A. and Frost, C. and Furman, JJ and Ghemawat, S. and Gubarev, A. and Heiser, C. and Hochschild, P. and others. Spanner: Googles Globally-Distributed Database. In *Proceedings of OSDI* (2012).

[15] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of OSDI* (Jan. 2004), p. 10.

[16] Druschel, P. and Banga, G. and Mogul, JC. Resource containers: A new facility for resource management in server systems. In *Proceedings of OSDI* (1999).

[17] Engler, D., and Kaashoek, M. Exterminate all operating system abstractions. In *Proceedings of HotOS* (1995), pp. 78–83.

[18] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. In *SIGOPS Operating Systems Review* (New York, NY, USA, 2003), vol. 37, ACM, pp. 29–43.

[19] Ghodsi, A. and Zaharia, M. and Hindman, B. and Konwinski, A. and Shenker, S. and Stoica, I. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of NSDI* (2011).

[20] Giampapa, M. and Gooding, T. and Inglett, T. and Wisniewski, R.W. Experiences with a lightweight supercomputer kernel: Lessons learned from Blue Gene's CNK. In *Proceedings of IEEE SC* (2010).

[21] Hand, S., Warfield, A., Fraser, K., Kotsovinos, E., and Magenheimer, D. Are virtual machine monitors microkernels done right? In *Proceedings of HotOS* (2005), USENIX Association, pp. 1–1.

[22] HÄRTIG, H., HOHMUTH, M., LIEDTKE, J., SCHÖNBERG, S., AND WOLTER, J. The performance of -kernel-based systems. In *Proceedings of SOSP* (1997), pp. 66–77.

[23] HEISER, G. AND ELPHINSTONE, K. AND KUZ, I. AND KLEIN, G. AND PETTERS, S.M. Towards trustworthy computing systems: taking microkernels to the next level. *ACM SIGOPS Operating Systems Review 41*, 4 (2007), 3–11.

[24] HEISER, G. AND ELPHINSTONE, K. AND VOCHTELOO, J. AND RUSSELL, S. AND LIEDTKE, J. The Mungi single-address-space operating system. *Software: Practice and Experience 28*, 9 (1998), 901–928.

[25] HOELZLE, U., AND BARROSO, L. A. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 1st ed. Morgan and Claypool Publishers, 2009.

[26] HOLLAND, D.A. AND SELTZER, M.I. Multicore OSes: looking forward from 1991, er, 2011. In *Proceedings of HotOS* (2011), pp. 9–11.

[27] HUNT, P. AND KONAR, M. AND JUNQUEIRA, F.P. AND REED, B. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of USENIX ATC* (2010).

[28] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review 41*, 3 (June 2007), 59.

[29] ISARD, M., PRABHAKARAN, V., CURREY, J., WIEDER, U., TALWAR, K., AND GOLDBERG, A. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of SOSP 2009* (2009), pp. 261–276.

[30] KLEIN, G., ELPHINSTONE, K., AND HEISER, G. seL4: Formal verification of an OS kernel. In *Proceedings of SOSP* (2009), pp. 207–220.

[31] LEACH, P., LEVINE, P., DOUROS, B., HAMILTON, J., NELSON, D., AND STUMPF, B. The architecture of an integrated local network. *Selected Areas in Communications, IEEE Journal on 1*, 5 (nov 1983), 842 – 857.

[32] LESLIE, I., MCAULEY, D., AND BLACK, R. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications 14*, 7 (1996), 1280–1297.

[33] LEVIN, R., COHEN, E., CORWIN, W., POLLACK, F., AND WULF, W. Policy/mechanism separation in hydra. In *Proceedings of SOSP* (1975), pp. 132–140.

[34] LIEDTKE, J. Improving IPC by kernel design. *ACM SIGOPS Operating Systems Review* (1994), 175–188.

[35] MADHAVAPEDDY, ANIL AND MORTIER, RICHARD AND ROTSOS, CHARALAMPOS AND SCOTT, DAVID AND SINGH, BALRAJ AND GAZAGNAIRE, THOMAS AND SMITH, STEVEN AND HAND, STEVEN AND CROWCROFT, JON. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of ASPLOS* (2013). To appear.

[36] MARS, J. AND VACHHARAJANI, N. AND HUNDT, R. AND SOFFA, M.L. Contention aware execution: online contention detection and response. In *Proceedings of CGO* (2010), pp. 257–265.

[37] MULLENDER, S., VAN ROSSUM, G., TANANBAUM, A., VAN RENESSE, R., AND VAN STAVEREN, H. Amoeba: a distributed operating system for the 1990s. *Computer 23*, 5 (may 1990), 44 –53.

[38] 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).

[39] NIGHTINGALE, E. AND ELSON, J. AND HOFMANN, O. AND SUZUE, Y. AND FAN, J. AND HOWELL, J. Flat Datacenter Storage. In *Proceedings of OSDI* (2012).

[40] OUSTERHOUT, J., CHERENSON, A., DOUGLIS, F., NELSON, M., AND WELCH, B. The sprite network operating system. *Computer 21*, 2 (feb. 1988), 23 –36.

[41] PAN, X., TAN, J., KAVULYA, S., GANDHI, R., AND NARASIMHAN, P. Ganesha: Black-Box Fault Diagnosis for MapReduce Systems (CMU-PDL-08-112). Tech. rep., CMU Parallel Data Laboratory, 2008.

[42] POPEK, G. J., AND GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. *Commun. ACM 17*, 7 (July 1974), 412–421.

[43] PORTER, D.E. AND BOYD-WICKIZER, S. AND HOWELL, J. AND OLINSKY, R. AND HUNT, G.C. Rethinking the library OS from the top down. In *Proceedings of ASPLOS* (2011), pp. 291–304.

[44] REISS, C. AND TUMANOV, A. AND GANGER, G.R. AND KATZ, R.H. AND KOZUCH, M.A. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of SoCC* (2012).

[45] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop distributed file system. In *Proceedings of IEEE MSST* (2010), pp. 1–10.

[46] SMALL, C., AND SELTZER, M. Vino: an integrated platform for operating systems and database research. Tech. rep., Harvard University, 1994.

[47] STEINBERG, U., AND KAUER, B. NOVA: a microhypervisor-based secure virtualization architecture. In *Proceedings of EuroSys* (2010), pp. 209–222.

[48] VARIAN, M. Vm and the vm community: Past, present, and future. *SHARE 89* (August 1997), Sessions 9059–9061.

[49] WATSON, R.N.M. AND ANDERSON, J. AND LAURIE, B. AND KENNAWAY, K. Capsicum: practical capabilities for UNIX. In *Proceedings of USENIX Security* (2010).

[50] WAYCHINSON, MIKE AND CORBET, JONATHAN. KS2009: How Google uses Linux, 2009. `http://lwn.net/Articles/357658/`.

[51] WILKES, M. V. *The Cambridge CAP computer and its operating system (Operating and programming systems series).* North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 1979.

[52] WITTORFF, V.W. Control of data in communication networks, Mar. 6 2008. US Patent 8,130,652.

[53] 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).

[54] ZAHARIA, M., KONWINSKI, A., JOSEPH, A., KATZ, R., AND STOICA, I. Improving MapReduce performance in heterogeneous environments. In *Proceedings of OSDI* (2008), pp. 29–42.