DBOS: A DBMS-oriented Operating System

Athinagoras Skiadopoulos[1*], Qian Li[1*], Peter Kraft[1*], Kostis Kaffes[1*], Daniel Hong[2], Shana Mathew[2], David Bestor[2], Michael Cafarella[2], Vijay Gadepally[2], Goetz Graefe[3], Jeremy Kepner[2], Christos Kozyrakis[1], Tim Kraska[2], Michael Stonebraker[2], Lalith Suresh[4], and Matei Zaharia[1]
Stanford[1], MIT[2], Google[3], VMware[4]
dbos-project@googlegroups.com

ABSTRACT

This paper lays out the rationale for building a completely new operating system (OS) stack. Rather than build on a single node OS together with separate cluster schedulers, distributed filesystems, and network managers, we argue that a distributed transactional DBMS should be the basis for a scalable cluster OS. We show herein that such a database OS (DBOS) can do scheduling, file management, and inter-process communication with competitive performance to existing systems. In addition, significantly better analytics can be provided as well as a dramatic reduction in code complexity through implementing OS services as standard database queries, while implementing low-latency transactions and high availability only once.

PVLDB Reference Format:
Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael Cafarella, Vijay Gadepally, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. DBOS: A DBMS-oriented Operating System. PVLDB, 15(1): 21-30, 2022.
doi:10.14778/3485450.3485454

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/DBOS-project.

1 INTRODUCTION

At Berkeley, one of us was an early user of Unix in 1973. Linux debuted 18 years later in 1991 with the same general architecture. Hence, the ideas in prevailing operating systems are nearly half a century old. In that time, we note the following external trends:

(1) Scale Today, enterprise servers routinely have hundreds of cores, terabytes of main memory, and hundreds of terabytes of storage, a stark contrast from the uniprocessors of yesteryear. A large shared system such as the MIT Supercloud [12, 42] has approximately 10,000 cores, a hundred terabytes of main memory, and petabytes of storage. Obviously, the resources an OS is called to manage have increased by many orders of magnitude during this period of time.

(2) Clouds The popularity and scale of public cloud services generate ever larger configurations, amplifying the management problem.

(3) Parallel computation Most data warehouse DBMSs harness a "gang" of CPUs to solve complex queries in parallel. The same observation can be applied to parallel computing platforms like Apache Spark. In fact, one of us reported that it is common for a large cloud to be managing 10^8 Spark tasks at a time [7]. Interactive parallel computing systems require launching thousands of jobs in a few seconds [21, 32]. Obviously, one needs "gang" scheduling and straggler mitigation support.

(4) Heterogeneous hardware Hardware like GPUs, TPUs, Intelligent SSDs, and FPGAs have become omnipresent in large configurations and bring new optimization opportunities and constraints [22, 43]. Heterogeneous multi-tier memory systems and even non-deterministic systems are just around the corner. Such special purpose hardware should be managed by system software. Ideally, an OS should be able to manage tasks that require multiple kinds of computing assets.

(5) New applications The need for the hardware mentioned above is driven by performance/cost requirements of new applications, most recently machine learning (ML), Internet of Things (IoT), and "big data" applications [41].

(6) New programming models Users ideally want to pay for resources only when they are active. All major cloud vendors now offer serverless computing APIs [9] where a user divides their computation into short "tasks" with communication between tasks supported through an object store. Resources can scale from zero to thousands and back to zero in seconds. The user only pays for the resources used in each such task, without explicitly provisioning machines or otherwise paying for idle resources waiting for input. According to one estimate [40] 50% of AWS customers are using Lambda, up from 35% a year ago. Obviously, this programming model is enjoying widespread acceptance.

(7) Age Linux is now "long in the tooth" having been feature-enhanced in an ad-hoc fashion for 30 years. This has resulted in slow forward progress; for example, Linux has struggled for a decade to fully leverage multi-cores [14, 20, 22, 34].

(8) Provenance It is becoming an important feature for policy enforcement, legal compliance, and data debugging. Provenance data collection touches many elements of the system but is totally absent in most current OSes.

We note that Unix/Linux falls far short of adequately supporting these trends. At scale, managing system services is a "big data" problem, and Linux itself contains no such capabilities. As we noted, it has struggled for a decade to support multiprocessors in a single node and has no support for the multiple nodes in a cluster. Scheduling and resource management across a cluster must be accomplished by another layer of management software such as Kubernetes or Slurm [41]. The two layers are not well integrated,

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 15, No. 1 ISSN 2150-8097.
doi:10.14778/3485450.3485454

*These authors contributed equally to this submission.

so many cross-cutting issues, such as provenance, monitoring, security and debugging, become very challenging on a cluster and often require custom built solutions [51]. Furthermore, Linux has weak support for heterogeneous hardware, which must be managed by non-OS software. Even for common hardware such as network cards and storage, practitioners are increasingly using kernel bypass for performance and control [19]. Lastly, serverless computing can be supported by a much simpler runtime; for example, there is no need for demand paging. Hence, some functions in Linux are possibly not needed in the future.

Unix offers abstractions that are too few and too low-level for managing the multiple levels of complexity and huge amounts of state that modern systems must handle. Our conclusion is that the "everything is a file" model for managing uniprocessor hardware [44] — a revolutionary position for Unix in 1973 — is ill-suited to modern computing challenges. Layering an RPC library on top of the Unix model in order to build distributed systems does not address the abstraction gap. The current model must be replaced by a new architecture to make it significantly easier to build and scale distributed software. In Section 2, we specify such an architecture based on a radical change towards an "everything is a table" abstraction that represents all OS state as relational tables, leveraging modern DMBS technology to scale OS functionality to entire datacenters [23, 33]. In Section 3, we indicate our "game plan" for proving our ideas. Our timeline contains three phases, and we report on the results of the first phase in Section 4. Section 5 discusses related work, and Section 6 contains our conclusions.

2 RETHINKING THE OS

The driving idea behind our thinking is to support vast scale in a cluster or data center environment. Fundamentally, the operating system resource management problem has increased by many orders of magnitude and is now a "big data" problem on its own. The obvious solution is to embed a modern high performance multi-node transactional DBMS into the kernel of a next-generation OS. As a result, we call our system DBOS (Data Base Operating System). This results in a very different OS stack.

2.1 The DBOS Stack

Our four-level proposal is shown in Figure 1.

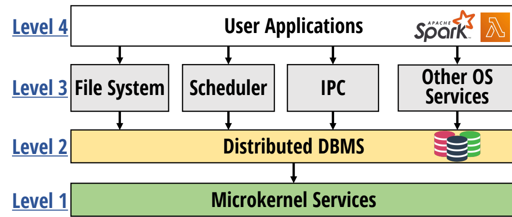


Figure 1: Proposed DBOS stack. Level 1 is the bottom layer.

Level 4: User space — At the top level are traditional user-level tasks, which run protected from each other and from lower levels, as in traditional OSs. In DBOS, we primarily target *distributed applications*, such as parallel analytics, machine learning, web search, and mobile application backends, which are the most widely used applications today. To support these applications, we encourage a serverless model of computation, whereby a user decomposes her task into a graph of subtasks, each running for a short period of time. Hence, subtasks come into existence, run, and then die in a memory footprint they specify at the beginning of their execution. As such, we do not plan to support the complex memory management facilities in current systems; a subtask can only run if there is available memory for its footprint. The serverless computing model is supported by capabilities at levels 2 and 3 of the DBOS stack.

Common serverless models (e.g. AWS Lambda) dictate that data be passed between subtasks via a shared object store or filesystem. This may due to the difficulties in current systems in managing task sets, and in locating the recipients of interprocess communication (IPC) messages. In a DBOS environment that stores all OS state in a distributed DBMS, neither impediment exists. Hence, DBOS provides fast IPC between live subtasks, along with table-based data sharing. Both capabilities are supported via DBMS-based primitives.

In addition, DBOS's cluster-scale approach makes it significantly easier for developers to monitor, debug, and secure their distributed applications. Today's distributed computing stacks, such as Kubernetes, provide few abstractions beyond the ability to launch tasks and communicate between them via RPC, so they require developers to build or cobble together a wide range of external tools to monitor application metrics, collect logs, and enforce security policies [10, 27, 39, 45]. In contrast, in DBOS, the entire state of the OS and the application is available in structured tables that can simply be queried using SQL. For example, finding all the process IDs in an application, their current memory usage, or aggregate information about custom metrics in the application is trivial; it is also easy to ask more complicated questions, such as finding all processes from users in a certain organization. Likewise, all in-flight IPC messages are queriable as a table, and can be retained for later analysis. We believe that these capabilities will greatly simplify the development of distributed applications, in the same way that developing and debugging single-process multi-core applications today is dramatically easier than working with distributed ones.

Level 3: OS Functionality The applications in level 4 are supported by a set of data-center OS services in level 3, such as task schedulers, distributed filesystems, interprocess communication (IPC) and others. We demonstrate high-performance implementations of these services later in the paper (Section 4).

In DBOS, all OS services benefit from capabilities provided by the distributed DBMS (level 2), such as high-availability, transaction support, security, and dynamic reconfiguration. These capabilities are known pain points in today's cluster managers, which routinely re-invent these wheels and yet, provide weaker guarantees than what a DBMS provides. For example, the Kubernetes control plane cannot perform multi-object transactions [11], and the HDFS NameNode requires more servers for a highly available configuration than if they had used a distributed DBMS [37].

In DBOS, all OS services are implemented using a combination of SQL and user-defined functions. These services all operate on a *consistent global view* of the OS state in the form of DBMS tables. This makes it easy for services to support cross-cutting operations. For example, modern task schedulers routinely rely on historical performance profiles of the tasks, data placement information, resource utilization, machine properties and myriad other state to make high quality placement decisions. A scheduler in today's cluster managers like Kubernetes and YARN needs to gather such

information from disparate layers, each with ad-hoc APIs to expose the required state, with no consistency guarantees between layers.

Similarly, our programming model should make it easier to build novel OS services from scratch — say, a privacy reporting tool about which files' data have been transmitted over the network — far more easily than with today's cluster managers, which would require intrusive changes to support similar features.

Level 2: The DBMS — In level 2 we propose to utilize a high-performance multi-node main-memory transactional DBMS. There are several such DBMSs in commercial service today including VoltDB [8] (and its predecessor H-Store [3]), SAP-Hana [13], Microsoft Hekaton [25], and SingleStore (MemSQL) [15], just to name a few. These systems are all SQL databases, offer multi-node support, low latency distributed transactions (concurrency control and crash recovery) and most offer real-time failover (high availability). The fastest of these are capable of millions of transactions per second on modest hardware clusters costing a few thousand dollars. These standard DBMS services, combined with some standardized OS-specific schemas, comprise all of the second level.

We have restricted our attention to SQL DBMSs in order to get the analytic power of a high level query language. However, there are many No-SQL DBMSs with lower-level interfaces that could also be considered in a DBOS-style architecture.

Level 1: Microkernel services — We expect a DBMS to be runnable on top of a minimal microkernel in level 1. It comprises raw device handlers, interrupt handlers, very basic inter-node communication, and little else.

Long ago, DBMSs tended not to use OS services, and one of us wrote a paper in 1981 complaining about this fact [47]. There is no reason why we cannot return to running a DBMS on a "raw device". Moreover, DBMSs do their own admission control, so as not to overtax the concurrency control system. Also, a DBMS running near the bottom of the OS stack will be processing very large numbers of short transactions. As such, we want simple preemption to allow the DBMS to run whenever there is work to do. A DBMS can also control its own memory footprint by dynamically linking and unlinking user-defined functions as well as little-used support routines.

In DBOS we do not plan to support sophisticated memory management. The DBMS in level 2 does its own memory management. One might wonder if level 2 should also run in a Lambda-style serverless environment. In fact, it would certainly be possible to specify that nodes in a query plan be serverless computations. However, doing so would force serverless support into layer 1, which we want to keep as small as possible. Also, we expect the DBMS to be running millions of transactions per second, so all the pieces of a DBMS would, in fact, be memory resident essentially all the time. Hence, limited value would result from such a decision.

2.2 Design Discussion

It is worth considering the core benefits of the DBOS architecture. OS state in current operating systems is typically supported piecemeal, usually in various data structures with various semantics. Moving to DBOS will force DBMS schema discipline on this information. It will also allow querying across the range of OS data using a single high-level language, SQL. In addition, transactions, high availability and multi-node support are provided exactly once, by the DBMS, and then used by everybody. This results in much simpler code, due to avoiding redundancy. Also, current non-transactional data structures get transactions essentially for free.

Effective management and utilization of large-scale computing systems require monitoring, tracking, and analyzing the critical resources, namely processing, storage, and networking. Currently, additional utilities and appliances are often built or purchased and integrated into each of these functions [10, 27, 39, 45]. These capabilities provide database functionality to the process scheduling logs, storage metadata logs, and network traffic logs. As separate adjunct bolt-on capabilities are highly sensitive to processing, storage, and network upgrades, they require significant resources to maintain and significant knowledge by the end-user to use effectively. DBOS obviates the need for these separate add-on capabilities.

The principal pushback we have gotten from our initial proposal [23] is "you won't be able to offer a performant implementation". Of course, we have heard this refrain before — by the CODASYL advocates back in the 1970's. They said "you can't possibly do data management in a high-level language (tables and a declarative language). It will never perform." History eventually proved them wrong. Our goal in DBOS is analogous, and we turn in the next section to three successive prototypes with that goal.

3 DBOS STAGES

We will build out the DBOS prototype in three stages of increasing strength: from "straw", to "wood", and finally "brick".

3.1 DBOS-straw

Our first prototype demonstrates that we can offer reasonable performance on three OS services: scheduling tasks, providing a filesystem, and supporting interprocess communication (IPC). Constructing the DBOS-straw prototype entails using Linux for level 1, an RDBMS for level 2, writing a portion of level 3 by hand, and creating some test programs in level 4. Results in these areas are documented in Section 4 of this paper. We expect this exercise should convince the naysayers of the viability of our proposal. This system builds on top of VoltDB, which offers required high performance.

3.2 DBOS-wood

With the successful demonstration of DBOS-straw, we will install our prototype in a Linux cluster in user code and begin supporting DBOS functions in a real system. We expect to use a serverless environment to support user-level tasks. Hence, any user "process" is a collection of short running "tasks" assembled into a graph. We expect this graph to be stored in the DBMS to facilitate better scheduling, and tasks, to the extent possible, will be user-defined DBMS functions. We are currently implementing a DBOS-based serverless environment. Moreover, we will "cut over" a sample of Linux OS functions to our model, implemented in level 3 DBOS code. The purpose of DBOS-wood is to show that OS functions can be readily and compactly coded in SQL and that our filesystem, scheduling and IPC implementations work well in a real system. The implementation of DBOS-wood is now underway. When it is successfully demonstrable, we will move on to DBOS-brick.

3.3 DBOS-brick

Our last phase will be to beg, borrow, steal, or implement a microkernel for level 1 and potentially revisit the DBMS decision in level

2. On top of this framework we can port our serverless environment and the collection of OS services from DBOS-wood. In addition, we expect to reimplement enough level 3 OS services to have a viable computing environment. During this phase, we expect to receive substantial help from our industrial partners (currently Amazon, Google and VMWare), since this is a sizeable task.

We expect to have sufficient resources to implement level 1 from scratch, if necessary. Level 2 will be too large a project for us to implement from scratch, so we expect to adapt existing DBMS code to the new microkernel, perhaps from VoltDB or H-Store.

4 DBOS-STRAW

We first discuss a few characteristics of our chosen DBMS, VoltDB. Then we turn to the DBOS-straw implementations of the scheduling, IPC, and the filesystem. We show that the DBOS architecture can deliver the performance needed for a practical system.

4.1 Background

4.1.1 VoltDB. As noted earlier, there are several parallel, high performance, multi-node, transactional DBMSs with failover that we could have selected. VoltDB was chosen because of the relationship one of us has with the company. VoltDB implements SQL on top of tables which are hash-partitioned on a user specified key across multiple nodes of a computer system. VoltDB supports serializability and transactional (or non-transactional) failover on a node failure. It is optimized for OLTP transactions (small reads and writes) and insists transactions be user-defined DBMS procedures [16] which are compiled and aggressively optimized. All data is main-memory resident and highest performance is obtained when:
(1) Transactions access data in only a single partition; concurrency control is optimized for this case.
(2) The user task is on the same node as the single partition accessed. This avoids network traffic while executing a transaction.

4.1.2 Supercloud. We ran all of our experiments on the MIT Supercloud [12]. Supercloud nodes have 40-core dual-socket Intel Xeon Gold 6248 2.5GHz CPUs, 378GB memory, a Mellanox ConnectX-4 25Gbps NIC, and a 4.4TB HDD. We use up to 8 nodes (320 cores) of VoltDB. We use a configuration with 1 partition per core, so tables are partitioned 320 ways in a maximum configuration.

4.2 Scheduling Studies

We assume that in DBOS, a scheduler runs on each partition of our VoltDB installation. To implement a multi-node scheduler in SQL, we must create relational tables to store scheduler metadata. The first of these is a **Task** table; the scheduler adds a row to this table whenever a task is created. Scheduling the task to a worker can be done at creation time or deferred; we will show example schedulers that do each. The fields of the **Task** table are:

```
Task (p_key, task_id, worker_id, other_fields)
```

The `p_key` is the current partition of the task. The **Task** table is partitioned on this field. If task assignment is done at creation time, the `worker_id` field stores the worker to which the task is assigned. If scheduling is deferred, that field is instead initialized to `null`; when the scheduling decision is made, the task will move to the assigned worker's partition and the field will be set. `other_fields` hold scheduling relevant information, such as priority, time the task was created, time since the task ran last, etc. Such information might be useful for debugging, provenance, and data analytics.

```
schedule_simple(P, TID) {
  select worker_id,unused_capacity from Worker
    where unused_capacity>0 and p_key=P
    limit 1;
  if worker_id not None:
    WID = worker_id[0];
    UC = unused_capacity[0];
    update Worker set
      unused_capacity=UC - 1
      where worker_id=WID and p_key=P;
    insert into Task (P, TID, WID,...);
}
```

Figure 2: Simple FIFO scheduler.

Our schedulers also require a **Worker** table:

```
Worker (p_key, worker_id, unused_capacity)
```

Each worker is associated with a VoltDB partition defined by its `p_key`. The **Worker** table is partitioned on this field. Workers have capacity in units of runnable tasks. The table specifies the unused capacity of each worker. We can extend this table to add fields such as accelerator availability or other worker metadata.

It is the job of the scheduler to assign an unassigned task to some worker and then decrement its unused capacity. It is possible to implement a diverse range of schedulers by changing the stored procedure which performs this assignment.

In Figure 2 we sketch a stored procedure implementing a simple FIFO scheduler with a mix of SQL and imperative code. The input to this scheduler is a randomly chosen partition `P` and the unique identifier of the task `TID` to be scheduled. The scheduler first selects at most one worker (`limit 1`) that has unused capacity (`unused_capacity>0`) from the target partition (`p_key=P`). If one is found, the scheduler decrements the selected worker's unused capacity by one in the **Worker** table. Then, the scheduler inserts a row for task `TID` into the **Task** table. The scheduler iterates through random partitions until it finds one with unused capacity for the task. In the interest of brevity, we omit dealing with the corner case when worker capacity is completely exhausted.

To demonstrate this simple scheduler's performance, we implement it in VoltDB as a stored procedure and measure median and tail latencies as we vary system load. Our experiments used forty parallel schedulers on two VoltDB hosts and two client machines. They are synthetic–tasks are scheduled, but not executed. Hence, the latencies reflect the scheduling overhead of DBOS. As shown in Figure 5, the simple FIFO scheduler can schedule 750K tasks per second at a sub-millisecond tail latency while the median latency remains around 200 μs even at 1M tasks/sec load, which outperforms most existing distributed schedulers.

However, this simple scheduler has some disadvantages. For example, it has no notion of locality, but many tasks perform better when co-located with a particular data item. Moreover, it may need multiple tries to find a worker with spare capacity.

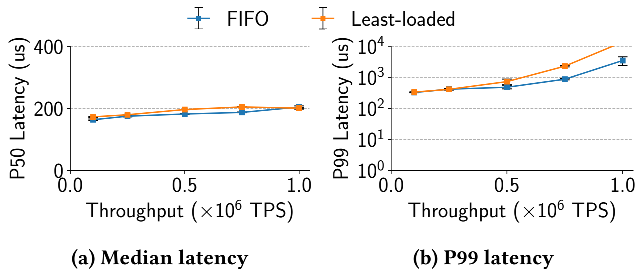
Our second scheduler (locality-aware scheduler, Figure 3) attempts to avoid both issues. Specifically, it schedules the task at the worker machine where the task's home directory partition resides.

```
schedule_locality(HP, TID) {
  ... // Same code as FIFO scheduler
  if worker_id not None:
  ... // Same code as FIFO scheduler
  else:
    insert into Task (HP, TID, null, ...);
}
```

Figure 3: Locality-aware scheduler.

```
schedule_least_loaded(P, TID) {
  select worker_id,unused_capacity from Worker
    where unused_capacity>0 and p_key=P
    order by unused_capacity desc
    limit 1;
  ... // Same code as FIFO scheduler
}
```

Figure 4: Least-loaded scheduler.



(a) Median latency **(b) P99 latency**

Figure 5: Performance of schedulers.

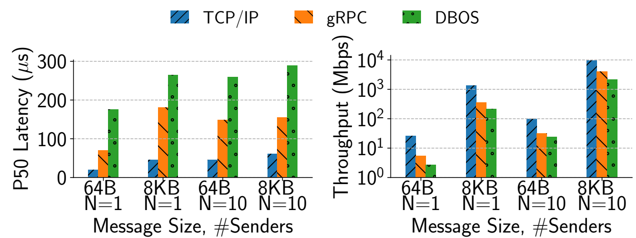
We can query this information from our filesystem (§4.4). The input to this scheduler is the home directory partition HP and the ID of the task TID to be scheduled. The first three SQL queries in its stored procedure are similar to those in the FIFO scheduler; the difference is the `insert` query at the end (highlighted in red in Figure 3). This queues tasks that cannot be immediately run at their target partition until it has unused capacity. Hence, this locality-aware scheduler may defer task assignment under high load.

Our last scheduler (least-loaded scheduler, Figure 4) ensures the best load balance possible, assuming fairly long running tasks. This scheduling strategy is commonly used in many existing systems. Specifically, we schedule the next incoming task to the worker with the greatest unused capacity within a partition.

Implementing this scheduler only required changing a single line of code to the simple FIFO scheduler, i.e., adding a '`order by unused_capacity desc`' clause (highlighted in red in Figure 4). As shown in Figure 5, this scheduler is only slightly slower than the previous two because of the additional operation. If we wanted to find the least-loaded worker across all partitions, we could simply remove the '`p_key=P`' clause. This showcases the strength and expressiveness of the relational interface–significant changes to scheduler behavior require only a few lines of code to implement.

4.3 IPC

4.3.1 DBMS-backed IPC. Inter-process communication is the task of sending messages from a sender to a receiver. Typical messaging systems, such as TCP/IP and gRPC [2] provide reliable delivery,



(a) Median latency **(b) Throughput**

Figure 6: Performance of ping-pong benchmark.

in-order delivery, exactly-once semantics, and flow control. Implementing a system with stronger guarantees is straightforward using a multi-node DBMS backend. A **Message** table is required:

```
Message(sender_id, receiver_id, message_id, data)
```

This table is partitioned on `receiver_id`. To send a message, the sender just adds a row to this table, which is a single-partition insert to a remote partition. To read a message, the receiver queries this table via a local read. These two commands are transactional.

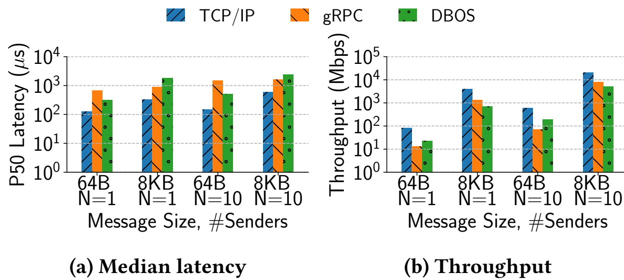
If we replicate the **Message** table, failover will allow the IPC system to continue in the face of failures, without loss of data, a stronger guarantee than what TCP or existing RPC systems provide. This is a required feature of a message system. Another feature, in-order delivery, can be achieved by indexing the **Message** table on an application- or library-specific `message_id` field. For exactly-once semantics, the receiver just needs to delete each message upon receiving it. Lastly, DBMSs can store massive amounts of data, so we expect flow control will not be needed.

4.3.2 Limitations. One limitation of our approach is that the receiver must poll the **Message** table periodically if expecting a message, which may increase CPU overhead. However, support for database triggers would avoid polling altogether. Several DBMSs implement triggers, e.g., Postgres [48], but VoltDB does not. A production implementation of DBOS would require a trigger-like mechanism to avoid the "busy waiting" that polling entails.

4.3.3 Evaluation. We compare the performance of our DBMS-backed IPC (DBOS) against two baselines, gRPC [2] and bare-bones TCP/IP. gRPC is one of the most widely-used messaging libraries today and offers most of the features that we aim to support. TCP/IP is also a widely used communications substrate. In our benchmarks, the DBOS receivers and senders run on separate VoltDB hosts. gRPC runs natively on top of TCP/IP, with SSL disabled. TCP/IP is an OS service running on top of Supercloud's network fabric.

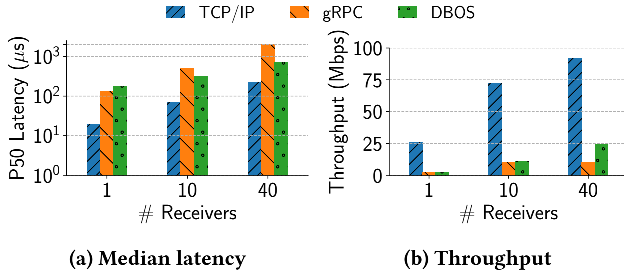
We first measure DBOS's performance in a *ping-pong* benchmark. The sender sends a message to the receiver which replies with the same message. We vary the message size as well as the number of concurrent senders and receivers, each in its own thread. In Figure 6 we show the (a) median latency and (b) throughput of the different messaging schemes. DBOS achieves 24%–49% lower throughput and 1.3 – 2.5× higher median latency compared to gRPC, and DBOS achieves 4–9.5× lower performance than TCP/IP.

The comparison against gRPC is more favorable for DBOS in the other two benchmarks we tested. Similar to the first benchmark, Figure 7 shows the performance of a ping^{20}-pong^{20} benchmark



(a) Median latency (b) Throughput

Figure 7: Performance of ping^{20}-pong^{20} benchmark.



(a) Median latency (b) Throughput

Figure 8: Performance of multicasting benchmark.

where each sender sends 20 messages to the receiver, expecting 20 messages back. This workload is characteristic of batch communication applications such as machine learning inference serving.

Here, DBOS outperforms gRPC by up to 2.7× with small messages, while it achieves 48% lower performance with 8KB messages. The small performance gap between DBOS and gRPC is impressive considering that the DBOS message scheme is implemented in a few lines of SQL code running on top of an unmodified DBMS, while gRPC is a specialized communication framework developed by many engineers over many years. The gap between TCP/IP and DBOS has narrowed somewhat, but it is still substantial.

Figure 8 shows the results of our third benchmark. In this benchmark, a single sender sends a small 64-byte message to a varying number of receivers, expecting their replies. This workload is characteristic of fan-out communication applications such as web search or parallel aggregation. The results show that as the number of receivers increases, the performance difference between DBOS and gRPC widens, which is in agreement with the previous results on small messages. For 40 receivers, DBOS achieves 2.3× higher throughput and 64% lower median latency than gRPC. Not surprisingly, the gap between DBOS and TCP/IP is still substantial.

We are encouraged by these results for the following reasons. First, VoltDB uses TCP/IP as its message substrate. In DBOS-wood, we plan to run on a bare-bones data transport, which should bridge much of these gaps. Second, DBOS uses polling, another source of significant overhead, which we expect to eliminate in DBOS-wood. Furthermore, DBOS requires an additional copy relative to the other schemes. A future DBMS could avoid this overhead.

Our conclusion is that DBOS IPC can be made a great deal more efficient. However, even in its current form, DBOS is reasonably competitive against gRPC. Since this is the most popular messaging system, we are encouraged that DBOS IPC is "fast enough".

Also, it should be noted clearly that new messaging services can be quickly coded in SQL. In current hard-coded systems they require extensive development. For example, a "hub and spoke" implementation of messaging is a few lines of SQL.

```
Map (p_key, partition_id, host_name, host_id)
User (user_name, home_partition, current_path)
Directory (d_name, content, content_type,
  user_name, protection_info, p_key)
Localized_file (f_name, f_owner, block_no, bytes,
  f_size, p_key)
Parallel_file (f_name, f_owner, block_no, bytes,
  f_size)
```

Figure 9: Filesystem tables for data and metadata.

4.4 The DBOS Filesystems

DBOS supports two filesystems, both including the standard POSIX-style interface. The first filesystem stores all data for a user, U, on a single VoltDB partition divided into blocks of size B. In this implementation, the file table, noted below, is partitioned on `user_name`. This will ensure that U is localized to a single partition and offers very high performance for small files.

The second filesystem partitions files on `block_no`, thereby spreading the blocks of a file across all VoltDB partitions. In this case, reads and writes can offer dramatic parallelism, but lose node locality. For this paper's scope we focus on in-memory operations, so we leave the implementation of a spilling disk scheme, similar to the one used in the H-Store project [3], to future work. Figure 9 shows the tables that contain filesystem data and metadata. These tables are accessed directly by DBOS level 3 SQL services, in order to provide functionality to user code in level 4.

The **Map** table specifies the physical local VoltDB partitions that implement a given VoltDB database. `partition_id` is the primary key, and is a foreign key in the other tables. The Map table is read-almost-always and is replicated on all nodes of a database.

The **User** table specifies the home partition (`partition_id`) for each user. This partition holds their file directory structure. This table is also replicated on all partitions. The **Directory** table holds standard directory information, and is partitioned on `p_key`. The **Localized_file** table holds the bytes that go in each block in a localized file. It is partitioned on `p_key`. The **Parallel_file** table holds the bytes for parallel (partitioned) files. It has the same information as the previous table. However, file bytes must be stored in all partitions, so this table is partitioned on `block_no`.

Our design uses fully qualified Linux file names for `f_names`. Traversing directories generates simple lookup queries and the "current path" is stored in the **User** table. That way there is no need for "open" and "close", so both are no-ops in DBOS.

To access block B in localized File F with a fully qualified name for user U, we can look up the home partition from **User**, if it is not already cached. With that partition key, we can access **Localized_file**. Parallel files are automatically in all partitions. With access to the VoltDB hashing function, we know which partition every block is in. However, it will be rare to read a single block from a parallel file. In contrast, "big reads" can send a query with a block range to all partitions that will filter to the correct blocks.

For both implementations, there are stored procedures for all filesystem operations. In the case of a localized file, the VoltDB "stub"

in the user task sends the operation to the node holding the correct file data, where a stored procedure performs the SQL operation and returns the desired result. For partitioned tables, the stub invokes a local VoltDB stored procedure, which fans the parallel operations to the various partitions with data. These partitions return the data directly to the stub which collates the returns and alerts the task.

Note that block-size can be changed by a single SQL update to the length of the "bytes" field in either file table. Also, it is straightforward to change the DBOS filesystem into an object store, again with modest schema changes. Lastly, it is not a difficult design to allow the file creator complete control over the placement of blocks in a parallel file.

For our first experiment, we run our localized file system with a Supercloud configuration of 1 VoltDB node (40 partitions) for 40 users. Each user has 100 files of size 256KB split into 128 2KB blocks and runs the following operation loop:

```
while(true) {
  Open a random file
    Read or write twenty random 2KB blocks
      with equal probability
  Close the file
}
```

We compare our VoltDB filesystem with the Linux filesystem ext4 [4]. Ext4 is a journaling filesystem that uses delayed allocation, i.e., it delays block allocation until data is flushed to disk. It should be clearly noted that this comparison is "apples-to-oranges", since we are comparing a VoltDB implementation of a transactional, multi-node filesystem with a direct implementation of a local, non-transactional one.

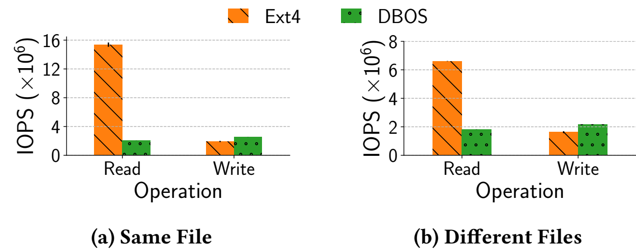
As we can see in Figure 10a, DBOS matches or exceeds ext4's write performance because it avoids global locks that become a bottleneck for ext4 [36]. However, ext4 read performance considerably exceeds that of DBOS. The basic reason is the invocation cost of VoltDB is around 40 microseconds, whereas the cost of a Linux system call is approximately 1 microsecond. In a production DBOS deployment we would need to reduce this latency by, for example, using shared memory for task communication.

To demonstrate the significance of invocation overhead, we ran a slightly different benchmark where each user reads 20 blocks, each from a different file.

```
while(true) {
  Repeat twenty times
    Open a random file
      Read or write one random 2KB blocks
        with equal probability
    Close the file
}
```

In this case, Linux will need multiple system calls, while DBOS still requires only one. As can be seen in Figure 10b, the performance gap between DBOS and ext4 has narrowed somewhat.

Besides reading and writing data, there are two other operations a filesystem must do. The first is metadata operations such as creating or deleting files. The second is analytics operations such as finding the size of a directory or listing its contents. We now benchmark each operation.



(a) Same File **(b) Different Files**

Figure 10: Throughput for 40 clients that read 20 blocks of 2KB each from (a) the same and (b) different files.

Table 1: Performance of file operations.

Operation	FS	Avg Latency (μs)	Max Ops/sec ($\times 10^3$)
Create File	Ext4	656.78 (±8.99)	31.39 (±0.56)
	DBOS	67.48 (±6.98)	303.85 (± 1.33)
Delete File	Ext4	654.04 (±8.99)	30.53 (± 0.61)
	DBOS	65.58 (±7.1)	302.30 (±2)

Table 2: Performance and LoC of conditional size aggregate.

Language	Time (msec)	Lines of Code
C++	9.90	98
SQL	0.65	2

In the first one shown in Table 1, we demonstrate the performance of DBOS and ext4 when 40 threads create and delete files. Since DBOS uses fully qualified names, there is a single insert to create a file instead of a directory traversal, leading to a 10× performance advantage. A similar advantage is, of course, true for deletes. As noted earlier, it is quite easy to change the implementation of DBOS files. Hence, we could move between fully qualified names and context-sensitive names if circumstances dictated. This implementation flexibility is one of the key advantages of DBOS.

Our second benchmark is an analytics operation that determines the total size of files owned by each user, counting only those that are greater than 20K in size. This is a simple SQL query in DBOS:

```
select sum(f_size) from Localized_file
  where f_owner = user and f_size >= size;
```

On the other hand, this is not a built-in operation in ext4, and must be coded in a user program using some set of tools. Since this is a simple task, we wrote a program to recursively scan the home directory for a user, computing the necessary statistic. Lines of code and running time for a simple 4000-file directory are listed in Table 2. Like other analytics operations, this one is faster in DBOS, and requires an order of magnitude less programming effort.

So far, we have exercised the localized filesystem. Now, we turn to benchmarking the DBOS parallel filesystem, where files can be distributed across multiple Supercloud nodes. We compare our implementation with the Supercloud preferred multi-node filesystem, Lustre [1]. Lustre has a block size of 1 MB but VoltDB's performance drops off for large block size. As a result, we use a block size of 8 KB in DBOS. The idea behind a parallel filesystem is to

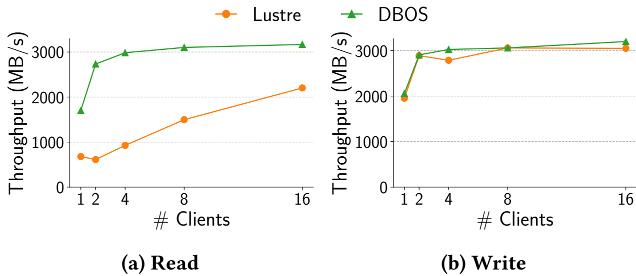


Figure 11: Throughput for 1 to 16 remote clients, co-located in a single node. Each client (a) reads and (b) writes parallel files of 8KB blocks.

deliver maximum throughput until the backplane bandwidth is exhausted. In theory, the aggregate user load on both Lustre and DBOS should increase until the Supercloud backplane bandwidth (25 Gbps between nodes) is reached.

We first test how many clients on a single node are needed to saturate the network bandwidth. VoltDB runs on 8 server nodes, each with 40 partitions, for 320 partitions in total. Client processes read and write a collection of 4096 128 MByte files, belonging to 128 different users, for a total database size of 512 GBytes. Each client process runs the following loop:

```
while(true) {
  Open a random file
  Read/Write 128 MBytes in parallel
  Close the file
}
```

In Figure 11, we see that both DBOS and Lustre can quickly saturate Supercloud's 25 Gbps network with writes. However, DBOS requires only 4 client processes to saturate the network with reads, while Lustre can only reach 70% of the network's capacity with 16 workers. This behavior might be associated with some documented issues that affect Lustre's read operations' parallelism [5].

Then, we evaluate our parallel filesystem's scalability. In this experiment, we scan across the number of nodes running VoltDB and test the maximum throughput the filesystem can achieve. We run clients in the same set of nodes as VoltDB servers. The dataset consists of 1280 128MByte files, belonging to 320 users, for a total size of 160GBytes. Files are partitioned on block number, so each file is distributed over all nodes.

In Figure 12, we plot the throughput achieved by our filesystem over the number of nodes VoltDB is deployed on. We observe a throughput drop when we move from one to two nodes. In the single-node case, all transactions are local, while when we have two nodes, half of the transactions are remote and thus much more expensive. However, we observe that after paying for that initial remote penalty the DBOS parallel filesystem's performance scales almost linearly with the number of VoltDB nodes used.

5 RELATED WORK

Using declarative interfaces and DBMS concepts in system software is not new. In the OS world, filesystem checkers often use constraints specified in a declarative language [29, 35]. More broadly, tools like OSQuery [6] provide a high-performance declarative

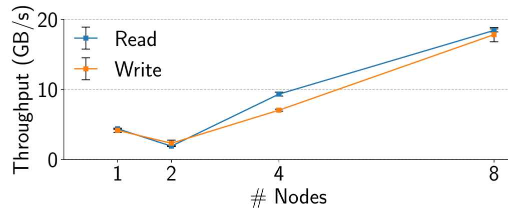


Figure 12: Throughput scalability with increasing DB nodes.

interface to OS data for easier analytics and monitoring. In a distributed setting, Cloudburst [46] and Anna [52] also build on DBMS technology. Similar ideas have been proposed in Tabularosa [33]. Declarative programming and DBMSs have already been used in some of our target applications: DCM [49] proposes a cluster manager whose behavior is specified declaratively, while HopFS [37] proposes a distributed filesystem whose metadata is stored in a NewSQL DBMS. However, none of these efforts proposed a radically new OS stack with a DBOS-style DBMS at the bottom.

In the programming languages community, there has been considerable work on declarative languages, dating back to APL [26] and Prolog [24]. More recent efforts along these lines include Bloom [18] and Boom [17]. Recently, many high-performance declarative DBMSs have been proposed, including H-Store [3] and its successor VoltDB [8]. Our contribution is to argue for a new OS stack, not a particular programming language.

Lastly, "serverless computing", or Function as a Service (FaaS), has been widely suggested as a new programming model for user programs [28, 30, 31, 38, 50]. It enables massive parallelism and flexibility in developing and deploying cloud applications, by executing stateless functions operating on externally stored state.

6 CONCLUSION

In this paper, we have shown the feasibility of layering a cluster operating system on a high performance distributed DBMS. We provided experiments showing that such an architecture can provide basic OS services with performance competitive with current solutions. We also showed anecdotally that services can be written in SQL with dramatically less effort than in a modern general-purpose language. Moreover, all OS state resides in the DBMS and can be flexibly queried for monitoring and analytics. We are presently at work on a DBOS-wood prototype which will include a serverless environment and end-to-end applications built on top of it.

ACKNOWLEDGMENTS

This material is based upon work supported by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001, National Science Foundation CCF-1533644, and United States Air Force Research Laboratory Cooperative Agreement Number FA8750-19-2-1000. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Assistant Secretary of Defense for Research and Engineering, the National Science Foundation, or the United States Air Force. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation.

REFERENCES

[1] 2003. Lustre. Retrieved September 21, 2021 from https://www.lustre.org/
[2] 2015. gRPC: A high performance, open source universal RPC framework . Retrieved September 21, 2021 from https://grpc.io/
[3] 2015. H-Store. Retrieved September 21, 2021 from https://db.cs.cmu.edu/projects/h-store/
[4] 2016. Ext4 Wiki. Retrieved September 21, 2021 from https://ext4.wiki.kernel.org/index.php/Main_Page
[5] 2018. Lustre Mailing List. Retrieved September 21, 2021 from http://lists.lustre.org/pipermail/lustre-discuss-lustre.org/2018-March/015406.html
[6] 2019. OSQuery. Retrieved September 21, 2021 from https://osquery.io/
[7] 2020. Matei Zaharia. Personal Communication.
[8] 2020. VoltDB. Retrieved September 21, 2021 from https://www.voltdb.com/
[9] 2021. Amazon Lambda. Retrieved September 21, 2021 from https://aws.amazon.com/lambda/
[10] 2021. Envoy Proxy. Retrieved September 21, 2021 from https://www.envoyproxy.io/
[11] 2021. Kubernetes Single Resource Api. Retrieved September 21, 2021 from https://kubernetes.io/docs/reference/using-api/api-concepts/#single-resource-api
[12] 2021. MIT Supercloud. Retrieved September 21, 2021 from https://supercloud.mit.edu/
[13] 2021. SAP Hana. Retrieved September 21, 2021 from https://www.sap.com/products/hana.html
[14] 2021. Scaling in the Linux Networking Stack. Retrieved September 21, 2021 from https://www.kernel.org/doc/html/latest/networking/scaling.html
[15] 2021. SingleStore. Retrieved September 21, 2021 from https://www.singlestore.com/
[16] 2021. VoltDB Stored Procedures. Retrieved September 21, 2021 from https://docs.voltdb.com/tutorial/Part5.php
[17] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. 2010. Boom Analytics: Exploring Data-Centric, Declarative Programming for the Cloud. In *Proceedings of the 5th European Conference on Computer Systems* (Paris, France) *(EuroSys '10)*. Association for Computing Machinery, New York, NY, USA, 223–236. https://doi.org/10.1145/1755913.1755937
[18] Peter Alvaro, Neil Conway, Joe Hellerstein, and William Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach. *CIDR 2011 - 5th Biennial Conference on Innovative Data Systems Research, Conference Proceedings*, 249–260.
[19] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 49–65. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay
[20] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 1–16.
[21] Chansup Byun, Jeremy Kepner, William Arcand, David Bestor, Bill Bergeron, Vijay Gadepally, Michael Houle, Matthew Hubbell, Michael Jones, Andrew Kirby, et al. 2020. Best of Both Worlds: High Performance Interactive and Batch Launching. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.
[22] Chansup Byun, Jeremy Kepner, William Arcand, David Bestor, William Bergeron, Matthew Hubbell, Vijay Gadepally, Michael Houle, Michael Jones, Anne Klein, et al. 2019. Optimizing Xeon Phi for Interactive Data Analysis. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.
[23] Michael Cafarella, David DeWitt, Vijay Gadepally, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, and Matei Zaharia. 2020. DBOS: A Proposal for a Data-Centric Operating System. *arXiv preprint arXiv:2007.11112* (2020).
[24] Alain Colmerauer and Philippe Roussel. 1996. The Birth of Prolog. In *History of programming languages—II*. 331–367.
[25] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1243–1254.
[26] Adin D. Falkoff and Kenneth E. Iverson. 1973. The Design of APL. *IBM Journal of Research and Development* 17, 4 (1973), 324–334.
[27] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. 2007. X-Trace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*. USENIX Association, Cambridge, MA. https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework
[28] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 363–376.
[29] Haryadi Gunawi, Abhishek Rajimwale, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2008. SQCK: A Declarative File System Checker. 131–146.
[30] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless Computing: One Step Forward, Two Steps Back. *arXiv preprint arXiv:1812.03651* (2018).
[31] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv preprint arXiv:1902.03383* (2019).
[32] Michael Jones, Jeremy Kepner, Bradley Orchard, Albert Reuther, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Vijay Gadepally, Michael Houle, et al. 2018. Interactive launch of 16,000 microsoft windows instances on a supercomputer. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–6.
[33] Jeremy Kepner, Ron Brightwell, Alan Edelman, Vijay Gadepally, Hayden Jananthan, Michael Jones, Sam Madden, Peter Michaleas, Hamed Okhravi, Kevin Pedretti, et al. 2018. Tabularosa: Tabular operating system architecture for massively parallel heterogeneous compute engines. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–8.
[34] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 1, 16 pages. https://doi.org/10.1145/2901318.2901326
[35] Marshall Kirk McKusick, Willian N Joy, Samuel J Leffler, and Robert S Fabry. 1986. Fsck- The UNIX† File System Check Program. *Unix System Manager's Manual-4.3 BSD Virtual VAX-11 Version* (1986).
[36] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. 2016. Understanding Manycore Scalability of File Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 71–85. https://www.usenix.org/conference/atc16/technical-sessions/presentation/min
[37] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. 2017. HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 89–104. https://www.usenix.org/conference/fast17/technical-sessions/presentation/niazi
[38] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 193–206.
[39] Björn Rabenstein and Julius Volz. 2015. Prometheus: A Next-Generation Monitoring System (Talk). USENIX Association, Dublin.
[40] Gladys Rama. 2020. Report: AWS Lambda Popular Among Enterprises, Container Users. https://awsinsider.net/articles/2020/02/04/aws-lambda-usage-profile.aspx.
[41] Albert Reuther, Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Michael Jones, Peter Michaleas, Andrew Prout, Antonio Rosa, and et al. 2018. Scalable system scheduling for HPC and big data. *J. Parallel and Distrib. Comput.* 111 (Jan 2018), 76–92. https://doi.org/10.1016/j.jpdc.2017.06.009
[42] Albert Reuther, Jeremy Kepner, Chansup Byun, Siddharth Samsi, William Arcand, David Bestor, Bill Bergeron, Vijay Gadepally, Michael Houle, Matthew Hubbell, et al. 2018. Interactive supercomputing on 40,000 cores for machine learning and data analysis. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–6.
[43] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. 2020. Survey of Machine Learning Accelerators. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–12.
[44] D. M. Ritchie and K. Thompson. 1974. The Unix Time-Sharing System. *Commun. ACM* 17 (1974), 365–375.
[45] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc. https://research.google.com/archive/papers/dapper-2010-1.pdf
[46] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *arXiv preprint arXiv:2001.04592* (2020).
[47] Michael Stonebraker. 1981. Operating System Support for Database Management. *Commun. ACM* 24, 7 (July 1981), 412–418. https://doi.org/10.1145/358699.358703
[48] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of POSTGRES. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data (SIGMOD '86)*. Association for Computing Machinery, Washington, D.C., USA, 340–355. https://doi.org/10.1145/16894.16888
[49] Lalith Suresh, João Loff, Faria Kalim, Sangeetha Abdu Jyothi, Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, and Michael Gasch. 2020. Building Scalable and Flexible Cluster Managers Using Declarative Programming.

In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 827–844. https://www.usenix.org/conference/osdi20/presentation/suresh

[50] Mike Wawrzoniak, Ingo Müller, Rodrigo Fraga Barcelos Paulus Bruno, and Gustavo Alonso. 2021. Boxer: Data Analytics on Network-enabled Serverless Platforms. In *11th Annual Conference on Innovative Data Systems Research (CIDR'21)*.

[51] Rebecca Wild, Matthew Hubbell, and Jeremy Kepner. 2019. Optimizing the Visualization Pipeline of a 3-D Monitoring and Management System. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–5.

[52] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. 2018. Eliminating Boundaries in Cloud Storage with Anna. *arXiv preprint arXiv:1809.00089* (2018).