OSDB: Exposing the Operating System's Inner Database

Robert Soulé
Yale University
USA

George Neville-Neil
Yale University
USA

Stelios Kasouridis
Yale University
USA

Alex Yuan
Yale University
USA

Avi Silberschatz
Yale University
USA

Peter Alvaro
UCSC
USA

ABSTRACT

Operating systems must provide functionality that closely resembles that of a data management system, but existing query mechanisms are ad-hoc and idiosyncratic. To address this problem, we argue for the adoption of a relational interface to the operating system kernel. While prior work has made similar proposals, our approach is unique in that it allows for incremental adoption over an existing, production-ready operating system. In this paper, we present progress on a prototype system called OSDB that embodies the incremental approach and discuss key aspects of the design, including the data model and concurrency control mechanisms. We present four example use cases: a network usage monitor, a load balancer, file system checker, and network debugging session, as well as experiments that demonstrate the low overhead for our approach.

CCS CONCEPTS

• **Software and its engineering** → **Operating systems**; • **Information systems** → *Entity relationship models*; **Data access methods**.

KEYWORDS

Operating Systems, Databases, Structured Query Language (SQL)

1 INTRODUCTION

Operating systems (OS) are conventionally defined as the software that manages hardware. Unfortunately, this definition is incomplete, as it ignores the fact that modern OSs are open systems with people attached to them. To say that an OS is software that virtualizes and abstracts hardware in order to present it to users would be more accurate, but this definition remains somewhat idealized. In practice, OSs do have a "bottom layer" dedicated to management of hardware, namely compute, memory, and peripherals. They also have an "upper layer" providing a collection of mechanisms that allow users—be they administrators, application developers, or end users—to observe and manage system state. Users engage with this top layer in myriad ways; to diagnose system-wide problems, debug individual programs, identify performance bottlenecks, and ultimately define system policies for concerns as diverse as scheduling, resource allocation, cache management, and storage.

Viewed in this way, the upper layer of an OS closely resembles a data management system! But it is a very ad-hoc DBMS with a query mechanism that leaves much to be desired. If the "query" is performed from within kernel space, then the operating system may provide some APIs, but, more often than not, developers will need to write code to traverse global data structures to find the desired information. This requires that a developer understand the relevant data structures and the locking discipline to access the data correctly. On the other hand, performed from user space, a modern OS provides a number of mechanisms for observing kernel state (e.g., system calls and command-line utilities, etc.) and a handful of mechanism for managing them (e.g., `ioctl`, `/etc/password`). However, using these mechanisms can be awkward. Command line utilities provide a non-uniform query language in which the command names and meaning of the flags have to be learned on a case-by-case basis. Moreover, UNIX/Linux's file abstraction, in which everything is an unstructured stream of bytes, means that every application has to enforce an implicit schema that is lost when the structured kernel data is mapped to a stream of bytes.

OS data model and query mechanisms have evolved organically over time, spurred by developer needs and attempts at standardization (e.g., POSIX). In contrast, the database community has taken a much more intentional approach in developing the relational model. Based formally on first-order predicate logic, the relational model organizes data into a uniform and simple data structure (i.e., relations) and offers both an algebra for specifying queries and a method to express integrity constraints. This approach offers a number of benefits; not least of which is that it allows users to manipulate the data in high-level declarative languages that naturally allow for the composition of analytical operations.

Much ink has been spilled [1, 2, 7, 9, 13, 14] demonstrating how database ideas, creatively applied, can significantly simplify systems concerns from networking to cluster management to single-host operating systems. Existing tools such as osquery [12] mirror important OS state in a user-space database, providing coarse-grained, periodic mechanisms for observing and changing the state. Overlaying a DB on the OS in this way is non-invasive and can be adopted incrementally, but offers only weak semantics including stale reads and non-atomic updates to kernel state. At the other end of the design spectrum are "boil the ocean" approaches such as DBOS [13], which propose to entirely replace the OS with a query processor. While the capabilities of such an approach are theoretically limitless, the path to adoption is less clear.

We advocate for an incremental infiltration of the OS that we call the "Ship of Theseus" approach. We ported an existing database, SQLite, to run inside the FreeBSD and Linux kernels. Once this effort was complete, we then leveraged the database's support for foreign data sources, implementing an interface that allows the database engine to traverse existing kernel data structures as if they were tables. This permitted us to write SQL queries directly over kernel data structures, providing the benefits of a relational interface and rich query language in a pay-as-you-go fashion.

In this paper, we present progress on our prototype system, which we call OSDB. We describe how we mapped OS data structures to relations and used kernel memory addresses as join keys. We detail the concurrency control mechanisms we implemented to ensure consistent reads and atomic modifications of those structures, while prioritizing noninterference with other kernel operations over query performance. We also report on our experiences using OSDB through four case studies: monitoring system state; enforcing system policies (e.g., for load balancing among cores); identifying and fixing correctness issues in the file system; and diagnosing a performance issue in NFS. Common among these disparate use cases is the ease of use of a query language, and most importantly the use of JOIN rather than pipes as the basis for combining information.

2 EXAMPLE AND MOTIVATION

To make the problems of kernel state management in Section 1 concrete, we start with a small example that is likely familiar to anyone who has done network programming. Imagine a user would like to discover which process is currently listening on a particular network port, perhaps to kill the process and free the port. On a UNIX/Linux system, the user would probably run the `ps` command to see the list of processes, and the `netstat` command to see the status of various ports, and with that information, try to sleuth out the offending process. This example is small, but it helps illustrate several important challenges faced by users when trying to gather and take action on information in current operating systems. We enumerate these challenges beginning with the trivial and progressing to the fundamental:

(1) *Non-standard interfaces.* How does the user know which commands to run? If the user is not an experienced UNIX user, how would they know that the program to list running programs is called `ps`, which is short for "process status"? After all, the equivalent program on Windows is called `tasklist`. Even if you are an experienced UNIX user, finding the appropriate commands can be confusing. Should you use `who` or `w` to see who is logged into a server? Which command tells you the status of the print queue? Man pages can tell you how to use a command if you already know the command, but if you don't, a user can guess at command names or consult Stack Overflow. Contrast this experience with that of the physical world. For example, someone who knows how to drive can drive *any* car without instruction, regardless if it is a Ferrari or a Kia. The car's user interface is standardized.

(2) *Platform-specific implementations.* Even for the same command, it can have different usage patterns for different tasks, different implementations, or versions of the operating system. To see a list of print jobs, a user can use `lpstat`, but if the jobs are on a shared printer, they must use `lpstat -p`. For our simple example, on a system using standard syntax, we would invoke the process status (`ps`) command with the `-e` flag to see all processes. But, if they were on a system using BSD syntax, they would need to invoke the command with the `-a` flag to see all the processes.

(3) *Loss of structure.* Let's expand our example a bit, and imagine that the user runs `ps`, but gets too many results, so they want to filter is in some way, e.g., to get only recent processes. Embracing the "everything is just a stream of bytes" philosophy of UNIX, the user might use a pipe and `awk`, to write something like `ps -a | awk '$3 > "0:00.10"'`. At the risk of starting a flame war with `awk` devotees, we argue that the syntax for this command is not obvious. But, the more serious problem is that the *structured* output of the `ps` command is turned into an *unstructured byte stream*, which then must be *parsed back* into a structured representation that allows for filtering on the desired predicate.

(4) *No join.* Finally, and most importantly, there is no way to naturally compose the queries. In our example, there is no way to correlate the information from `ps` with the information from `netstat`. Even if there were some key that related the two data sets, we are forced to use tools like `cut`, `head`, `tail`, and `awk` to manipulate the byte streams. Converting to unstructured streams and composing with pipes always loses the implicit structure and type information of the kernel state.

All of these challenges can be addressed by representing the operating system kernel state relationally. SQL is a standardized language based on firm formal foundations. To learn what tables exist and their schemas, one can use the `SHOW TABLES` or equivalent commands. If everything is a relation (instead of everything is an unstructured byte stream) then we preserve the inherent structure of the data across composition. And, most importantly, we can use the power of "joins" to compose the data naturally. We acknowledge that a relational representation is not a panacea. SQL is not without its critics [3] and there are some variations in syntax across vendor offerings. However, our experiences with the reported case studies have been that the relational interface simplifies state monitoring and management.

To see the list of running processes and the associated TCP five-tuple using OSDB, a user can use the following query:

```sql
SELECT p.pid, p.name,
       t.faddr, t.fport, t.laddr, t.lport, t.t_state
FROM procs AS p, files AS f, tcps AS t
WHERE f.f_addr = t.inp_addr AND f.pid = p.pid
```

Intuitively, the query joins a table that exposes process information with a table that contains the open files and a table that includes the socket information for those files that are of a socket type. Note that we didn't need to add any code to the kernel to achieve this: we are combining data between already existing data structures.

There are a few notable features of this query that merit further discussion. In particular: how do we model the OS state as tables? Given a set of tables, what constraints (e.g., primary and foreign keys) are appropriate to define? How do we ensure consistency when querying existing data structures? We discuss these details, and more, in the next section.

3 DESIGN

There are several approaches that can be taken to make operating system state more visible and manageable. Two contemporary systems share similar motivations to OSDB, but occupy different points in the design space. osquery [12] extracts unstructured data from the kernel and imports it into a user-space database. While this results in a richer user interface, it only supports read operations and fails to address issues of data consistency, or the ability to react to changing conditions within the kernel in real time. and cannot present a transactionally-consistent view of kernel state. DBOS [13] aims to build a brand new operating system around a query processor. This green field approach could be impractical because the effort to build such an operating system from scratch requires a significant development effort as well as a long lead time before benefits can be shown.

3.1 The OSDB Incremental Approach

With OSDB, we explore an alternative approach. We ported an existing database, SQLite, to run within the FreeBSD and Linux kernels. We then leveraged the database's support for foreign data sources, implementing an interface that allows the database engine to traverse existing kernel data structures as if they were tables. This permits us to write SQL queries directly over kernel data structures.

This is an incremental, "Ship of Theseus" approach to moving the operating system state into a relational representation: to provide, entity-by-entity, a relational interface to existing operating system structures without having to re-write the entire operating system. Although we currently only traverse existing structures, one could imagine a more ambitious step of replacing the structures we currently abstract with a table instance directly. As part of future research we will convert C structures to tables where that conversion improves safety and/or multi-processing performance, which is where such conversions would show the clearest benefits.

3.2 Data Model

In presenting a relational interface to operating system state, we face a very large data modeling task: identifying every logical entity in the operating system as well as the relationships between them. We are aided in these efforts both by our incremental approach—which means that we don't need to model the entire operating system in order to start seeing the benefits of OSDB—and a couple of key observations about the operating system code. First, operating system developers have already naturally modularized their code into the relevant entities, i.e., as C structures. Second, the majority of the data structures in an operating system kernel are lists of C structures. In FreeBSD nearly every complex data structure depends on macros from a single header file (`queue.h`) which provides singly and doubly linked lists, and tail queues.

Lists of structures map well onto the relational model, as every list can be a table, every element (struct) in the list is a tuple (row), and every data member of a structure is a field (column). Figure 1 shows the relationships among four of the tables that are exposed from `OSDB`. An entry in the `procs` table has pointers which act as foreign keys. For brevity, in this diagram we only show the relationship between the `proc` structure and its associated `files` and `threads` as well as the back pointer from a thread to its process.

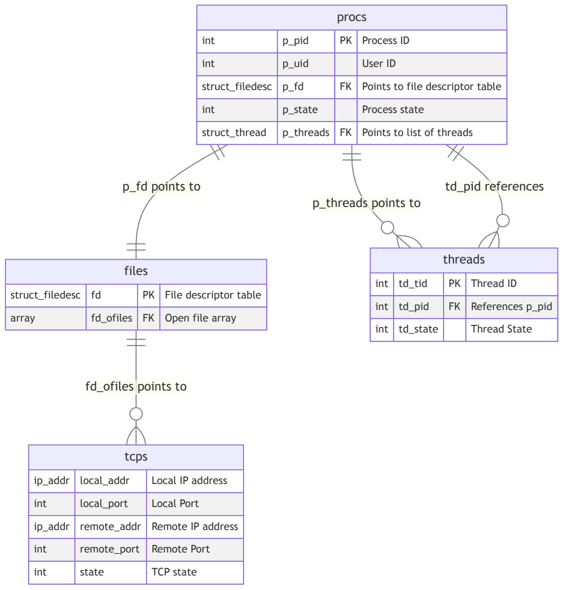

procs
int | p_pid | PK | Process ID
int | p_uid | | User ID
struct_filedesc | p_fd | FK | Points to file descriptor table
int | p_state | | Process state
struct_thread | p_threads | FK | Points to list of threads
p_fd points to
p_threads points to
td_pid references
files
struct_filedesc | fd | PK | File descriptor table
array | fd_ofiles | FK | Open file array
threads
int | td_tid | PK | Thread ID
int | td_pid | FK | References p_pid
int | td_state | | Thread State
fd_ofiles points to
tcps
ip_addr | local_addr | Local IP address
int | local_port | Local Port
ip_addr | remote_addr | Remote IP address
int | remote_port | Remote Port
int | state | TCP state

Figure 1: An entity relationship diagram for four of the OSDB tables, modeling processes, threads, file descriptors, and sockets, as well as their constraints. Note that `p_fd` and `p_threads` are pointers to a list of structs in the kernel. In the case of the relationship between processes and threads, these pointers are redundant (because the thread struct has a natural foreign key back to processes via `td_pid`) and used for fast traversals. This is not the case for the relationship between processes and their list of open files: no natural foreign key exists, so we infer the relationship via the `p_fd` pointer.

Sockets for TCP connections are also in the file descriptor table and we see this relationship at the very bottom of the diagram. Note that the TCP connection does not have a relationship back to its process, but through the power of joins we are still able to relate these two entities in our system.

Foreign Key Constraints. Unfortunately, with the data modeling approach described above, there is usually not a natural data item that serves the role of the foreign key. For example, each process structure in kernel has a `pid` that can serve as a primary key, but each thread structure does not include the process that owns the thread. Rather, relationships between these in-memory structures are established through pointers, e.g., an instance of a process structure would include a list of pointers to thread structures. That is to say, the arrows go the wrong way [10].

Therefore, to establish foreign key relationships, we include pointers as a field in many table representations of OS structures. We note that the pointers in an operating system kernel are unique because the kernel program executes in a single address space, making them a suitable choice for foreign keys. Once a kernel data structure is allocated, it is never moved and its address never changes.

We see the use of a pointer address used as a key in the example in Section 2, where (`f.f_addr = t.inp_addr`). Here, the query says that the address of the file must be the same as the address in the IPv4 structure (`inp_addr`).

3.3 Query Semantics

OSDB users provide SQL queries to observe and optionally modify kernel state. For usability, we wish to provide familiar semantics for query writers; for performance and correctness, we wish to utilize the synchronization mechanisms already available in the OS kernel whenever possible. We discuss *observation* (i.e., read-only queries) first, then proceed to queries that modify state.

Read-only queries. Whenever a query executes, OSDB acquires the required kernel locks in a two-phase fashion, and makes a copy of all kernel data structures in one go. We refer to the copy of the data as a snapshot. After copying, the locks are released and the query executes on the snapshot. Each snapshot is assigned a timestamp when it is created. As we will discuss below, OSDB maintains a history of snapshots, allowing for historical analytics.

This approach is feasible because kernel structures tend not to be overly large, and copying does not impose undue strain on the system as a whole. This approach also means that the system does not have to hold the locks during query execution, which can be arbitrarily long. Our experiments in Section 7 demonstrate that this approach is stable, i.e., free of kernel panics, and performant.

Each snapshot is a transactionally-consistent view of the kernel state; the sequence of snapshots defines a straightforward time series over that state. Snapshots are made visible using transactions, hence any read-only query over a snapshot (or across snapshots) is serializable. Existing tools that query the operating system state from user-space cannot ensure the same strong consistency guarantees as OSDB. A pipeline of cooperating processes on a UNIX system does not operate in parallel, but in series, meaning that the data that arrives at the last program in pipeline may be out of data by the time it arrives. It is precisely because we interact directly with the existing kernel data structures using a two-phase locking protocol that we can make atomic observation of kernel state. In contrast, existing tools, such as a pipeline of user space processes, can not ensure the same consistency guarantee.

Snapshot Creation. OSDB currently works in either of two modes. A snapshot can be created every time a query is submitted to the database, or at periodic intervals according to a user-supplied parameter. The latter approach effectively implements sampling of the kernel state. Sampling allows users to see how system state evolves over time, for example, to correlate file operations, network operations and CPU utilization.

Historical Analytics and Memory Management. Each time a snapshot is created, OSDB inserts the snapshot into a circular queue of a configurable size, depending on the amount of available memory in the system and the needs of the user. If the queue is full when a new snapshot is created, the oldest snapshot not currently in use is garbage collected. By "not in use", we mean that no query is currently running against that snapshot, which OSDB tracks using a basic reference counting scheme. This design ensures that the amount of memory used by OSDB is bounded.

In addition to the historical data, we have defined views that show the data from only the most recent snapshot. Querying the view of the process table, for example, gives a super set of the information returned by the `ps(1)` command. Querying the history allows for time series analytics. For example, one can query to see how the RTT estimate for TCP connections changes over time.

Read/Write Operations. Up until this point, our discussion has mainly focused on read operations. However, a powerful aspect of our design is being able to perform insert, update, and delete operations as well.

Here, too, we are taking an incremental approach. All relations are initially defined as read-only; hooks to implement insert, update, and delete operations are created on a table-by-table basis as we model the kernel structures and their constraints. For example, OSDB currently supports delete on the process table, since this can be mapped in a straightforward way to existing kernel APIs that perform the appropriate cleanup (i.e., to delete a process is to *kill* it), as well as update on certain columns (e.g., `lastcpu`, which can be used to migrate a process between cores), but it has not yet defined insert for processes.

The other key challenge in implementing a write path for OSDB is concurrency control. The snapshotting system described earlier makes read-only observations of system state simple and unobtrusive, but it does not automatically lend itself to a read/write synchronization strategy, since kernel-native locks are released before any query processing begins.

We opted again for a simple design that reuses existing mechanisms and prefers optimism (even at the cost of aborts) over interference with critical kernel functions. By default, a read/write query is always executed over a fresh snapshot, which it generates in its first phase just as a read-only query does. Alongside the snapshot, OSDB generates a *digest* via a hash function. In the second phase—again, just as in read-only execution–the query executes within SQLite. This time, however, the query produces a set of *effects* to be carried out atomically if the transaction should commit.

In phase 3, OSDB again queries the kernel state to obtain a digest—this time, it does not immediately release its locks. If the digest matches the digest taken at snapshot time, the query is allowed to commit, and its updates are applied by invoking the relevant kernel APIs. Otherwise the query is re-executed; either way the locks are released.

Some kernel state (e.g., counters that track the resource usage of threads or the number of transmitted packets associated with a network socket) are volatile enough that they are likely to always change between the first and third phase of transaction execution described above. One way to address this issue would be to omit all volatile state from snapshots. Unfortunately, this would also prevent read-only queries from referencing this state, limiting the system's observability. Instead, we chose to handle volatile kernel state as a special case. As part of our data modeling, certain fields of kernel structures are flagged as volatile. These fields are included in the snapshots, but their values are not hashed, and read/write queries are forbidden from referencing them. This default semantics provides an intuitive query-the-present model with serializable executions. Moreover, it also readily supports transactionally reading from an old snapshot; similarly, the effects will be installed only if the digest has not changed, but these queries will become increasingly unlikely to succeed. We are also exploring a snapshot isolated semantics in which we compute digests of sub-snapshots, performing commit-time write checks only on the intersection of sub-snapshots.

4 APPLICATION PROGRAMMING INTERFACE

For user-space programs to interact with OSDB, we extended the FreeBSD system call interface with one object and 14 system calls. The main interface to OSDB is as follows:

- `osdb_statement`: an object that represents a prepared statement that is ready to be evaluated.
- `osdb_prepare()`: compile SQL text into a format that will execute the query. This method returns an `osdb_statement`.
- `osdb_step()`: advance a statement to the next row.
- `osdb_column_int()`, `osdb_column_text()`, etc.: Return column values in the current row in the specified type. For brevity, we do not enumerated all of the type-specific column commands.
- `osdb_finalize()`: destructor for an `osdb_statement`.

Note that the user must pass the `osdb_statement` object instance to each invocation of `step`, `column`, and `finalize`, similar to how a file descriptor object is used in the `open`, `read`, `write` and `close` system call API that is familiar to every UNIX developer, and a cursor interface familiar to database users.

We have provided both command-line and library-based APIs. The command line tool, `osdb_query`, takes SQL commands and passes them into the OSDB kernel module. Alternatively, users may write their own code using the system calls we provide.

On Linux, we took a different approach. The Linux kernel module exposes a new device and user space programs access the in-kernel version of SQLite via an `ioctl()` call.

5 IMPLEMENTATION

We have implemented OSDB twice, first embedding into the FreeBSD kernel and later into Linux. In the section, we describe the required steps for such an implementation. The four key steps include: making modest changes to the OS kernel; porting SQLite to the target kernel; creating a loadable kernel-module; and following a process for incrementally exposing kernel state as tables for observation and management.

5.1 OS Kernel Changes

The operating system must be augmented with an interface to access OSDB. On FreeBSD, we added several system calls that provide the API to user space programs. These were described in detail in Section 4. Note that we only need to declare the stubs of the system calls in the kernel. The implementation of those methods are provided in the kernel module, described below. This adds 45 lines of code to the FreeBSD kernel in a single file. The Linux port followed a different implementation, exposing an `ioctl` to user space code that interacts with the OSDB module. Like FreeBSD, the Linux port makes minimal changes in the main line of the operating system and provides nearly all of its real code via a kernel module.

5.2 Porting SQLite to the Kernel

SQLite is an embedded database designed with portability in mind. Many extensibility-supporting features are already provided via C macro-based configuration options. Porting the database to run outside user space we had to deal with three issues. First, we needed to provide SQLite with a mechanism for allocating memory that "played nice" with the OS kernel. Second, we needed to provide support for floating point operations, which are typically disabled in the kernel. Third, we needed to translate the logical lock requests that occur as part of SQLite's concurrency control into methods that operate over kernel latches, following the synchronization discipline already in place in the OS.

Allocation. In the operating system, kernel code can choose to block or not block waiting for memory, which is not an option in user space libraries. We took the conservative approach of making SQLite wait for memory from the kernel before it could proceed. As SQLite is not a time critical component of the overall system it was acceptable to wait for memory if it was not immediately available.

Floating Point Support. One significant challenge in porting SQLite into the kernel is the latter's use of floating point operations. Operating systems traditionally do not allow floating point operations as this would require the saving of extra registers on each context switch, increasing overhead and reducing performance. Both FreeBSD, and our second target, Linux, have the ability to allow floating point operations in particular sections of the code that need them. Bracketing routines are used to tell the kernel that on a context switch it must preserve the floating point registers. Our FreeBSD implementation provides new system calls for access to the SQLite library, and these system calls provide a natural location for the floating point save and restore routines. On Linux we have wrapped the specific uses of `double` with the floating point bracket routines.

Locking. Translating row-level read/write locks into appropriate latching protocols for kernel data structures seems like a daunting task. Luckily, the extensibility features of SQLite significantly simplified the ports to both FreeBSD and Linux. Although this effort will differ for every OS target, it can be completely factored into macro configuration and implementing wrapper functions.

The FreeBSD and Linux ports were quite similar. Although there were differences between the kernel services (e.g., `malloc` vs `kmalloc`), the internal APIs of the two operating systems were often nearly identical. This is unsurprising, as they both aim to provide similar functionality, namely, a POSIX-like programming environment.

5.3 OSDB Module

OSDB is provided as a loadable kernel module on both FreeBSD and Linux. The kernel module is responsible for: instantiating the OSDB database; registering the virtual tables that expose kernel data structures (described in more detail below); orchestrating the two-phase locking protocol (i.e., acquiring/releasing locks and invoking `snapshot` methods to copy data), and exposing the OSDB API.

Although there are platform-specific differences between the FreeBSD and Linux versions (e.g., the kernel module APIs are different, and the OSDB API is provided via system call or `ioctl`, respectively), much of the code is re-usable across operating systems.

Our modification required very few lines of code. On FreeBSD the entire kernel module has only 3670 lines of non-comment code, of which 2318 lines are used to implement our ten virtual tables.

5.4 Exposing Kernel Data Structures

SQLite includes support for reading from or writing to data sources other than the database files via a mechanism called *virtual tables* [19]. This mechanism allows the SQLite query engine to treat in-memory data structures as if they were standard tables or views. To create a virtual table, a developer must implement an API that includes set of methods used to open/close the database, open/close an iterator to scan, advance the iterator, and access values stored in a particular columns. A user may optionally provide an `update` method to modify the data.

OSDB includes code that provides a generic table data structure (i.e., a list of union types) and implementations of the SQLite virtual table API targeting that data structure. The table implementation includes functionality to store metadata (i.e., column names and types) as well as the actual data.

To expose a kernel data structure to OSDB, a developer must implement three methods. First, they must implement a `snapshot` method that that walks kernel data structure in place and copies it into the generic table representation. Then, they must implement `lock` and `unlock` routines that acquire and release the locks associated with the kernel data structure during the two-phase locking protocol.

As mentioned above, a developer may additionally provide an update method to insert, update or delete data. The update method can invoke arbitrary C code. Our current prototype only supports update operations that map to existing kernel APIs, e.g., deleting from the process table corresponds to calling the method that kills a process. This ensures that the kernel is not left in an unstable state.

6 CASE STUDIES

We now present four diverse use-cases for OSDB. The first use-case, drawn from our experience deploying and operating production systems in high performance computing environments, overrides the kernel process scheduler to reduce resource contention. The second use-case recreates a result from academic literature. The third adds new functionality to the kernel which replaces work that is currently done by hand in user-space. The fourth is a case of debugging an NFS performance problem.

Load Balancer. Figure 2 shows a plot of jitter for two competing network receivers over time. Jitter [4] is the continuous variance between best and worst network latency over a set of packets, and is a good predictor of other network performance problems such as high tail latencies and decreased overall bandwidth. Jitter often is a result of improper resource sharing. In this study, we used the `iperf3` program with UDP data to generate traffic, with client-server pairs assigned to different cores.

At the start of the experiment both receivers are placed on the same CPU core, and therefore compete for resources. We can see

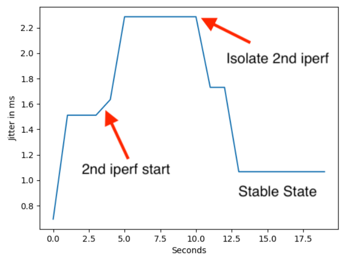


Figure 2: Jitter Improvement with Two Competing Receivers

how the jitter doubles, from 1 to 2 milliseconds as the second receiver begins to receive data. At the 10 second mark we used the following OSDB query to move one of the instance of `iperf3` to an idle core, with the effect that the jitter measurements decrease and become stable through the end of the experimental run:

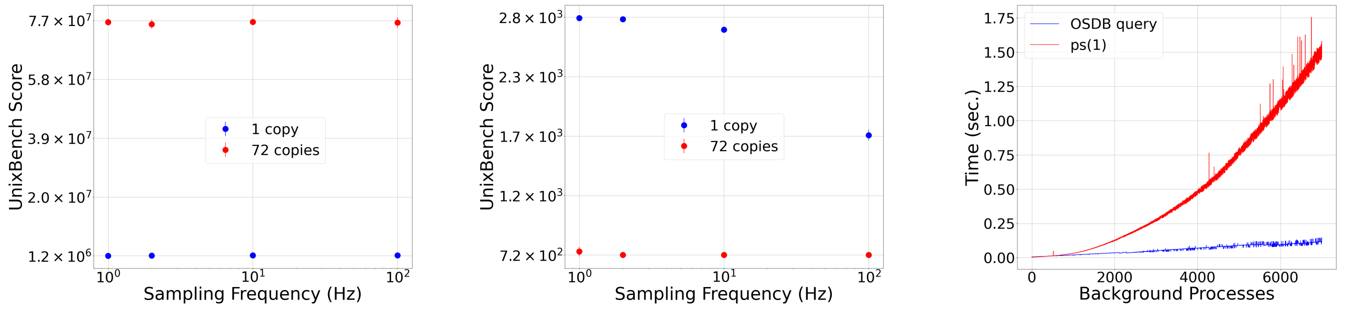
```
UPDATE all_threads SET lastcpu=(SELECT lastcpu FROM
  (SELECT min(num), lastcpu FROM
    (SELECT COUNT(DISTINCT pid) AS num, lastcpu FROM
      threads WHERE lastcpu !=-1 GROUP BY lastcpu)))
WHERE pid=(SELECT pid FROM procs WHERE name="iperf3"
     LIMIT 1)
```

The SQL statement picks the first instance of the `iperf3` program it finds in the process table and moves it to the core that currently has the least number of threads running on it. For the purposes of this demonstration, the process that we are moving is `iperf3`, allowing us to generate the time series in Figure 2. However, we could easily update this query to use more general selection criteria, e.g., the process consuming the most network bandwidth, CPU time, etc.

FSCK. SQCK [6] demonstrated that a file system checker like `e2fsck` could be written decralatively in SQL. We recreated one of their checks to demonstrate the same ability. Specifically, we check for a file system corruption in which a directory X claims to have a child directory Y, but Y claims another directory Z as its parent, where $Z \neq X$, and Z does not claim Y as its child directory. The following query tests for this corruption condition, and, if detected, fixes the file system by setting the parent of Y to be X.

```
WITH bad_entries AS
  (SELECT c.dot_inode AS child, p.dot_inode AS
       correct_parent
  FROM dirents AS p, dirents AS c
  WHERE p.entry_inode = c.dot_inode
  AND p.name != '.' AND p.name != '..'
  AND c.name = '..' AND c.entry_inode != p.dot_inode)
UPDATE all_dirents SET entry_inode =
  (SELECT correct_parent FROM bad_entries
  WHERE bad_entries.child = all_dirents.dot_inode)
  WHERE name='..' AND dot_inode
  IN (SELECT child FROM bad_entries)
```

Our system differs from SQCK in an important respect. SQCK, like `fsck`, runs offline in user-space. The OSDB version runs online in kernel space, by which we mean that it runs on the in-memory representation of the file system and writes the modifications to disk. Because we don't have a full implementation of all the repairs that SQCK and `fsck`, we have not bench-marked the performance.



(a) UnixBench context1, context switching (b) UnixBench shell16, script executions (c) ps(1) vs OSDB

Figure 3: Evaluating OSDB overhead and compared to existing tools.

Out of Network. While all operating system kernels have some form of "out of memory killer" (OOM) which protects the system from running out of RAM by killing off processes that consume too much memory, the same does not exist for programs that consume too much of the network. High performance network systems often run very close to the edge of total capacity. When applications overuse the network they can get into a state where a group of such applications must be terminated and restarted [18]. With OSDB, we implemented this "out of network" killer:

```
DELETE FROM all_procs WHERE pid=
  (SELECT t.pid FROM threads AS t
   WHERE t.pid > 1000 AND t.msgsend>10000
         AND t.timestamp > unixepoch('now')-1)
```

The query terminates any user process (pid > 1000) which has sent more than 10,000 messages in the last second.

Diagnosing a Performance Issue in NFS. Debugging network problems is a complex and tedious task in part because it involves more than one computer, but also because the sources of data to be analyzed are poorly correlated. A typical debugging session includes statistics collection on all of the hosts involved, as well as gathering a trace of all of the network packets that were transmitted and received during the operation. Multiple executions of a test are used to gather sufficient statistics and then these are visually inspected, as is the packet trace, to look for clues as to the source of the problem. While the statistics are cleared between each test execution, current tooling does not support showing how the statistics change over time.

To ease development of kernel code, we use a virtual machine with a shared, NFS, mount. Source code resides on the host laptop, but builds take place within the VM. As we started developing our kernel module, we noticed that the start of each module build would have a noticeable, 1 to 2 second pause. To track down the source of this delay we added an NFS client statistics table to OSDB and extracted the relevant data with the following query:

```
SELECT latest.count - previous.count as count_difference,
       latest.category, latest.operation, latest.type
FROM all_nfsclients latest
JOIN all_nfsclients previous
 ON latest.category = previous.category
 AND latest.operation = previous.operation
 AND latest.type = previous.type
 WHERE latest.timestamp =
   (SELECT MAX(timestamp) FROM all_nfsclients)
 AND previous.timestamp =
   (SELECT max(timestamp) FROM all_nfsclients  WHERE
        timestamp <
     (SELECT MAX(timestamp) FROM all_nfsclients))
```

The query compares statistics from the most recent snapshot to the one previous, and the snapshots are taken both before and after the build command is issued. A pair of lines immediately stand out:

```
22010, rpc, getattr, count
22010, cache, getattr, miss
```

The number of remote procedure calls for getting a file's attributes (getattr) and the number of cache misses were both large, and identical on each run, indicating that some part of the build was trying to look up a large number of files on the server that did not exist. Using OSDB we were able to diagnose the problem, which we then confirmed through discussions with the kernel developers. The FreeBSD build expects that there is a generated file corresponding to each source file of the kernel code. The build tools do a search for these files at the start of every build. On NFS, the failed search requests correspond to the cache misses, which results in the delay we experienced. In other environments, such as on fast NVME or SSD, this part of the build is not noticeable, because those types of media are much faster than NFS access.

7 EVALUATION

To evaluate OSDB, we carried out two sets of experiments. The first set quantifies the overhead that OSDB adds to OS runtime behavior. The second establishes a baseline comparison against an existing command line tool for querying kernel state, ps(1). The experiments demonstrate that OSDB is stable and adds an acceptable amount of overhead. Moreover, they show, perhaps surprisingly, that querying OS state via a database can be faster than existing command line utilities.

Test Environment. Our test environment is a Dell PowerEdge R740 server with 2 Intel Xeon Gold 6240 CPUs clocked at 2.60GHz and each comprising 18 dual hyper-thread cores for a total of 72 cores. The system contains 786G of RAM and 512G of striped secondary storage provided by ZFS. The system is running FreeBSD CURRENT (15.0) with a NODEBUG kernel. Our code is contained in a single, loadable kernel module which is loaded at run time and provides the new system calls as well as the embedded SQLite database. In all experiments, our module includes five tables: processes, threads, files, TCP connections, and UDP connections.

Overhead. To quantify the overhead that OSDB adds to OS runtime behavior, we ran UnixBench [17] while increasing the OSDB sampling rate from 1Hz to 100Hz. UnixBench includes 10 different benchmarks, each designed to test a different aspect of the operating system. Each benchmark reports a score. The score does not directly map to a key performance indicator, such as throughput or operations per second. Rather, it is a composite value that incorporates various aspects of the system's performance. A higher score indicates better performance. For brevity, we report only two results: context switching and shell script invocation. Context switching is a common measurement of operating system overhead and we found that the shell script invocation test was the one that showed the clearest amount of OSDB overhead. The results appear in Figures 3a and 3b, respectively. UnixBench runs each benchmark on a single core (blue), and on all of the available cores (red), to measure uni-processor and multi-processor performance. For `context1`, we see that the context switching overhead is unchanged as we increase the sampling rate, even though the performance is different between the uni-processor and multi-processor cases. For the `shell16` benchmark, we see that the single core performance is affected by our test query as the rate increases, but that for the multi-processor benchmark performance overall is not affected by our single query. In all of the benchmarks that we have run we have not seen any indication that OSDB overhead presents a barrier to adoption.

Comparison to Existing Tools. To compare the performance of OSDB against an existing command line tool for querying kernel state, we measured the run time of the `ps(1)` command to an OSDB query for an increasing number of background processes. Both commands returned the same data: pid, uid, name, group id, tty, state and parent pid, in the same order (i.e., sorted by pid). Each command was executed in a loop 10,000 times with the output redirected to `/dev/null`. We report the total time for all 10,000 invocations. We note that the background processes were all suspended, so although they increase the number of processes reported by the commands, they do not add additional load on the system.

Figure 3c shows the results, which we found surprising. As we discussed in Section 4, the OSDB API imposes a significant number of user/kernel boundary crossings, so we expected that OSDB would be slower than `ps(1)`. However, as we increased the number of processes, the running time of OSDB grows at a much slower rate than `ps(1)`. Profiling `ps(1)` reveals that most of the time in the program is spent in the use of `qsort(3)`. In hindsight, perhaps we should not have been surprised—if you have a lot of data that you want to sort, it turns out that databases are pretty good!

8 RELATED WORK

Operating systems and databases have a complex relationship. Early on, databases offered an alternative to general-purpose operating systems for resource and data management [5]. Today, databases are often thought of as applications that make use of the operating system's facilities for managing hardware resources. However, databases and operating systems are often at odds, e.g., with respect to cache replacement, scheduling, and file management [16].

ROSI [14] is the first work that we are aware of to propose a more extreme position: treat an operating system as if it were a database to improve usability. osquery [12] mirrors important OS state in a user-space database, but offers only weak semantics including stale reads and non-atomic updates to kernel state. At the other end of the design spectrum, DBOS [13] advocates for a "clean slate" approach that entirely replaces the OS with a query processor.

Numerous systems [1, 2, 7, 9, 13, 14] have applied ideas from databases to simplify systems concerns from networking to cluster management to single-host operating systems. There also has been significant prior work on database-based file systems [8, 11, 15]. All of these prior projects propose greenfield designs, while OSDB allows incremental adoption of database techniques to a production-ready operating system.

9 CONCLUSION

We have begun our exploration of ideas in exposing and managing operating system state in a principled way using the relational model. Our initial experiences have been overwhelmingly positive, providing a novel system with low overhead and high levels of functionality. Our approach provides strong consistency, powerful join operations that provide new ways to understand and act upon system state in real time, and at the cost of a less than a hundred lines of code per kernel data structure. Our incremental, "Ship of Theseus" approach to extending the operating system provides better results to potential consumers of a production operating system, while avoiding challenges inherent in green field approaches.

ACKNOWLEDGMENTS

This work is partially supported by the National Science Foundation award #2212235. We would like to thank Pasquale Polverino for his work on porting OSDB to Linux.

REFERENCES

[1] 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 *Proc. 5th European Conference on Computer Systems*. 223–236.

[2] Nalini Belaramani, Jiandan Zheng, Amol Nayate, Robert Soulé, Mike Dahlin, and Robert Grimm. 2009. PADS: A Policy Architecture for Distributed Storage Systems. In *Proc. 6th ACM/USENIX Symposium on Networked Systems Design and Implementation*. 59–73.

[3] C. J. Date. 1984. A Critique of the SQL Database Language. *SIGMOD Record* 14, 3 (nov 1984), 8–54.

[4] Ron Frederick, Stephen L. Casner, Van Jacobson, and Henning Schulzrinne. 1996. RTP: A Transport Protocol for Real-Time Applications. RFC 1889.

[5] Jim Gray. 1978. *Notes on Data Base Operating Systems*. Springer-Verlag, 393–481.

[6] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2008. SQCK: A Declarative File System Checker. In *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation*.

[7] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: Automatic Management of Data and Computation in Datacenters. In *Proc. 9th USENIX Symposium on Operating Systems Design and Implementation*. 75–88.

[8] Yige Hu, Zhiting Zhu, Ian Neal, Youngjin Kwon, Tianyu Cheng, Vijay Chidambaram, and Emmett Witchel. 2018. TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions. In *Proc. 2018 USENIX Annual Technical Conference*. 879–891.

[9] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E Gay, Joseph M Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2006. Declarative Networking: Language, Execution and Optimization. In *Proc. 2006 ACM SIGMOD International Conference on Management of Data*. 97–108.

[10] Erik Meijer and Gavin Bierman. 2011. A Co-Relational Model of Data for Large Shared Data Banks. *Commun. ACM* (2011).

[11] Michael A. Olson. 1993. The Design and Implementation of the Inversion File System. In *Proc. 1993 Winter USENIX Conference*.

[12] osquery [n.d.]. osquery. Online. https://osquery.io[Accessed July 2024].

[13] 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. 2021. DBOS: A DBMS-Oriented Operating System. *Proceedings of the VLDB Endowment* 15, 1 (sep 2021), 21–30.

[14] Robert Soulé, Peter Alvaro, Henry F. Korth, and Abraham Silberschatz. 2024. Research Pearl: The ROSI Operating System Interface. arXiv:2409.14241 [cs.DB] https://arxiv.org/abs/2409.14241

[15] R. P. Spillane, S. Gaikwad, E. Zadok, C. P. Wright, and M. Chinni. 2009. Enabling Transactional File Access via Lightweight Kernel Extensions. In *Proc. 7th USENIX Conference on File and Storage Technologies*. 29–42.

[16] Michael Stonebraker. 1994. *Operating System Support for Database Management.*

[17] UnixBench [n.d.]. byte-unixbench. Online. https://github.com/kdlucas/byte-unixbench[Accessed July 2024].

[18] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-Scale Cluster Management at Google with Borg. In *Proc. 10th European Conference on Computer Systems*.

[19] vtab [n.d.]. The Virtual Table Mechanism Of SQLite. Online. https://www.sqlite.org/vtab.html[Accessed November 2024].