Debugging the OmniTable Way

Andrew Quinn, *UC Santa Cruz;* Jason Flinn, *Meta;* Michael Cafarella, *MIT;*
Baris Kasikci, *University of Michigan*

https://www.usenix.org/conference/osdi22/presentation/quinn

This paper is included in the Proceedings of the
16th USENIX Symposium on Operating Systems
Design and Implementation.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-28-1

Open access to the Proceedings of the
16th USENIX Symposium on Operating
Systems Design and Implementation
is sponsored by

NetApp

Debugging the OmniTable Way

Andrew Quinn, *UC Santa Cruz*
Jason Flinn, *Meta*
Michael Cafarella, *MIT*
Baris Kasikci, *University of Michigan*

Abstract

Debugging is time-consuming, accounting for roughly 50% of a developer's time. To identify the cause of a failure, a developer usually tracks the state of their program as it executes on a failing input. Unfortunately, most debugging tools make it difficult for a developer to specify the program state that they wish to observe and computationally expensive to observe execution state. Moreover, existing work to improve our debugging tools often restrict the state that a developer can track by either exposing incomplete execution state or requiring manual instrumentation.

In this paper, we propose an `OmniTable`, an abstraction that captures all execution state as a large queryable data table. We build a query model around an `OmniTable` that supports SQL to simplify debugging without restricting the state that a developer can observe: we find that `OmniTable` debugging queries are more succinct than equivalent logic specified using existing tools. An `OmniTable` decouples debugging logic from the original execution, which SteamDrill, our prototype, uses to reduce the performance overhead of debugging. The system employs *lazy materialization*: it uses deterministic record/replay to store the execution associated with each `OmniTable` and resolves queries by inspecting replay executions. It employs a novel multi-replay strategy that partitions query resolution across multiple replays and a parallel resolution strategy that simultaneously observes state at multiple points-in-time. We find that SteamDrill queries are an order-of-magnitude faster than existing debugging tools.

1 Introduction

Developers spend the majority of their time debugging their software [26]. Usually, a developer debugs by iteratively executing their program and using debugging tools to observe its state during the failing execution.

A developer can often identify the root cause of a simple bug by making a few observations about their program's execution state. However, to identify the root cause of a complex bug, such as a atomicity violation or performance degradation, the developer will need to make sophisticated observations. Conceptually, we can model the logic for such sophisticated observations as a *debugging program*, designed to make sense of the failing program. For example, when debugging, a developer may observe all of the values to which a variable is assigned during an execution. Their debugging program consists of a set data structure to store the values, logic after each assignment in the failing execution that adds the assigned value to the set, and a print statement to print the set when the execution terminates.

Unfortunately, many debugging tools, such as `gdb`, "printf"-debugging, and binary instrumentation, support debugging programs that have both high programming complexity and high performance overhead. Such tools support procedural debugging programs that observe state as a failing program executes. Procedural debugging programs have considerable programming complexity, especially for sophisticated tasks that track execution state over time (§6.2). High complexity can lead to bugs [15, 44] that prevent a developer from understanding the failing program. Additionally, such debugging programs impose high performance overhead since sophisticated debugging programs observe *a lot* of execution state which existing tools extract within the same execution context as the failing program. Consequently, procedural debugging programs can slow execution by between a factor of 2–1000 (§6.3), which can preclude the use of sophisticated debugging programs [11].

Alas, prior debugging work retains, or even exacerbates, high programming complexity or high performance overhead to improve the other. Some proposals lower the performance overhead of debugging by employing parallelism (e.g., Speck [30], SledgeHammer [35]) or low-level optimizations (e.g., optimistic hybrid analysis [9], efficient path profiling [3]). At best, such techniques require redesigning debugging programs, at worst, they require novel research contributions to accelerate even a single task (e.g., taint tracking [4]).

High-level debugging tools decrease programming complexity by allowing a developer to observe and summarize execution state using a high-level programming model (e.g., Fay [12], G2 [17], EndoScope [7]). However, such tools retain high performance overhead since they perform a debugging program's observations while executing the original failing program. To curtail the effect of high performance overhead, high-level debugging tools restrict the execution state that a developer can observe, either explicitly (e.g., by minimizing the times when a debugging program can observe state [12, 14, 25]) or implicitly (e.g., by requiring extensive manual instrumentation to specify observable execution state [17, 24, 40]). Such systems are well suited for tasks that only need to observe partial execution state, such as distributed tracing [24] or identifying specific classes of bugs [25], but are less suited for debugging complex issues.

This paper proposes the `OmniTable` query model, a new debugging paradigm that reduces the programming complexity and performance overhead of debugging without restricting the execution state that a developer can observe. The new `OmniTable` abstraction empowers the model. An `OmniTable` reduces programming complexity by presenting all of an execution's state as a large queryable data object. An `OmniTable` reduces performance overhead by decoupling a debugging program's observations from the original programs' execution to enable automated optimizations of debugging programs.

The `OmniTable` query model enables debugging programs that can observe any execution state with low programming complexity by turning to relational logic. Concretely, an `OmniTable` is a database table representation of an execution that contains all architectural state (i.e., the value of all bytes of memory and all registers) before every instruction executed by the program. From a developer's perspective, an `OmniTable` is extracted as a program executes and can later be queried using an extended SQL language to observe the execution's state. The model bridges the gap between the architectural state in an `OmniTable` and common debugging abstractions (e.g., the functions executed, variables assigned, etc.) by re-purposing existing database primitives (e.g., high-level views) and creating new query operators (e.g., traversal functions).

Unfortunately, naively materializing an `OmniTable` would lead to considerable performance overhead, since it would require performing a core-dump before every instruction. Instead, our prototype, SteamDrill, employs *lazy materialization*. Lazy materialization defers the calculation of an `OmniTable`'s state until a developer queries it. Rather than extract an `OmniTable` in its entirety during execution, SteamDrill uses deterministic record and replay to store the execution associated with the `OmniTable`. Deterministic record and replay enables SteamDrill to compress and store years worth of `OmniTables` on a commodity hard drive [10]. When a developer issues a query over an `OmniTable`, SteamDrill generates instrumentation which it injects into a new replay of the execution associated with the `OmniTable` to produce the execution state needed for the query.

SteamDrill reduces performance overhead by decoupling a debugging query's execution from the original program execution. SteamDrill uses a query planning approach that decomposes a debugging query into independent stages. SteamDrill implements a novel multi-replay query resolution strategy that executes each stage in a separate replay so that it can use data that is computationally inexpensive to observe (e.g., data about functions in an `OmniTable`) to reduce the compute cost of observing data that is computationally expensive to observe (e.g., data about each instruction in an `OmniTable`). In essence, multi-replay resolution uses the decoupling between an `OmniTable` query and the original execution to repeatedly observe `OmniTable` state at increasing detail. SteamDrill also uses decoupling to observe execution state from multiple points-in-time in parallel using thousands of machines [35, 47].

We built a SteamDrill prototype on top of Spark [47] and Arnold [10]. We evaluate the prototype using 5 detailed case studies of bugs reported in popular open-source applications (Memcached, redis, Apache, and SQLite). We identified 14 debugging programs that a developer would use to identify the root cause of each bug, including ad-hoc programs (e.g., "How many control-flow instructions did my function issue?") and standard dynamic analyses (e.g., a memory leak detector). We implemented the debugging programs using `OmniTable` queries and `gdb`'s python bindings, which provide a high-level language over `gdb` features. We found that `OmniTable` queries require up to 11.67 times fewer lines (with a geometric mean of 3.74 times fewer lines), up to 5.73 times fewer terms (with a geometric mean of 1.70 times fewer terms), and up to 23.49 times less estimated development time (with a geometric mean of 2.75 times less estimated development time) than `gdb` scripts. We evaluated the performance of SteamDrill on 3 representative debugging queries and find that it is faster than `gdb` by a factor of 99 based upon geometric mean.

We make the following contributions:

- The `OmniTable` query model, which decouples a debugging program from a failing execution to reduce the programming complexity and performance overhead of debugging.
- SteamDrill, which optimizes `OmniTable` queries using query planning, cluster-scale parallelization, and a novel multi-replay query resolution approach.
- An evaluation of 5 case studies and 14 queries that shows that `OmniTable` queries are more succinct and SteamDrill has lower latency than state-of-the-art tools.

2 Motivation

In this section, we describe a motivational case study showing how the `OmniTable` query model simplifies debugging. In the case study, a developer uses an `OmniTable` to diagnose a performance problem in redis [36]. In the study, a developer deploys redis as an in-memory key-value LRU cache for a

Vars(ot)	
Column Name: Type	Description
time: Long	Time of instruction
thread: Long	Thread that executed instruction
eip: Long	Program counter of instruction
name: String	Variable name
value: Any	Assigned value

Funcs(ot)	
Column Name: Type	Description
enterTime: Long	Time of function entry
exitTime: Long	Time of function exit (or null)
name: String	Function name
thread: Long	Thread that executed function
callStack: String	Call stack of function
args: Map[String->Any]	Argument values
rVal: Any	Return value of function execution

Figure 1: The schema of the `Funcs(ot)` and `Vars(ot)` views. Each line in each table describes a column in the view.

slow back-end service. Over time, the average end-to-end latency of their deployment creeps upwards; the developer notices that the increase correlates with the back-end service processing a higher percentage of requests.

The bug is challenging to diagnose since the developer only starts with a high-level symptom and is unaware of which parts of the program are related to the error. To determine the root cause of the bug, the developer summarizes an execution's behavior over time. The `OmniTable` allows the developer to observe all of the execution state of the program without requiring instrumentation; the query model's support for SQL aggregations allows the developer to succinctly summarize large amounts of execution state. Moreover, the `OmniTable` enables repeated queries over the same buggy execution, instead of requiring the bug be reproduced for each query.

In contrast, summarizing execution state over time is challenging with existing tools (§7). To use a procedural debugging tool (e.g., `gdb`), the developer must identify numerous instrumentation points, track execution state over time in complex data-structures, and implement algorithms to group data and calculate statistics. Other debugging tools simplify execution summarization, but provide incomplete interfaces in that they do not expose the execution state (e.g., PTQL [25], Fay [12]) or do not support the operators (e.g., Pivot Tracing [24], Execution Mining [20]) required for this case study. Finally, instrumentation-based tools (e.g., G2 [17], Pivot Tracing [24]) require extensive manual instrumentation to perform the necessary observations.

The developer uses 5 `OmniTable` queries to identify the root cause of the performance degradation. Rather than query an `OmniTable` directly, the developer uses *derived views* to simplify their queries. A derived view labels execution state according to an abstraction of execution behavior, such as the functions, in-scope variables, or memory read by each instruction in an `OmniTable`. Below, we describe the derived views

time	eip	name	value
100	0x1000	“used”	1000
100	0x1004	“entry”	NULL
102	0x1004	“used”	1000

Figure 2: Example data from `Vars(ot)` .

enterTime	exitTime	name	args	rVal
100	200	“lookupKey”	{“key”:“k1”}	100
100	200	“lookupKey”	{“key”:“k2”}	100
100	200	“incrRefCount”	{“key”:“k1”}	NULL

Figure 3: Example data from `Funcs(ot)` (omitting the `callStack` and `thread` columns). .

that the developer uses. Then, we describe the `OmniTable` queries that the developer uses and compare them to debugging programs expressed using existing debugging tools.

2.1 Views

The developer uses two derived views, `Vars(ot)` and `Funcs(ot)`, which can be calculated over an `OmniTable`: `ot`. Figure 1 shows their schemas.

Vars(ot). The `Vars` view contains the value of all in-scope variables at each instruction in an `OmniTable`. Each row identifies the value of a single in-scope variable at a single instruction, regardless of whether that instruction accesses the variable. Figure 2 shows a few rows of the `Vars(ot)` view for the `OmniTable` for the buggy execution of redis used during this case study. A developer references the `Vars` view of an `OmniTable`, `ot`, by specifying `Vars(ot)` in their query.

Funcs(ot). The `Funcs(ot)` view contains information about the functions executed in an `OmniTable`—each row contains state from either the entry to or exit from a function execution contained in the `OmniTable`, `ot`. For example, Figure 3 contains a few rows of the `Funcs(ot)` view for the `OmniTable` for the buggy execution of redis used during this case study. The `enterTime`, `callStack`, and `args` columns are extracted upon function entry; the `exitTime` and `rVal` columns are extracted upon function exit (and are `NULL` for functions that never return); and the `name` and `thread` are extracted at both entry and exit and joined to match the entry and exit of each function. The `rVal` and `args` columns use the polymorphic type, Any, to encode different function signatures. For example, a developer specifies `args["i"]` to get the value of the argument `i` passed to a function.

The `time`, `enterTime` and `exitTime` columns from `Vars(ot)` and `Funcs(ot)` expose an ordering of events contained in the views and provide a primary key that uniquely identifies each row in the views. Moreover, a developer can use the the `time`, `enterTime`, and `exitTime` columns to correlate data across the `Funcs(ot)` and `Vars(ot)` views for the `OmniTable`. For example, a developer can determine the value of each in-scope variable at the entry to each function by joining `Funcs(ot)` and `Vars(ot)` on `enterTime = time`; the second query uses this feature (§2.2.2).

```
1 Select enterTime, count(distinct args["key"]) Over(
2   Order By enterTime
3   Rows Between 10000 Preceding and Current Row)
4 From Functs(ot)
5 Where f.name="lookupKey"
```

Listing 1: The developer's first query.

```
1  from gdb import Breakpoint, parse_and_eval
2  from collections import deque, defaultdict
3  class bp(Breakpoint):
4      keys=deque()
5      indexed=defaultdict(int)
6      def stop(self):
7          keys.append(parse_and_eval("key"))
8          indexed[keys[-1]] += 1
9          if len(keys) > 10000:
10             indexed[keys[0]] -= 1
11             if indexed[keys[0]] == 0:
12                 del indexed[keys[0]]
13             keys.popleft()
14         print (len(indexed))
15         return False
16 bp("lookupKey")
```

Listing 2: The developer's first query written for gdb's Python bindings.

2.2 Queries

Next, we describe how the developer diagnoses the cause of the performance bug. First, they use deterministic record and replay to capture the `OmniTable` for an execution of redis during which the issue occurs. Then, they construct and execute the following five `OmniTable` queries.

2.2.1 First Query

The developer's first query (Listing 1) uses a windowed aggregation to approximate the number of items that the deployment caches in redis (i.e., the working set size) during the performance degradation. The developer suspects that the working set size increases over time, which would lead to additional cache misses in redis. Each cache miss sends a request to the back-end service, so this hypothesis would explain the creeping latency of the deployment.

The developer begins by inspecting redis's source code to identify the function, `lookupKey`, that finds an item in the cache. For each `lookupKey` execution, the developer creates a window containing the preceding 10,000 executions of `lookupKey` and counts the number of distinct keys passed to each function call in each window. The `OmniTable` query model succinctly represents this logic using *SQL aggregates*. SQL aggregates calculate a mathematical operation (e.g., **`count`**, **`sum`**) over a group of rows. An aggregate can operate over a window of requests, in which each group is an ordered list of rows that match an **`Over`** clause, as is the case in this query. Alternatively, an aggregate can operate over a group of rows that match a **`Group By`** clause, as is the case in the developer's third query (§2.2.3).

In detail, the query uses the **`Over`** operator to create sliding windows, each of which contains 10,000 consecutive calls to `lookupKey`, by ordering `Funcs(ot)` by `enterTime` (Lines 2–3). The query filters non-`lookupKey` windows (Line 5). It counts the number of distinct keys passed to each function call in each window using the `key` argument (`args["key"]`) and the **`count`** and **`distinct`** operators.

Existing debugging tools either cannot support the query, impose high programming complexity, or impose high performance overhead. EndoScope [7] and Fay [12] could support the developer's query, but imposes a high overhead since they tightly couple debugging logic's execution with the original program execution. Most high-level debugging tools do not support windowed-aggregations and are either unable to compute the query (e.g., Pivot Tracing [24], Execution Mining [20]) or require a developer to write a custom operator to compute the query (e.g., G2 [17] requires expressing the window clause in terms of a vertex-based graph-traversal). Instrumentation-based debugging tools (e.g., Pivot Tracing [24]) would require the developer manually instrument the `lookupKey` function to produce the value of `key`.

An equivalent procedural debugging program is complex. The debugging program must navigate the performance-complexity tradeoff—creating a program with high overhead is straightforward, but creating one with low overhead requires complex logic to ensure consistency of two data structures. A mistake can lead to a misdiagnosis of the bug—our first version of the debugging program included such a mistake.

Listing 2 shows an implementation for `gdb`'s Python bindings, which provide a Python interface for `gdb` features such as breakpoints and backtraces. The developer creates a custom `Breakpoint` class, `bp` (Lines 6–15); by creating a `bp` with the argument `"lookupKey"` (Line16), the developer instructs the `gdb` framework to call the developer-supplied `stop` function at each call to `lookupKey`. The developer tracks the sliding window of 10,000 requests by storing the value of the `key` argument into `keys`, a `queue`, and removing the first element if there are more than 10,000 elements in `keys` (Lines 7, 9, and 13). The developer could recompute the unique values in `keys` in `stop`, but that would add significant performance overhead since `lookupKey` is executed frequently. Instead, the developer uses a dictionary object, `indexed`, to track the number of times each `key` value appears in the `keys` window (Lines 5, 8, and 10–12). This logic is subtle and challenging to get right—for example, we initially used a `set` to track the unique elements in `keys` instead of using a dictionary to track the number of times each element appears in `keys`. Our buggy implementation erroneously removes elements from `indexed` and produces misleading results.

```
1  Select v.time, v.value
2   From Vars(ot) as v Join Funcs(ot) as f
3  Where v.name = "used" And f.name = "lookupKey"
4    And f.enterTime = v.time
```

Listing 3: The developer's second query.

```
1  Define DefinedMemory(ot) as:
2  Select m.rVal as pointer, m.exitTime as start,
3         m.callStack as allocSite, f.enterTime as end
4         m.arg["size"] as size
5  From Funcs(ot) as m Left NextJoin Functs(ot) as f
6    On m.exitTime, f.enterTime,
7    m.name="malloc" And m.exitTime<=f.enterTime
8   And f.name="free" And m.rVal=f.arg["ptr"]
9
10 Select start, allocSite, sum(size)
11   Over(Partition By allocSite Order By start)
12 From DefinedMemory(ot)
13 Where end=NULL
```

Listing 4: The definition of the `DefinedMemory(ot)` view (Lines 1–8) and the developer's third query (Lines 10–13).

2.2.2 Second Query

Surprisingly, the working set size of the cache is fairly constant throughout the execution. Consequently, poor cache performance may arise from poor eviction decisions for the workload or from a decrease in the number of items in the cache over time. The developer's second query determines the number of items in the cache over time by checking the number of items in the cache before each execution of the `lookupKey` function (Listing 3). redis stores the size of the cache in a global variable, `used`; the query uses the `Vars(ot)` view to access the value of the variable (Lines 2–3). It prevents the query from producing extremely large amounts of data by using a `Join` to limit the rows to only those when the execution enters `lookupKey` (Lines 2–4).

Unlike the `OmniTable` query model, many debugging tools do not expose the value of variables at arbitrary points-in-time and cannot support the developer's second query (e.g., Fay [12], PTQL [14]). Endoscope [7] could support the query, but only exposes variable values when they are assigned and requires the query identify the most recent preceding assignment of `used` for each call to `lookupKey`. Instrumentation-based tools (e.g., Pivot Tracing [24], G2 [17]) and procedural tools (e.g., `gdb`) require instrumenting the `lookupKey` function to produce the value of `used`.

2.2.3 Third Query

The second query's output shows that the number of items in the cache decreases throughout the execution. Since the redis configuration specifies a total memory size for the cache and the deployment uses constant sized items, a declining number of items in the cache implies that there is a memory leak. Unfortunately, redis does not clean up memory on shutdown, so existing leak detection tools (e.g., memcheck [29] and AddressSanitizer [39]) report nearly all memory allocations as leaks.

```
Select dm.pointer, Count(*)
From Funcs(ot) as r Join DefinedMemory(ot) as dm
Where dm.allocSite=leakSite And dm.exit=NULL
  And r.name="decrRefCount" Or r.name="incrRefCount"
  And dm.pointer=r.arg["obj"]
Group By dm.pointer, r.name
```

Listing 5: The developer's fourth query.

The developer's third query uses an alternative approach: it tracks the number of leaked bytes by each allocation site (defined as the call stack of the allocation) over time. Allocation sites that produce bug-inducing leaks will have a gradual increase of leaked bytes throughout the execution. The developer's query observes three separate types of execution events with different happens-before relationships, which is greatly simplified by the `OmniTable` query model.

The developer first creates a view, `DefinedMemory(ot)` that contains the window of time during which each memory object is defined, i.e., allocated and not freed (Listing 4). The view joins each call to malloc with the subsequent call to free whose pointer argument, `ptr`, is equal to the return value from malloc (Lines 5–8). Since a pointer could be reallocated by malloc after being freed, the query only matches calls to free that occur after the call to malloc (`m.exitTime<f.enterTime` at Line 7). Additionally, it only matches each malloc with the next matching call to free, as ordered by `exitTime` and `enterTime`, respectively, by using **`NextJoin`**, a new operator provided by the `OmniTable` query model (Lines 5–6). The developer uses **`Left NextJoin`**, which produces output from the left relation even if there is no matching row in the right relation, so that memory which is never freed (i.e., leaked) has a `NULL` value for the `end` column.

The third query tracks the amount of data leaked by each allocation site over time. For each leaking allocation (Lines 12–13), the developer constructs a window containing all preceding leaking allocations from the same allocation site by using the **`Over`** operator (Line 11). They sum the number of bytes leaked within each window (Line 10).

Like with the previous queries, existing debugging tools either cannot support the third query, impose high programming complexity, or impose high performance overhead.

2.2.4 Fourth Query

The output of the third query identifies a single leaking allocation site, `leakSite`. redis uses reference counters to manage allocations from `leakSite`. Each counter tracks the number of live references to each object; redis should garbage collect the object when the count reaches 0. So, the developer suspects a problem in the reference counting and writes a query to count the updates to the reference counters of leaking objects (Listing 5). They identify leaked objects that were allocated at `leakSite` (Lines 2–3) and match each leaked object with corresponding executions of `decrRefCount` and `incrRefCount`, the functions that modify reference counts (Lines 2, 4, and 5). The developer groups the rows by object and function name (Line 6) and determines the number of calls to increment and decrement the counter (Line 1).

Like the previous queries, existing debugging tools either cannot support the developer's fourth query, impose high programming complexity, or impose high performance overhead.

2.2.5 Fifth Query

The fourth query's output shows that the execution calls `incrRefCount` and `derRefCount` the same number of times on the leaked objects, indicating a problem in the implementation of `incrRefCount` or `decrRefCount`. The developer chooses a few candidate objects and determines the call stack of the calls to `incrRefCount` and `decrRefCount` for these objects. The final query[1] shows that the leaked object's reference counts are decremented by a lazy deallocation thread and by a logging thread and points to the root cause of the bug, a race condition in `decrRefCount`. In the fix for the original bug report, the developer redesigning the logging thread to copy objects instead of sharing them.

3 The `OmniTable` Query Model

We outline the features of the `OmniTable` query model that enable a developer to succinctly reason about the entire history of execution state. The central abstraction is an `OmniTable`, a database table containing all user-level architectural state of an execution immediately before every instruction in the execution. Concretely, an `OmniTable` contains a column for every byte of architectural state and a row immediately before each instruction. The model supports debugging queries over an `OmniTable` expressed using SQL-style **`Select`**...**`From`** ...**`Where`** queries.

Alas, an `OmniTable` alone offers an inadequate debugging interface, since a developer would need to reference execution state in architectural terms. For example, a developer would need to determine the exact memory location of each variable whose value they wish to observe. So, the `OmniTable` model adopts and extends database concepts to enable debugging abstractions. It uses `Generators`, user-defined-table-functions that allow queries to reference non-execution state (e.g., debugging symbols). It adds new operators for debugging, such as traversal functions and new `Join` variants. Finally, the model uses derived views to label an `OmniTable`'s state according to familiar debugging abstractions such as the functions executed in an `OmniTable` or the variables in scope at each instruction in an `OmniTable`. A single row in a high-level view can expose execution state from multiple points-in-time during the execution (e.g., `Funcs(ot)`). Below, we elaborate on the model's components.

[1]Omitted for brevity, this query is a self-join of the `Funcs(ot)` view

3.1 Relations

The `OmniTable` query model supports two relational base tables, `OmniTables` and `Generators`. It supports columns with primitive types (e.g., Long, String), Structs, Maps, Arrays, and Any, a polymorphic type.

OmniTable. An `OmniTable` is a database table that includes all architectural execution state immediately before each instruction in the execution; Figure 4 shows an example. Before each instruction, the `OmniTable` contains the current `thread`, the value of all registers and memory addresses, and the top of the `stack` of the thread. To dereference an address, `addr`, a query specifies `Memory[addr]`. Additionally, each row includes a monotonically increasing logical `time`, which provides a total ordering of events in the `OmniTable` and uniquely identifies each row. In a multi-threaded program, the `time` field is a total ordering that is consistent with the partial ordering of the original execution. Together, the `thread` and `time` columns enable a developer to reason about concurrency.

Generators. `Generators` allow developers to bridge the semantic gap between traditional programming abstractions (e.g., functions, lines of code) and an `OmniTable`'s architectural state by referencing non-execution state (e.g., debugging symbols). For example, `Defs` identifies the functions defined in a binary; the following produces all such definitions for an executable, "a.out": **`Select`** `*` **`From`** `Defs("a.out")`. `Generator` input can depend on query data. For example, `Binaries` is useful for bootstrapping queries; it uses the deterministic record/replay log to identify the binaries mapped into the address space of an `OmniTable`. The following determines all functions defined in all binaries that are loaded in `ot`, an `OmniTable`, which we use to define the `FuncDefs` view: **`Select`** `*` **`From`** `Defs(`**`Select`** `*` **`From`** `Binaries(ot))`. Developers create `Generators` by writing a program that produces relational output; we have built a `Generator` that determines all variables defined in in all binaries mapped into an `OmniTable`, and one that creates stored procedures that produce the memory read and written by each instruction in an `OmniTable`.

3.2 Relational Operators

The model supports **`join`**, **`group by`**, **`order by`**, and **`pivot`**. It also introduces three `Join` variants for debugging.

StackJoin. SQL is unable to model a function stack, which would prevent the `OmniTable` query model from expressing critical debugging abstractions, such as `Funcs(ot)`, which is used in all of the queries in the redis case study (§2). Prior high-level debugging tools either remove support for such se-

Metadata				Registers				Memory			
time	**thread**	**stackTop**	**…**	**eip**	**eax**	**ebx**	**…**	**0x0**	**0x1**	**…**	**0xffffffff**
1	100	0x2000	…	0x1000	1	1	…	1	1	…	1
2	100	0x2000	…	0x1004	1	1	…	1	1	…	1
					…						
1000	100	0x2000	…	0x1064	1	1	…	1	1	…	1

Figure 4: An `OmniTable` for a short execution.

```
Select *
From fenter(ot) as e StackJoin freturn(ot) as r
On e.time, r.time, e.thread=r.thread AND e.name=r.name
```

Listing 6: An Example StackJoin.

mantics (e.g., PTQL [14]) or require manual instrumentation (e.g., G2 [17]). Instead, the `OmniTable` model creates a new operator, **`StackJoin`**.

As an example, suppose that `fenter(ot)` is a view that contains a row for each function entry in `ot`, an `OmniTable`, with columns `name`, `thread`, and `time` for the name of the function, thread that entered the function, and time of entry; and that `freturn(ot)` is a view conataining a row for each function return in `ot`, an `OmniTable`, with columns `name`, `thread`, and `time` for the name of the function, thread that returned from the function, and time that the function returns. Listing 6 shows a **`StackJoin`** that matches each function entry with its function return. **`StackJoin`** partitions `fenter(ot)` and `freturn(ot)` into groups that match on `thread` and `name`. For each group, the operator orders the rows `fenter(ot)` by `time` and orders the rows from `freturn(ot)` by `time`. Repeatedly, the operator pushes all rows from `fenter(ot)` onto a stack until it finds a row that occurs after the next row in `freturn(ot)`; it then produces a row by joining the last row added to the stack and the next row in `freturn(ot)`.

OrderedJoins. When debugging, developers often reason about the next, or previous, event that satisfies some condition. For example, in the `DefinedMemory(ot)` view, the developer matched each call to `malloc` with the next call to `free` on the same pointer (§2.2.3). SQL requires inconvenient subqueries for this reasoning, so, the `OmniTable` query model adds two new ordered join operators. The `NextJoin` operator determines the next matching row across two relations and can be used to determine the next function executed by a thread or the next access to a shared variable: **`NextJoin on`** `ord1, ord2, equals` joins each row in the left relation, ordered by `ord1`, with the next row from the right relation, ordered by `ord2`, where `equals` is true. The `PrevJoin` operator does the opposite.

3.3 Column Operators

The `OmniTable` query model supports many column operators, including arithmetic and conditional operators, field expressions (`a.b`), subscript expressions (`a[b]`), traversal functions, stored procedures, and standard aggregations (e.g., **`Count`**, **`Max`**, **`Min`**, etc.) over groups and windows. The model also supports pointer dereferences by converting them into expressions over the `Memory` column (e.g., `a->b` becomes `Memory[a].b`). We elaborate on traversal functions and stored procedures.

Traversal Functions. SQL makes it difficult to traverse the elements in a data structure since it does not support unbounded traversals. So, the `OmniTable` query model builds new primitives for these operations. Given a pointer-typed `column` and a `field` within the pointed-to type, the `traverse (column, field)` expression produces a row of output for each element in the transitive closure of the structure by starting at `column` and following `field` pointers until the value is `NULL`. For example, `traverse(node, "next")` traverses the `next` pointer of all elements in a structure, starting at `node`.

Stored Procedures. Debugging logic often varies by execution context (e.g., the memory location of function arguments varies by function). Stored procedures [43] store relational logic in a table and allow a query to decide query logic during query resolution. Developers call stored procedures in their `OmniTable` queries with function syntax; for example, a developer could specify `Var_Loc(esp)` to use `Var_Loc`, a stored procedure that calculates the memory location of a variable given the value of the stack pointer.

3.4 Derived Views

The `OmniTable` query model allows developers to construct derived views for labeling execution state. The **Define** operator in Listing 4 shows how a developer constructs `DefinedMemory(ot)`. Our implementation provides three high-level views, `Funcs(ot)` (§2.1), `Vars(ot)` (§2.1), and `Insts(ot)`, a view that encodes information about each instruction in an `OmniTable`.

4 Design

In this section, we describe the design of SteamDrill, our system that supports the `OmniTable` query model. From a developer's perspective, SteamDrill computes queries over `OmniTables` that are extracted during execution and stored in a database. However, materializing an entire `OmniTable` is infeasible due to high storage and compute costs: an `OmniTable`'s size is equal to the addressable memory size

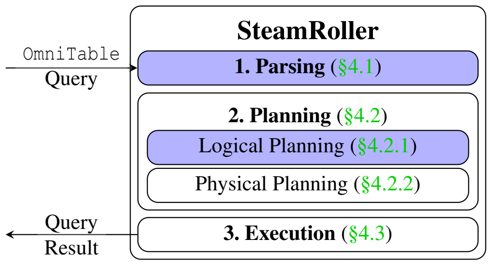

Figure 5: SteamDrill steps for query resolution. Blue steps re-use or customize existing approaches, and white steps are new designs.

```
Select eip, read, write
From Insts(ot) as i Left Join DefinedMemory(ot) as dm
   On (i.read=dm.pointer Or i.write=dm.pointer)
   And i.time>=dm.start And i.time<dm.end
Where dm.start=NULL
```

Listing 7: A simplified undefined use query.

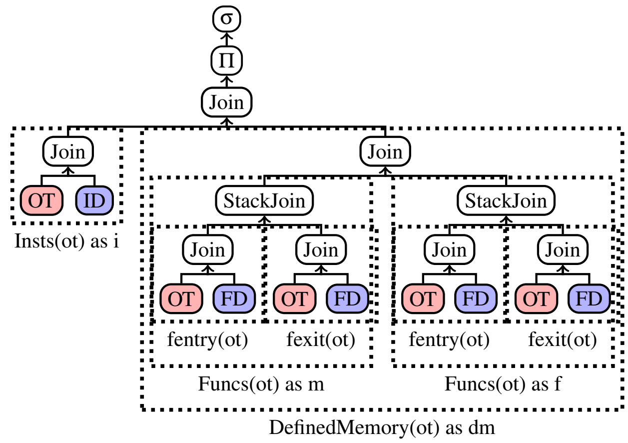

Figure 6: The relational tree for Listing 7. OmniTables are red ovals labeled with OT. Generators are blue ovals; InstructionDefs are ID nodes and FuncDefs are FD nodes. Relational operators are white rectangles; **where**, **join**, and **select** clauses are σ, Join, and Π nodes. The logic for each derived view is encapsulated in a dotted rectangle.

times the number of executed instructions and reaches petabytes, or even exabytes, for mere seconds of execution.

SteamDrill introduces lazy materialization as a solution. Rather than materializing an OmniTable during execution, SteamDrill uses deterministic record and replay [10] to capture a log of non-deterministic inputs to the execution. The system uses the log to generate OmniTable state on-demand by instrumenting and re-executing the original execution as necessary to resolve debugging queries. Delaying OmniTable materialization allows SteamDrill to filter OmniTable data *before* extracting state instead of afterwards.

In the rest of this section we describe how SteamDrill resolves debugging queries. Listing 7 presents a simplified use-after-free query as a running example. The query uses the Insts(ot) view to identify the memory read and written by each instruction in an OmniTable. It joins Insts(ot) with the DefinedMemory(ot) view from the third redis query, to match each memory access with the region of time during which its pointer was defined (Listing 4). The query uses a **Left Join** and identifies rows where start is NULL to identify the instructions that operate on undefined memory.

SteamDrill's design mirrors that of typical database management systems [2] (Figure 5). First, SteamDrill uses conventional SQL parsing to decompose a query into a tree of relational operators (internal nodes) over data tables (leaves) (§4.1). The tree contains a separate leaf node for each OmniTable referenced in the query. The relational operators that consume data from each OmniTable in the tree identify the execution state that the query needs from that OmniTable. During execution, SteamDrill uses these operators to limit the materialization of each OmniTable in the query by generating instrumentation that it injects into replay executions (§4.3).

The order of OmniTable materialization has a large impact on the amount of materialized data and the latency of query resolution. Delaying the materialization of an otherwise computationally expensive-to-materialize OmniTable often allows SteamDrill to filter the computationally expensive materialization using data from a less computationally expensive-to-materialize OmniTable. For example, to reduce the latency of the use-after-free query, the system first materializes the OmniTable state needed for the DefinedMemory view and uses the materialized data to filter the materialization of the Insts(ot) view.

Accordingly, SteamDrill uses multi-replay resolution—it splits OmniTable materialization across multiple replay executions. It uses query planning to determine the OmniTable materialization order and assigns OmniTable materialization to replay executions(§4.2). SteamDrill implements OmniTable-specific optimization strategies to decide the join order and join algorithm for each join in the tree (§4.2.1) which it uses to assign each operator to a stage. The system uses a single replay execution to materializes all OmniTables in the same stage. This approach minimizes the number of replay executions (since each replay execution adds additional overhead) and enables SteamDrill to limit OmniTable materialization.

4.1 Parsing

First, SteamDrill converts the query into a tree of relational operators over data tables using a standard SQL parser [2]; the tree encodes the logic required to resolve the query in terms of easy-to-optimize relational operations. The tree encodes each relational operator in the query as an internal node (i.e., a projection (Π), selection (σ), or join operator) and each OmniTable and each Generator in the query as a separate leaf node. SteamDrill recursively decomposes each view into the relational logic that generates them until the tree is comprised entirely of relational logic and base tables (§3.1). A directed edge from node n_1 to node n_2 in the tree identifies that the operator n_2 consumes the output of n_1.

Figure 6 is the relational tree produced by SteamDrill for Listing 7. SteamDrill contains the internal logic of the `Insts(ot)` and `DefinedMemory(ot)` views (shown as dotted rectangles). The tree contains the logic of the `Insts(ot)` view: a **`Join`** between an `OmniTable` and `InstructionDefs`, a `Generator` containing metadata about the instructions defined in binaries used by an `OmniTable`.

The tree includes the internal logic of `DefinedMemory(ot)` and, recursively, all of the derived views comprising `DefinedMemory(ot)`. The tree contains the `DefinedMemory(ot)` logic (Listing 4): a **`Join`** between two `Funcs(ot)` views, one for executions of malloc (`Funcs(ot)` as m) and one for executions of free (`Funcs(ot)` as f). The tree contains the logic of each `Funcs(ot)` view (§3.2 and Listing 6): a **`StackJoin`** that combines `fentry(ot)` and `fexit(ot)`, relations over the entry and exit to each function in the `OmniTable`. Finally, the tree contains the logic for each `fentry(ot)` and `fexit(ot)`: a **`Join`** between an `OmniTable` and `FuncDefs`. Note, the tree includes the `fentry(ot)` of malloc and `fexit(ot)` of free even though the query does not use their output; during planning, SteamDrill determines that the query does not use the views and prunes them.

4.2 Planning

SteamDrill performs two tasks during planning. During logical planning, the system optimizes the relational tree using standard optimizations (e.g., predicate push-down) and determines the join order and join algorithm for each join in the tree using `OmniTable`-specific strategies (§4.2.1). The most crucial task in logical planning is determining the join order and algorithms for the query, since the join order and algorithms imply the partial order in which SteamDrill will materialize the `OmniTable` nodes contained in the query. SteamDrill supports two join algorithms: merge joins, which operate over two fully realized relations, and block-nested-loop joins (loop joins), which first calculate the left relation and use the left relation's output to limit right relation materialization.

During physical planning, SteamDrill produces a staged execution plan, which uses the join order and algorithms assigned during logical planning to assign each operator in the tree to a stage. In particular, physical planning assigns the children of merge joins to the same stage (so SteamDrill materializes them using the same replay) and assigns the right child of a loop join to the stage after the loop join's left child (so SteamDrill materializes them using different replays).

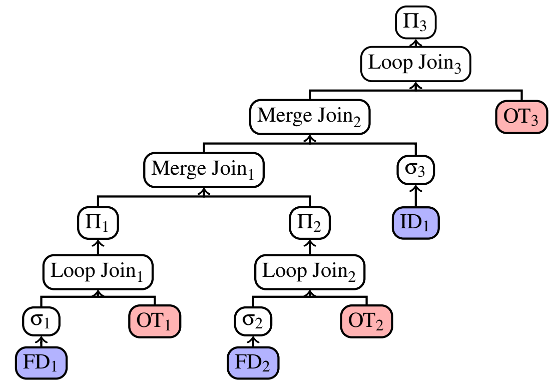

Figure 7: The relational tree for Figure 6 after logical planning.

4.2.1 Logical Planning

Traditional techniques for deciding join order and algorithms perform poorly on `OmniTable` queries for three reasons: First, similar subtrees in a relational tree have vastly different computational costs to materialize (e.g., the **`join`** subtree in the `Funcs(ot)` subtree is similar but much less computationally expensive than the **`join`** subtree in the `Insts(ot)` subtree). Second, the materialization cost of an `OmniTable` often depends on unpredictable properties of the underlying execution (e.g., it is difficult to predict the execution frequency of a particular function). Third, the enormous compute cost of materializing an `OmniTable` invalidates conventional rules.

Consequently, SteamDrill turns to a rule-based planner [1] that enables developers to encode semantic information that would be difficult or impossible for SteamDrill to deduce on its own. Each rule specifies regular-expression-like rules that pattern match subtrees of the relational tree and produce modified operators [1]. The join order and algorithm rules produce a left-deep join structure in which `OmniTable` nodes are isolated on the right-hand side of a join node (Figure 7), since these structures allow SteamDrill to perform as much filtering as possible when extracting data from an `OmniTable`. When queries join relations with different expected materialization compute costs, the rules place the less expensive relations on the left side, the more expensive relation on the right side and employ a loop join. When joining relations with the same expected materialization compute costs (e.g., two instances of `Funcs(ot)`), the rules use a merge join. Heuristically, SteamDrill expects that `Funcs(ot)` relations are less computationally expensive to materialize than `Vars(ot)` relations, which are less computationally expensive than `Insts(ot)` relations, which are less computationally expensive than `OmniTables`. Rules also encode traditional database optimizations.

Executing the relational tree in Figure 6 without logical planning would have high latency; SteamDrill would materialize the `OmniTable` five separate times! In contrast, SteamDrill's logical plan (Figure 7) uses multi-replay resolution to observe only the exit from malloc, entry to free, and load/store instructions. Moreover, the plan reduces latency by only producing data for load/stores to undefined memory as they are observed, rather than producing data for all loads/stores and performing a join to determine undefined uses afterwards.

First, SteamDrill uses traditional database optimizations (e.g., operator push-down) to push operators towards leaf nodes to (1) produce `FuncDefs` data for only malloc and free (σ_1 and σ_2), (2) produce `InstructionDefs` data only for loads and stores (σ_3) and (3) eliminate the fentry(ot) for malloc and fexit(ot) for free. The system uses loop joins for Loop Join_1 and Loop Join_2, which materialize σ_1 and σ_2 before OT_1 and OT_1 to limit `OmniTable` state to the exit of malloc and entry to free in OT_1 and OT_2, respectively. The system joins them using a Merge Join (Merge Join_1) to limit the number of replay executions that it uses. OT_3, created for the `Insts(ot)` view, is computationally expensive to materialize, so SteamDrill defers its materialization. SteamDrill uses a Merge Join (Merge Join_2) to join σ_3 and Merge Join_1, which requires a Cartesian product and violates the traditional rule that such approaches be avoided. Materializing Merge Join_2 and using a loop join (Loop Join_3) to join it with OT_3 allows SteamDrill to identify only the loads/stores to undefined memory (i.e., loads/stores that read/write an address at a time when it is not contained in `DefinedMemory(ot)`) as they are performed by the execution rather than in an expensive join afterwards. In some queries, using a loop join like Loop Join_3 enables SteamDrill to elide inspection of some instructions altogether (e.g., Listing 8).

4.2.2 Physical Planning

Next, SteamDrill converts the optimized relational tree into a staged execution plan by assigning each operator from the tree into a *stage*. Each stage corresponds to a new replay execution (§4.3). SteamDrill assigns operators to stages that follow the partial order of `OmniTable` materialization that is implied by the join order and algorithm, but uses as few stages as possible, since each stage will require the additional latency and overhead of a replay execution.

SteamDrill performs a depth-first traversal of the tree starting at the root node and maintains an integer id for the current stage, starting at 1. The system assigns leaf nodes (`OmniTable`, `Generators`) to the current stage and unary nodes (i.e., all non-join operators) to their child's stage. The system assigns merge join operators to the largest stage among the join's children. For loop join operators, SteamDrill first assigns stages to operators in the left (inexpensive) child, adds one to the current stage, assigns the loop join to the new stage and traverses the right (expensive) child.

Figure 8 shows the staged execution plan for Listing 7. SteamDrill assigns FD_1, σ_1, FD_2, σ_2, ID_1, and σ_3 to the first stage. It assigns OT_1 and OT_2 to the second stage since Loop Join_1 and Loop Join_2 indicate that OT_1 and OT_2 should be materialized after σ_1 and σ_2, respectively. SteamDrill also assigns Π_1, Π_2, Merge Join_1, and Merge Join_2 to the second stage since they inherit the largest stage of their children. The system assigns Loop Join_3, OT_3, and Π_3 to the third stage to follow the order required for Loop Join_3.

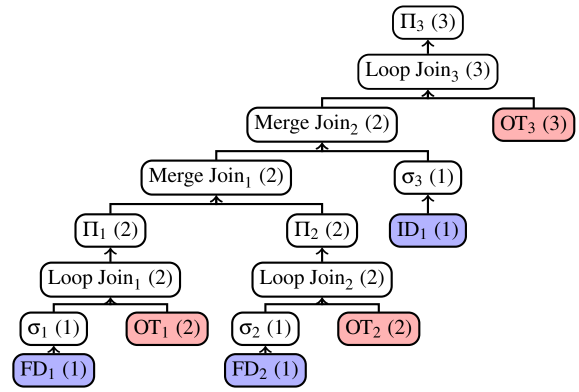

Figure 8: The staged execution plan for Figure 7. The stage of each node is shown in parentheses in the node.

4.3 Execution

Finally, SteamDrill executes the staged execution plan. For each stage, the system generates instrumentation to materialize the state needed from each `OmniTable`, materializes each `OmniTable`, and calculates each operator in the stage.

4.3.1 Instrumentation Generation

SteamDrill generates instrumentation that it will inject into a replay execution for the `OmniTables` in a stage by determining instrumentation operators for each `OmniTable` node in the stage. For each `OmniTable` node, the system gathers all stateless operators (e.g., projections (Π) and selections(σ)) that only consume data from (1) the `OmniTable` node, (2) nodes resolved in previous stages, or (3) other nodes satisfying (1) and (2). For example, the instrumentation operators for OT_1 in Figure 8 includes Π_1 and Loop Join_1. Selecting stateless operations ensures that the resulting instrumentation will be parallelizable during materialization.

Then, SteamDrill creates a cursor object for each `OmniTable` node that combines all of the node's instrumentation operations. Cursor objects contain a `filter` and an `output` clause; logically, a cursor inspects the execution instruction-by-instruction, producing the `output` whenever the `filter` is true. SteamDrill generates the filter clause of the cursor for each `OmniTable` in the stage by combining all selection (σ) and loop join instrumentation operators and generates the output clause using the output of the top-most projection (Π) instrumentation operator.

4.3.2 Materialization

Next, SteamDrill materializes `OmniTable` nodes by executing the cursor objects on top of a replay of the execution associated with the tables. It uses epoch parallelism [34, 35] to parallelize cursor evaluation. Epoch parallelism partitions a replay execution into time slices, called epochs. It assigns each epoch to a separate core in a compute cluster and uses checkpoints, generated during recording, so that each core executes each cursor over only its assigned epoch.

However, naive cursor evaluation (i.e., instrumenting every instruction) imposes a many orders of magnitude slowdown. So, SteamDrill analyzes the filter clause of each cursor to identify instructions at which the system can elide cursor evaluation to optimize performance. For example, SteamDrill identifies that the cursors in the second stage of Figure 8 only need to be evaluated at malloc and free and removes all other cursor evaluations. Our prototype identifies these optimizations by finding comparisons to the program counter.

Additionally, SteamDrill calculates operators in the stage that were not assigned as instrumentation operators for any `OmniTable` node (e.g., Merge Join_1 and Merge Join_2 in Figure 8). SteamDrill uses existing algorithms to calculate merge join and aggregation operators [1, 16]. Additionally, it executes the program associated with each `Generator` in the stage to calculate `Generator` operators.

5 Implementation

We implement our SteamDrill prototype on top of Spark [1] and Arnold [10]; below, we describe its key components.

Spark SQL. Our prototype introduces new relational operators and base tables for `OmniTables` and `Generators`. We added support for block-nested-loop joins, stored procedures, and polymorphic columns (§3) by serializing data to and from a JSON format. We added catalyst rules for our `OmniTable`-specific join order and algorithm preferences (§4.2.1). Each rule required 25 lines of code, so we expect that developers will be able to easily add custom rules as needed for their debugging workflows.

Instrumentation. Efficient cursor instrumentation plays a vital role in our prototype's performance. Debugging tools often use dynamic instrumentation frameworks (e.g., PIN [23]), which are a scalability bottleneck when SteamDrill parallelizes the replay execution across many cores [34]. Our prototype performs static binary instrumentation. It disassembles the application binaries and rewrites the basic blocks contained in the application to call cursors, as required for the breakpoints determined from each cursor. The system single-steps execution for cursors that do not produce breakpoints.

Time Column. The time column is a critical element of the `OmniTable` query model, but, deriving the column by counting all instructions or basic blocks would be too expensive. We observe that instructions progress from low to high, except in the case of a backwards control-flow (e.g., branch, call, or return instructions that jump to a program location at a lower address). Thus, our prototype uses the number of backwards control-flow operations as a the first element of the time column and breaks ties using the instruction pointer. Serendipitously, Intel provides deterministic performance counters for conditional branch and call instructions[2], which allow our prototype to compute the number of backwards control-flow operations by counting the number of unconditional backwards branches during execution and adding the value of these performance counters.

6 Evaluation

In this section, we evaluate the `OmniTable` query model and SteamDrill by answering the following questions: "Does the `OmniTable` query model improve upon existing debugging interfaces?", "Does SteamDrill accelerate debugging questions?", and "How do SteamDrill design decisions impact query performance?".

We perform 5 detailed case studies of how a developer could use an `OmniTable` and SteamDrill to solve real-world bugs from open-source servers (§6.1) from which we derive 14 debugging questions. We implement the debugging questions using `OmniTable` queries and `gdb`'s python bindings, which provide a python interface for traditional `gdb` features (e.g., breakpoints and backtraces). We compare the complexity of the 14 `OmniTable` queries and `gdb` scripts using metrics from the software engineering community (§6.2). We deploy SteamDrill on a CloudLab [37] cluster of 8 r320 machines (8-core Xeon E5-2450 2.1 GHz processor, 16 GB Ram, 10 Gbps NIC) to evaluate the performance for 3 representatives from the original 14 debugging questions (§6.3). We calculate the latency results below as the average over 10 trials and include 95% confidence intervals.

6.1 Case Studies

We performed 5 detailed case studies by identifying the debugging questions that a developer would ask when solving real-world bugs. We choose notoriously difficult bugs including livelock, intermittent performance problems, and atomicity violations (on average, the bugs in our study took 159 days from being opened to the commit that fixed the bug). We choose case studies from popular open-source applications: redis, Memcached, Apache, and Sqlite. The redis 4323 case study is described in §2; below, we describe case studies for debugging a livelock [28] and atomicity violation [27] in Memcached. We omit a description of a performance degradation in Apache [6] and a segmentation fault in Sqlite [41].

The case studies illustrate the benefits of the `OmniTable` query model along two key dimensions: first, the all-inclusive

[2] Note that most performance counters are not deterministic

```
1 Select f.Name, Count(*)
2 From Insts(ot) as i PrevJoin Funcs(ot) as f
3   On i.time, f.entryTime, i.thread=f.thread
4 Where f.exitTime=NULL
```

Listing 8: The First query for Memcached 271.

```
Select eip, True, False
From Vars(ot)
Where name="status"
Pivot Count() in (True, False)
```

Listing 9: The second query for Memcached 127.

state exposed by the table offers a powerful window into an execution's behavior. Second, SQL aggregations provide a powerful tool for summarizing and comparing program state. These features are particularly powerful when used in tandem. For example, in Memcached 271, the developer identifies the function that contains a livelock by counting the number of instructions executed by the functions left on the call stack at the end of the execution. This logic cannot be expressed in existing high-level debugging tools and is very complex when expressed using procedural tools such as `gdb`.

Memcached 271. In this case study, a developer observes livelock in the Memcached key-value store [28]. Livelock is notoriously difficult to diagnose since a developer needs to identify the cause of a missing property: forward progress [33].

In contrast, the `OmniTable` model allows the developer to succinctly track millions of execution events and use aggregations to identify anomalous execution state. Their first query, shown in Listing 8, identifies which function contains the livelock by counting the number of instructions executed during each function on each thread's call stack at program termination. The query matches each executed instruction with the most recent function called on the same thread to determine which function contained the instruction (Lines 2 and 3). It counts how many instructions were executed (Line 1) by each function that did not return (Line 4).

The output identifies a single function with a high number of branches. The function traverses a linked-list, which the developer suspects is corrupted. The developer's second query counts how many times each function that updates the linked list is called with every possible function argument value and shows a single anomalous call to free a linked-list item in which the item is still resident in the linked-list. Memcached reference counts linked-list items, so the developer's third and final query tracks all reference count updates and identifies an overflow that leads to the erroneous free of the item.

Memcached 127. This case study involves an atomicity violation in Memcached. An integer stored in the cache has the wrong value after all updates, which is challenging to debug since the developer does not know which program state to track or when to track it. Atomicity violation tools [32] use heuristics and may misidentify the root cause of the bug.

Query	Lines		Nodes		Halstead (s)	
	gdb	OT	gdb	OT	gdb	OT
Apache 60956 Q1	20	6	94	54	518	263
Apache 60956 Q2	30	9	113	122	989	1350
Memcached 127 Q1	7	4	48	39	147	82
Memcached 127 Q2	11	4	74	26	518	38
Memcached 271 Q1	35	3	149	26	1471	62
Memcached 271 Q2	12	4	69	23	397	34
memcached 271 Q3	10	3	45	26	140	39
redis 4323 Q1	17	3	74	23	529	45
redis 4323 Q2	7	3	24	31	35	65
redis 4323 Q3	22	3	113	83	930	757
redis 4323 Q4	33	5	147	112	1620	1033
redis 4323 Q5	7	3	19	19	17	19
sqlite 787fa71 Q1	22	10	110	77	911	520
sqlite 787fa71 Q2	41	8	151	96	1489	684
Average	20	5	88	54	694	357

Table 1: Lines, Nodes, and Halstead Complexity for debugging questions expressed using `OmniTable` queries (OT) and `gdb` python scripts (`gdb`).

The `OmniTable` model, particularly SQL aggregations, provide a powerful tool for comparing the state of their program at many points-in-time to identify anomalous program state. The developer first isolates the module that contains the error. In particular, they determine if the bug arises when initially parsing requests or when processing them by using a `count` aggregate to count the number of times the function at the boundary between parsing and processing is called with each possible set of arguments. The query shows that the problem arises when processing requests.

The processing code maintains a boolean variable, `valid`, that tracks the validity of a global pointer used by the code. The developer's second query, shown in Listing 9, identifies how often `valid` is set to `true` and `false` during each of the instruction within the processing logic. It uses a **`Pivot`** operator to produce a row for each instruction and show the number of times `valid` is set to `true` and `false` across all executions of the instruction. The second query identifies a few instructions at which status has an anomalous state. The anomalous instructions do not modify the status, so the developer concludes that another thread must modify the status and identifies a mistake in the processing logic's use of a mutex.

6.2 Complexity

We implemented the 14 debugging queries from our 5 case studies using `OmniTable` queries and implemented equivalent logic using `gdb` python scripts. Qualitatively, we observe that `OmniTable` queries are less complex due SQL aggregations, the all-inclusive nature of an `OmniTable`, and the structured approach provided by high-level views: `OmniTable` queries usually involve an aggregation after joining a few high level views, whereas imperative debugging scripts regularly use multi-dimensional data-structures to track state, nested control-flow to implement aggregations, and complex

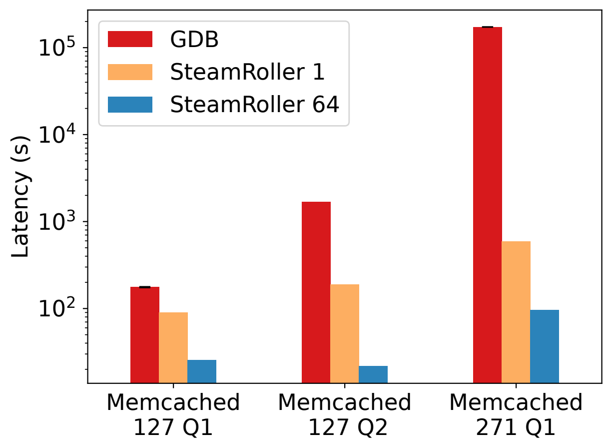

Figure 9: SteamDrill query latency on a single core and on 64 cores compared to `gdb` script latency (which is sequential). Y-axis is log-scale.

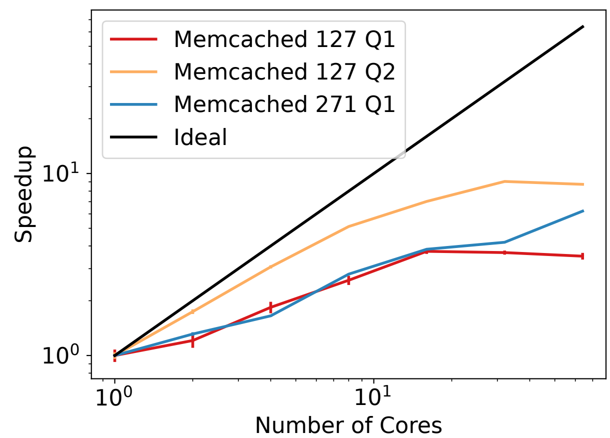

Figure 10: SteamDrill scalability. Shows number of cores on the x-axis vs. speedup on the y-axis; both axes are log-scale.

regular expressions to identify instrumentation points. We measure complexity of each `OmniTable` query and `gdb` script using three software engineering metrics: the number of lines of code, the number of terms in the abstract syntax tree (AST), and the Halstead complexity, which estimates the amount of time it would take to correctly produce the query or script using properties of the AST [18]. We included the definition of user-defined views (e.g., `DefinedMemory(ot)`) into the `OmniTable` queries that use them, so our results are an upper-bound on `OmniTable` query complexity.

Table 1 shows the results, indicating that `OmniTable` queries are less complex than `gdb` scripts. By geometric mean, `OmniTable` queries require 3.74 times fewer lines, 1.70 times fewer nodes, and 2.75 times less estimated time to develop than `gdb` scripts. There are only three queries that are more complex when expressed using the `OmniTable` model, the second and fifth redis 4323 queries, and the second Apache 60956 query. The two redis queries are small for both representations. The second Apache query suffers from the lack of kernel state in an `OmniTable`. The query identifies all blocking file descriptors, which requires substantial logic to track all function calls in the `OmniTable` model, but can be calculated in `gdb` using `fcntl`. Extending the `OmniTable` to include kernel state would reduce the complexity.

6.3 Query Latency

We evaluate the latency of `OmniTable` queries and `gdb` scripts for 3 representative queries from our case studies. We choose queries that use all of the high level views in our prototype (i.e., `Funcs(ot)`, `Vars(ot)`, and `Insts(ot)`) and offer a wide range of performance on current tools, from ~22 minutes to ~2 days. Figure 9 shows the latency of each debugging question evaluated using `gdb`, SteamDrill with a single core, and SteamDrill with 64 cores, with latency plotted on a log-scale. We executed Memcached 271 Q1 for 48 hours before killing the program and report its latency as 48 hours.

SteamDrill is significantly faster than `gdb`. SteamDrill query latency is between 2 and 290 times (with a geometric mean of 17) faster than `gdb` latency when using a single core, and between 6.9 and 1809 (with a geometric mean of 99) times faster than `gdb` latency when using 64 cores.

6.4 Optimizations

Next, we evaluate the impact of three optimizations on SteamDrill's latency: parallelization, multi-replay resolution, and performance-counters.

Scalability. We evaluate the query latency of SteamDrill queries when using 1–64 cores; Figure 10 shows the speedup on a log-log scale. SteamDrill queries are 10.5 times faster using 64 cores than when run sequentially. Importantly, whereas prior parallelization efforts require the developer to substantially redesign their debugging code [30, 34, 35, 38, 46], the parallelized and sequential `OmniTable` queries are identical. The current scalability bottlenecks are caused by high initialization and serialization cost in Spark and the high cost of compiling cursors.

Multi-Replay Resolution. We evaluate the impact of multi-replay query resolution on the Memcached 271 Q2 query. We calculate the query latency when using two rounds of replay (the approach chosen by the SteamDrill planner) and when using a single round of replay on 64 cores. SteamDrill is 3.6 times faster when using multi-replay resolution.

Performance Counters. We evaluate the impact of using performance counters to accelerate the calculation of the `time` column in an `OmniTable`. We executed the 3 queries with and

Tool	Model	Observations	Aggregates
Execution Mining [20]	Stream	All	No
Fay [12]	Stream	Partial	Partial
Pivot Tracing [24]	Relational	Log-Based	Partial
G2 [17]	Graph	Log-Based	Manual
PQL [25]	Stream	Partial	No
PTQL [14]	Relational	Partial	No
EndoScope [7]	Stream	Partial	Yes
EBBA [5]	Stream	Log-Based	No
TQuel [40]	Relational	Log-Based	Partial
`OmniTable`	Relational	Everything	Yes

Table 2: Feature comparison of high-level debugging tools .

without using performance counters (when disabled, SteamDrill instruments all `jump`, `call`, and `return` instructions) on 64 cores. The performance counter optimization accelerates query latency by a factor of 1.6.

7 Related Work

The `OmniTable` query model is the first debugging model that exposes all application state as a single entity and enables succinct observations via a high-level declarative language. Below we describe work related to high-level languages for debugging, using deterministic replay for debugging, and applying optimizations to accelerate debugging.

Existing systems support high-level debugging languages to reduce programming complexity; Table 2 illustrates the limitations of prior work compared to the `OmniTable` model. Execution Mining [20], PQL [25], EBBA [5] and EndoScope [7] expose a time-stream model of execution, which complicates debugging since it is difficult to summarize data over time (e.g., these tools cannot express the `Funcs(ot)` view since it contains execution data from multiple points-in-time). Other systems limit visibility of execution state: Fay [12], PQL [25], EBBA [5], EndoScope [7], and PTQL [14] expose partial program state consisting of only the function calls or global variables values in an execution. Pivot Tracing [24], G2 [17], EBBA [5], and TQuel [40] require manual instrumentation to enable observations, which essentially amounts to supporting queries over software logs. Finally, many tools provide no, or very few, aggregates [14, 20, 25]; G2 [17] supports aggregates but requires that they be expressed in terms of a graph processing language.

Many `OmniTable` queries compare correct execution behavior to incorrect execution behavior, similar to statistical debugging approaches [22]. There are two key differences (1) statistical bug isolation requires observing many correct and incorrect executions to come to a statistical verdict, whereas developers can often get a "sense" for correctness using an `OmniTable` query with fewer examples and (2) statistical debugging approaches hard code the values that they compare (e.g., function argument values), whereas developers can customize `OmniTable` queries to use program constructs best suited to their applications.

Many systems have noted that deterministic replay can be a great help when debugging software problems [8, 13, 19, 31, 42, 45]. Such systems enable a debugging program to explore an execution's time-sequence in reverse, but retain a procedural interface.

Recently, JetStream [34] and Sledgehammer [35] use deterministic replay as a vehicle for parallelizing debugging, which our prototype uses to accelerate `OmniTable` queries. However, these tools support procedural debugging models, similar to `gdb`, and consequently suffer from the programming complexity.

Existing tools do not decouple debugging logic's execution from the original execution to optimize query latency. PARTICLE [14], Fay [12], Pivot Tracing [24] and PMSS [21] reduce the debugging performance overhead using traditional SQL optimizations (e.g., predicate push-down). However, these tools add instrumentation to the program and re-execute it to recreate the bug, which tightly couples the execution of debugging and the original execution and increases performance overhead. Additionally, by inspecting new executions, these systems are cannot perform all SteamDrill performance optimizations, particularly multi-replay query resolution.

8 Conclusion

In this paper, we propose the `OmniTable` query model, a new debugging paradigm that reduces the programming complexity and performance overhead of debugging without restricting the execution state that a developer can observe. We show that the query model simplifies debugging questions compared to existing state-of-the-art tools by performing case studies of bugs reported in popular open-source software. Unfortunately, an `OmniTable`, the key abstraction in the model, cannot be stored or calculated due to its extreme size. So, our prototype, SteamDrill, implements *lazy materialization*: it delays an `OmniTable`'s calculation until a developer queries the table. It uses deterministic record and replay to store the execution associated with each `OmniTable` and then generates instrumentation and traces a new replay execution to resolve each developer query on-demand. The system uses declarative optimizations, debugging optimizations, and a novel multi-replay strategy to accelerate debugging queries by an order of magnitude compared to state-of-the-art tools.

9 Acknowledgements

We would like to thank our shepherd, Ding Yuan, and the anonymous reviewers for their insightful comments. The work was supported by the National Science Foundation under grant DGE-1256260.

References

[1] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, New York, NY, USA, 2015. Association for Computing Machinery.

[2] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. System r: relational approach to database management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[3] Thomas Ball and James R Larus. Efficient path profiling. In *Proceedings of the 29thACM/IEE international symposium on Microarchitecture*, pages 46–57. IEEE Computer Society, 1996.

[4] Subarno Banerjee, David Devecsery, Peter M Chen, and Satish Narayanasamy. Iodine: fast dynamic taint tracking using rollback-free optimistic hybrid analysis. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 490–504. IEEE, 2019.

[5] Peter C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems*, 13(1):1–31, February 1995.

[6] Bug 60956. https://bz.apache.org/bugzilla/show_bug.cgi?id=60956.

[7] Alvin Cheung and Samuel Madden. Performance profiling with endoscope, an acquisitional software monitoring framework. *Proc. VLDB Endow.*, 1(1):42–53, aug 2008.

[8] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. Rept: Reverse debugging of failures in deployed software. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation*, OSDI'18, pages 17–32, 2018.

[9] David Devecsery, Peter M Chen, Jason Flinn, and Satish Narayanasamy. Optimistic hybrid analysis: Accelerating dynamic analysis through predicated static analysis. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 348–362, 2018.

[10] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. Eidetic systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation*, Broomfield, CO, October 2014.

[11] Marc Eisenstadt. My hairiest bug war stories. *Commun. ACM*, 40(4):30–37, apr 1997.

[12] Ulfar Erlingsson, Marcus Peinado, Simon Peter, and Mihai Budiu. Fay: Extensible distributed tracing from kernels to clusters. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 311–326, October 2011.

[13] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global comprehension for distributed replay. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'07, pages 21–21, 2007.

[14] Simon F. Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 385–402, New York, NY, USA, 2005. ACM.

[15] Google sanitizers issues. https://github.com/google/sanitizers/issues?q=is%3Aissue+is%3Aopen+ASAN.

[16] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and subtotals. *Data mining and knowledge discovery*, 1(1):29–53, 1997.

[17] Zhenyu Guo, Haoxiang Lin, Mao Yang, Dong Zhou, Fan Long, Chaoqiang Deng, Changshu Liu, and Lidong Zhou. G2: A graph processing system for diagnosing distributed systems. In *Proceedings of the 2011 USENIX Annual Technical Conference*, 2011.

[18] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., USA, 1977.

[19] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 1–15, April 2005.

[20] Geoffrey Lefebvre, Brendan Cully, Christopher Head, Mark Spear, Norm Hutchinson, Mike Feeley, and Andrew Warfield. Execution Mining. In *Proceedings of*

the 2012 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), March 2012.

[21] Yingsha Liao and Donald Cohen. A specificational approach to high level program monitoring and measuring. *IEEE Transactions on Software Engineering*, 18(11):969–978, 1992.

[22] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, PLDI '05, page 15–26, New York, NY, USA, 2005. Association for Computing Machinery.

[23] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[24] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, 2015.

[25] Michael Martin, Benjamin Livshits, and Monica S Lam. Finding application errors and security flaws using pql: a program query language. *ACM SIGPLAN Notices*, 40(10):365–383, 2005.

[26] Steve McConnell. *Code complete*. Pearson Education, 2004.

[27] memcached - issue #127. https://code.google.com/archive/p/memcached/issues/127.

[28] Memcached gets a dead loop in func assoc_find. https://github.com/memcached/memcached/issues/271.

[29] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, San Diego, CA, June 2007.

[30] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, Seattle, WA, March 2008.

[31] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability. In *Proceedings of the 2017 USENIX Annual Technical Conference*, Santa Clara, CA, July 2017.

[32] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–36, 2009.

[33] Rahul Patil and Boby George. Tools and techniques to identify concurrency issues. hhttps://docs.microsoft.com/en-us/archive/msdn-magazine/2008/june/tools-and-techniques-to-identify-concurrency-issue.

[34] Andrew Quinn, David Devecsery, Peter M. Chen, and Jason Flinn. JetStream: Cluster-scale parallelization of information flow queries. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation*, Savannah, GA, November 2016.

[35] Andrew Quinn, Jason Flinn, and Michael Cafarella. Sledgehammer: Cluster-fueled debugging. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation*, pages 545–560, 2018.

[36] Redis 4.x lazyfree: memory leak may happen when free slowlog entry #4323. https://github.com/redis/redis/issues/4323.

[37] Robert Ricci, Eric Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), December 2014.

[38] Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. Parallelizing dynamic information flow tracking. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2008.

[39] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, December 2009.

[40] Richard Snodgrass. Monitoring in a software development environment: A relational approach. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 124–131, New York, NY, USA, 1984. ACM.

[41] Assertion fault when multi-use subquery implemented by co-routine. https://www.sqlite.org/src/tktview/787fa71.

[42] Sudarshan Srinivasan, Christopher Andrews, Srikanth Kandula, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 29–44, Boston, MA, June 2004.

[43] Michael Stonebraker and Lawrence A Rowe. The design of postgres. *ACM Sigmod Record*, 15(2):340–355, 1986.

[44] Kde bugtracking system. `https://bugs.kde.org/buglist.cgi?component=memcheck&product=valgrind&resolution=---`.

[45] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, Cascais, Portugal, October 2011.

[46] Benjamin Wester, David Devescery, Peter M. Chen Jason Flinn, and Satish Narayanasamy. Parallelizing data race detection. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, Houston, TX, March 2013.

[47] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, pages 15–28, San Jose, CA, April 2012. USENIX Association.