You can't debug what you can't see: Expanding observability with the OmniTable

Andrew Quinn, Jason Flinn, Michael Cafarella
University of Michigan

Abstract

The effectiveness of a debugging tool is fundamentally limited by what program state it can observe. Yet, for performance reasons, all current debugging tools restrict the program state that can be observed in some way. For example, tools like heap analysis restrict *what* can be observed (i.e., only global variables) and tools like core dump analysis restrict *when* observations may be made (i.e., only on program termination). Other tools effectively limit the scope of observation by requiring developers to specify what and when observations will be made before execution (e.g., logging) or during an execution (e.g., gdb).

We propose a new abstraction for debugging, called an `OmniTable`, that logically exposes unrestricted access to *all* program state at *all* points in an execution to developers. The `OmniTable` represents a program execution as a database-style table. Developers inspect the `OmniTable` using a familiar declarative query language: SQL. SQL simplifies the observation and analysis of large, complex execution state. Iterative queries are inherently consistent since they operate over the same logical table.

Clearly, materializing the `OmniTable` for even a simple program is infeasible due to storage and processing overheads. Thus, our prototype, SteamDrill, selectively materializes only the regions of the `OmniTable` required to answer each query by using deterministic record and replay to reproduce the execution and dynamic instrumentation to extract needed state. By expressing debugging queries with relational logic, SteamDrill leverages proven optimizations such as query optimization and caching. In addition, decomposition into relational logic allows a query to be executed via repeated replays, each replay extracting information needed by the next, which can often be more efficient than extracting all information during a single execution.

CCS Concepts • **Software and its engineering → Software testing and debugging**; • **Computer systems organization** → *Reliability*.

ACM Reference Format:
Andrew Quinn, Jason Flinn, Michael Cafarella. 2019. You can't debug what you can't see: Expanding observability with the OmniTable. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3317550.3321428

1 Introduction

"Everyone knows that debugging is twice as hard as writing a program in the first place. So, if you're as clever as you can be when you write it, how will you ever debug it?"

The Elements of Programming Style– Kernighan and Plauer

Studies estimate that developers spend as much as 50% of their time debugging their software [15] and companies spend upwards of 100 billion dollars on debugging each year [22]. When debugging, developers use tools to inspect their software as it executes. More powerful tools result in more productive debugging, so the community has invested tremendous effort to create new and powerful debugging tools.

Yet, current debugging tools are limited by the observations that they allow developers to make. For example, tools like heap analysis limit *what* state can be observed and tools like core dump analysis limit *when* observation can occur. Other tools effectively limit observations by forcing debuggers to commit to an observation before execution (e.g., logging) or during execution (e.g., gdb). Many tools effectively limit the size and complexity of state that can be observed because of cumbersome specification languages (e.g., pin/valgrind).

A new method is needed that removes these limitations on observation. An ideal method satisfies three criteria. First, an ideal method provides *visibility*; a developer should be able to observe all state of an execution at any point in time during the execution. Second, an ideal method provides *repeatability*; a developer should be able to reinspect the same execution many times. Finally, an ideal method provides *expressibility*; a developer should be able to easily specify even large and complex observations of a program execution.

We propose a new abstraction called the `OmniTable` to meet these ideal characteristics. The `OmniTable` is a database-style logical table; as a program executes, all user-level state (registers, memory values, etc.) is extracted at all points in time to populate the table. To observe an execution, developers write declarative queries over the OmniTable. The OmniTable provides visibility, since all state of the execution at every point in time can be queried; repeatability, since developers can ask multiple queries over the same logical table; and expressibility, since developers can easily specify complex observations using declarative queries.

However, the materialized OmniTable for even a simple execution is infeasible to store, since it often reaches petabytes in size. To reduce the storage burden, our prototype, SteamDrill, uses deterministic record and replay to store the execution associated with each OmniTable instead of storing the materialized OmniTable. SteamDrill maintains repeatability by replaying the computation of the execution to materialize OmniTable data.

Materializing the entire OmniTable for each query would impose high query latency. So, SteamDrill selectively materializes only the portions of the OmniTable needed to answer a developer query by using a query planning approach. Intuitively, instead of executing a query over an existing OmniTable, SteamDrill uses the query to filter the data from the OmniTable that is materialized in the first place. SteamDrill uses a novel approach that employes multiple rounds of replay to materialize OmniTable data instead of materializing all data in a single replay. By delaying the materialization of some OmniTable data, this approach uses data that is inexpensive to materialize (e.g., data at function granularity) to filter the expensive to materialize data (e.g., data at instruction granularity) needed to answer the query. In addition, the system uses traditional query optimizations [4, 13, 19] and caching.

SteamDrill materializes the OmniTable data needed for a query by instrumenting the replay to extract the required data. The multi-round replay approach taken by SteamDrill requires the ability to extract state with low latency. To meet these needs, SteamDrill turns to recent work on parallelizing software inspection across compute clusters [18].

2 Query Model

The SteamDrill query model allows developers to observe their software using declarative queries written in an extended SQL language. Compared with traditional debugging tools such as logging or gdb, SQL provides a simplified mechanism for reasoning about large amounts of process state across large regions of an execution. Queries can inspect any state of an execution at any instruction in the execution using the OmniTable. To make the OmniTable easier to consume, SteamDrill provides a number of high-level views over the table, such as the trace of functions or instructions executed in the program. Over time, OmniTable users will contribute new views to SteamDrill, creating an ecosystem of both general-purpose and application-specific views to be used as-is, or with slight modifications, during future debugging sessions. § 2.1 outlines the components of the query model; § 2.2 shows how the model is used to create a high-level view.

2.1 Model Components

The SteamDrill model is made up of four main components.

OmniTable The OmniTable is a database-style table that logically exposes all state in an execution at all instructions in the execution; Figure 1 shows an example of the table for a short execution. For each instruction in the execution, the OmniTable contains a monotonically increasing logical-time called the count, the thread that executed the row's instruction, and the value of all registers and memory addresses. The count field is not an exact instruction count, but does guarantee an ordering of events in the OmniTable; in multi-threaded programs, the count field reflects a total ordering that is consistent with the partial ordering of the execution.

Static Tables Static tables allow a developer to incorporate data from outside the OmniTable into their debugging queries, which is critical for constructing high-level views over the OmniTable. Static tables are data objects and often translate from OmniTable state into a high-level abstraction. Developers specify static tables by providing SteamDrill with a program and program inputs that produce the static table as output. SteamDrill has a number of static tables built-in, such as StaticFunctions, a static table that contains an entry for each function defined in a binary.

Tracing Functions In some cases it is challenging or impossible to express an observation using relational logic, such as determining all elements in a tree data-structure. For these tasks, SteamDrill supports tracing functions, which are similar to database-style user-defined-functions. Tracing functions operate in the context of a single row in the OmniTable. Memory values that are read by the function are populated with the corresponding value from the OmniTable, other columns of the table are accessed by requesting inputs from SteamDrill. For example, to track the elements in a tree over time, a developer would write a tracing function walk(root) that produces a string containing information for each node in the tree. If the tree is rooted at root, the developer specifies:

```
Select walk(OmniTable.root) From OmniTable
```

Operators Developers specify queries using SQL-style Select, From, Where syntax. SteamDrill supports standard relational operators including join operators, aggregations, and comparison operators. Additionally, SteamDrill provides two new operators that simplify queries that make observations about the ordering of events in the OmniTable. For example, a developer may use NextJoin to determine the next function executed by a thread or track the next access to a

Metadata		Registers				Memory			
count	**thread**	**eip**	**eax**	**ebx**	**...**	**0x0**	**0x1**	**...**	**0xffffffff**
1	100	0x80000000	1	1	...	1	1	...	1
2	100	0x80000004	1	1	...	1	1	...	1
⋮									
1000	100	0x80000064	1	1	...	1	1	...	1

Figure 1. An example `OmniTable` for a short execution.

shared variable. Specifying `NextJoin(col1, col2)` joins two tables such that items in the first table, ordered by the column `col1`, are joined to the next item in the second table, ordered by the column `col2`. The `PrevJoin` operator does the opposite. These operators are implemented using standard relational logic and are provided to simplify the construction of these ordering-based queries.

2.2 High-Level Views

SteamDrill provides a number of high-level views over the `OmniTable` that provide useful abstractions for debugging. Example high-level views include `Functions`, a view containing data about all functions executed, and `Instructions`, a view containing data about all instructions executed. Over time, developers will contribute new high-level views to SteamDrill to create an ecosystem of debugging views. With this ecosystem, we expect it to be rare for a developer to write a new `OmniTable` query entirely from scratch. Instead, debugging queries will primarily use existing high-level views to debug, occasionally making slight modifications to specialize an existing view to their application.

In this section, we describe a query to construct `Functions`, the trace of functions executed in an `OmniTable`. The query joins the `OmniTable` to `StaticFunctions`, a static table that contains the function name, starting instruction, return instructions, and tracing functions that produce the argument values, return value, and caller for all functions in a binary. The `functions` view is constructed as three queries; the first inspects data at the start of a function call:

```
Q1: Select count as strt, name, arg(esp), caller(esp), thread
    From OmniTable, StaticFunctions
    Where eip == startIP
```

The second inspects data at the end of each function invocation:

```
Q2: Select count as end, name, rtn(esp) as rtn
    From OmniTable, StaticFunctions
    Where eip in rtnIPs
```

Finally, `Functions` uses the `NextJoin` operator to combine the data from the start and end of each function:

```
Q3: Select * From Q1 NextJoin(st, end) Q2
    Where Q1.name == Q2.name
```

3 Usage Scenarios

In this section, we describe how a developer can use the SteamDrill data model to debug their software. First, we show how a developer can use SteamDrill to detect a common class of software bugs: use-after-free bugs [11]. Then, we show how SteamDrill allows developers to easily make iterative observations which are critical for understanding the root cause of software bugs. Finally, we show how SteamDrill allows developers to simultaneously reason about high-level application-specific abstractions and low-level systems abstractions over an execution.

3.1 Use-After-Free Detection

In a use-after-free bug, a program uses a pointer that was previously freed, which can lead to software crashes, data corruption or even arbitrary code execution.

To find use-after-free bugs, a developer tracks the intervals of time during which each address is free and all memory operations during these intervals to identify any accesses to freed memory. Logging and gdb provide poor interfaces for tracking large assembly-level state, so dynamic instrumentation tools (e.g., pin and valgrind) are generally used. These tools require writing complex observation code that must be hand-optimized to provide reasonable performance.

To identify use-after-free bugs, the developer constructs two queries. First, the developer identifies the interval of time during which each address has been freed but not yet reallocated. She uses the SteamDrill built-in view, `Functions`, to match each call to `free` with the next call to `malloc` that returns the same address. The developer uses the `NextJoin` operation to order the function invocations; she specifies arguments `end` and `start` to identify the columns to use for ordering the function calls. The developer uses a `Left NextJoin` so that addresses that are never reallocated appear in the relation. Her query is shown below:

```
Q4: Select f.arg('ptr') as p, f.caller, f.end as strt, m.strt as end
    From Functions f('free')
    Left NextJoin(end, start) Functions m('malloc')
    Where f.end < m.start And f.arg('ptr') >= m.rtn
      And f.arg('ptr') < m.rtn + m.arg('size')
```

Next, the developer identifies use-after-free bugs by identifying instructions that use an address during the interval identified by above relation. Her query produces the instruction that uses the freed memory, the caller of the `free` function call that freed the memory, and logical times for both the use and the `free`. She uses a built-in view provided by SteamDrill, `Instructions`, to inspect the memory addresses read by each instruction:

Q5: **Select** i.eip, i.count, f.caller, f.end
From Instructions i, Q4 f
Where f.p **in** i.AddrRead **And** i.time >= f.strt
And (i.time < f.end **Or** f.end == **NULL**)

3.2 Iterative Observations

After detecting a bug, a developer will often have to make additional observations over her program in order to understand the root cause. Subsequent observations often modify or enhance prior queries in order to build the context that leads to the error. The repeatability and expressibility offered by the `OmniTable` simplifies the process of constructing subsequent queries.

For example, a pbzip2 use-after-free bug was caused by an ordering violation where one thread frees a shared variable before the other thread has finished using the variable [24]. Existing tools help the developer find the use-after-free bug, but to understand the ordering violation the developer will either have to modify a low-level memory checking tool or manually correlate output from multiple tools (e.g., the memory checker and logging).

SteamDrill can simplify the diagnosis of this pbzip2 bug. Using Q5 from § 3.1, the developer identifies a few use-after-free bugs which all occur after the same call to `free`. This is surprising: the call to `free` occurs during cleanup which should happen after the last use of the shared variable. The developer determines that the use and call to `free` originate from different threads by modifying the query. In particular, she adds `f.thread` to the **Select** clause in Q3 and both `f.thread` and `i.thread` to the **Select** clause in Q4. The thread that calls free should call `pthread_join` on all the threads that use the shared variable; the developer confirms that her program is missing one of these calls by querying for all calls to `pthread_join` made by the main thread:

Q6: **Select Distinct** * **From** Functions, Q5
where Functions.thread == Q5.free_thread
AND Functions.name == "pthread_join"

3.3 Correlating Observations

Using the SteamDrill data model, developers can succinctly reason about multiple abstractions of program behavior; each abstraction can be expressed as a view over the `OmniTable` which can be merged and aggregated using SQL. For example, the use-after-free query (Q4) aggregates observations from both the instructions and the functions executed during the execution. This same mechanism allows developers to incorporate application-specific abstractions to filter and optimize their queries.

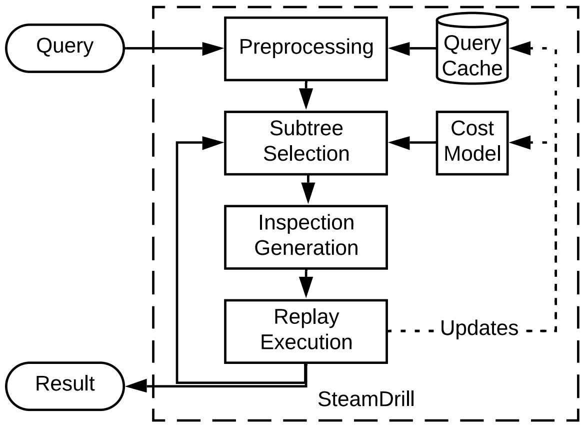

Figure 2. The design of SteamDrill.

For example, a MongoDB bug report identified a use-after-free bug that occurred when a rollback occurs because the node is stopped [1]. Current tools provide a poor interface for incorporating this type of observation, since the developer will either have to express the observation in a low-level tool (e.g., valgrind), or manually merge output from multiple tools (e.g., logging and a memory-checker). This can lead to an inefficient debugging process, since the developer struggles to use pertinent information about the bug.

In contrast, application-specific observations are straightforward to incorporate in the SteamDrill data-model: they are simply another high-level view over the `OmniTable`. For example, the MongoDB bug can be identified by searching for use-after-free bugs that occur after a rollback (a call to 'rollback') after the node has been stopped (a call to the function 'stop'):

Q7: **Select** * **From** Q5 **PrevJoin**(count, strt)
(**Select** r.strt **From** Functions s('stop')
NextJoin(end, strt) Functions r('rollback'))

4 Design

In this section, we outline how SteamDrill resolves a developer debugging query. From the developer perspective, SteamDrill executes queries to filter and select data from a fully materialized `OmniTable`. However, generating the `OmniTable` is extremely expensive. So, internally, SteamDrill uses logic from the query to filter data from the `OmniTable` before it is generated instead of afterwards.

To perform the filtering, SteamDrill first converts each query into a tree of relational algebra operations (internal nodes) over data tables (leaves). The relational tree representation decomposes a complex query into a larger number of simple operations which are easier to optimize and cache. To generate the `OmniTable` data needed for the query, SteamDrill extracts information from the replay using dynamic

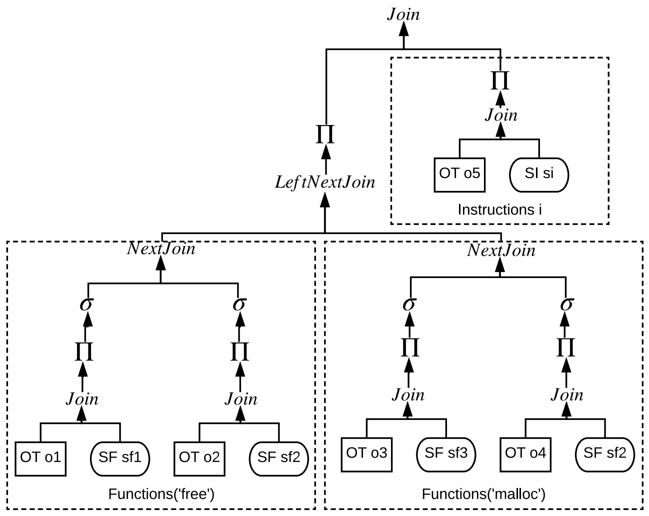

Figure 3. The tree of relational operators for the use-after-free query. Static tables are shown as ovals, `OmniTable` references are show as rectangles. Boxes with dotted lines identify subtrees for high-level views used in the query.

instrumentation. SteamDrill reduces the data extracted for a query by using multiple rounds of execution; data learned in each round of execution can be used to filter the data extracted in the subsequent rounds. This approach relies on consist state across the rounds of execution; deterministic replay satisfies this requirement. Figure 2 depicts the high level overview of the SteamDrill query processing algorithm.

4.1 Preprocessing

SteamDrill begins processing a query by converting it into a tree of relational operators over data tables; the tree encodes all data needed to resolve the query in terms of simple operations that can be optimized and cached. Figure 3 shows the relational tree for the use-after-free example in § 3. SteamDrill follows the standard database approach: selection statements become projection operators (π), joins become join operators and clauses from a where statement are converted to either selection operators (σ) or criteria for the join operator [19]. Tables in the relational tree are decoupled and may appear multiple times.

After determining the relational tree, SteamDrill resolves relations in the tree that do not depend upon `OmniTable` data. SteamDrill uses a cache to resolve any `OmniTable` regions that were materialized by previous queries; this cache reduces the work of resolving different queries over the same `OmniTable`. Next, SteamDrill calculates all static tables in the tree. SteamDrill caches static table output as an optimization.

SteamDrill finishes preprocessing by applying proven SQL planning optimizations [19], such as moving selections and projections towards the leaves of the tree. Figure 3 shows the relational tree after the preprocessing step; selection operators (σ) over the `Functions` views are pushed past the `NextJoin` operators.

4.2 Selection

After preprocessing, SteamDrill iteratively resolves the relational tree using rounds of replay. First, the system selects regions of the relational tree to resolve in the current iteration. Intuitively, SteamDrill delays generation of data from a `OmniTable` node, *o*, if resolving other regions of the tree first will filter the data needed from *o*. To determine whether to delay *o*, SteamDrill analyzes the join operators in the relational tree that depend on the node. If a join operator does not depend on any other unresolved `OmniTable` nodes, then the operator filters the data needed from *o*. If SteamDrill delays the resolution of *o* in the current iteration, additional join operators will filter *o* in subsequent iterations, reducing the data needed from *o*.

To determine the `OmniTable` nodes that should be resolved, SteamDrill walks the relational tree starting at the root. Each time SteamDrill encounters a join that depends on multiple unresolved `OmniTable` nodes, SteamDrill uses a cost model to decide which of the subtrees of the join operator to explore. If the join criteria will never filter either subtree (e.g., the $NextJoin$ operators in Figure 3), SteamDrill explores both subtrees. If filtering is possible (e.g., the top level $Join$ operator in Figure 3), the cost model determines which side to explore using a heurisitc, such as selecting the subtree that is least expensive to generate. When a path reaches a `OmniTable` node, the algorithm stops exploring the path and selects the `OmniTable` node. After selecting `OmniTable` nodes, SteamDrill adds all operators from the tree that depend only on selected nodes or previously resolved nodes.

In the Figure 3 example, the search first analyzes the top level $Join$. Since the left subtree is less expensive to calculate than the right subtree, SteamDrill explores the left subtree. At the $LeftNextJoin$ operator, SteamDrill determines that the left subtree can filter the right subtree. The following $NextJoin$ operator cannot filter either subtree, so SteamDrill explores both subtrees and selects both *o*1 and *o*2. SteamDrill then adds all operators that depend only on selected nodes, and ultimately selects to resolve the `Functions('free')` in the first iteration.

For the selection to be effective, the cost model must accurately estimate the cost of resolving a subquery. We expect subquery cost to be correlated with the portion of `OmniTable` that the subquery generates. We plan to use data from past queries in order to estimate the portion of the `OmniTable` generated by a particular subquery.

4.3 Generation

Next, SteamDrill generates the instrumentation code that it will dynamically inject into the replay to produce the selected regions of the relation tree. The SteamDrill iterative algorithm requires a low-latency dynamic instrumentation tool in order to achieve reasonable performance, so SteamDrill turns to Sledgehammer [18]. Sledgehammer is a replay-based debugging tool that accelerates program inspection by parallelizing it across a compute cluster. Using a compute cluster with thousands of cores, Sledgehammer can inspect an execution even faster than replaying the execution; inspecting even large state of an execution (e.g., all function invocations) completes in only a few seconds.

Sledgehammer inspection code is specified using two sets of functions: tracers, functions injected into the replay to inspect state, and analyzers, functions that combine the output from tracers in order to realize relationships over time. SteamDrill synthesizes a tracer for each selected `OmniTable` node. Relational operations that depend on only one `OmniTable` node reduce the data generated by the tracer. Tracing functions (§ 2) are injected into the tracer. SteamDrill specifies a tracer invocation through either a set of breakpoints or a set of watchpoints. When replay reaches a breakpoint or accesses a memory address from a watchpoint set, SteamDrill executes the associated tracer.

To materialize data that depends on multiple `OmniTable` nodes, SteamDrill uses built-in analyzers. We plan to adapt work from the database community so that SteamDrill will dynamically determine an optimal approach for each operator [7].

4.4 Execution

Lastly, SteamDrill executes the Sledgehammer inspection code. After Sledgehammer finishes, SteamDrill updates the cost mode and caches query results to reduce future `OmniTable` materialization. The caching granularity imposes a trade-off. Caching data from the `OmniTable` increases cache hits, but using cached data will likely require computation. In contrast, caching a high-level relation (e.g., `Functions` data) is more likely to be immediately usable, but decreases the cache hit rate.

5 Related Work

The `OmniTable` is the first debugging method that provides ideal visibility, repeatability and expressibility.

Prior systems provide expressibility by using high level languages for software inspection. However, prior systems limit visibility by limiting observations to specific points in the program [4], specific state of the program [2, 6, 12, 14], or require manual instrumentation to specify observations [8, 13, 20]. REPT [3] produces an approximation of the `OmniTable` for a short window of instructions and thus limits a developer's ability to diagnose bugs that introduce latent errors (e.g., data corruption, wild stores) which are difficult to detect until much later in the execution. In contrast, the `OmniTable` allows observation of all program state at every point in time during an execution.

Execution mining [10] models software at a similar granularity to the `OmniTable` and provides similar visibility, repeatability and expressibility. However, due to expensive materialization costs, the resulting system, Tralfamadore, has prohibitive space and computation overheads, making it more suitable for understanding programs rather than debugging them. Further, the relational model used by SteamDrill allows for more query optimizations.

Many systems have noted that deterministic replay can be a great help when debugging software problems [5, 9, 16, 21, 23]. More recently, JetStream [17] and Sledgehammer [18] use deterministic replay as a vehicle for parallelizing debugging queries. These tools provide visibility and repeatability, but limit observation due to poor specification languages; the `OmniTable` abstraction instead supports more expressive SQL queries.

PARTICLE [6], Fay [4], Pivot Tracing [13] and PMSS [12] reduce the overhead of inspection by using query planning. SteamDrill applies these techniques to reduce query latency, but innovates by structuring query processing as an iterative process to take advantage of filtering made possible by using rounds of deterministic replay to resolve a query.

6 Conclusion

In conclusion, we propose the `OmniTable`, the first abstraction of a program execution to provide visibility, repeatability and expressability. Our prototye, SteamDrill, allows developers to observe an execution by writing declarative queries over the `OmniTable`. Since storing the `OmniTable` is infeasible, SteamDrill uses deterministic record and replay to store the execution associated with the `OmniTable`. To reduce the cost of querying the `OmniTable`, SteamDrill minimizes the amount of the table that is materialized by using a novel iterative, multi-round replay-based approach.

7 Acknowledgments

We thank the anonymous reviewers for their thoughtful comments. This work has been supported by the National Science foundation under grants CNS-1513718 and DGE-1256260, and by a Microsoft Ph.D. Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] https://jira.mongodb.org/browse/SERVER-21568.
[2] Peter C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems*, 13(1):1–31, February 1995.
[3] 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.
[4] 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.

[5] 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.

[6] 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.

[7] 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 sub-totals. *Data mining and knowledge discovery*, 1(1):29–53, 1997.

[8] 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.

[9] 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.

[10] 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.

[11] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 25–33, New York, NY, USA, 2006. ACM.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM, 1979.

[20] 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.

[21] Sudarshan Srinivasan, Christopher Andrews, Srikanth Kandula, and Yuanyuan Zhou. Flashback: A light-weight 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.

[22] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, 2002.

[23] 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.

[24] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *Proceedings of the 36th International Symposium on Computer Architecture*, pages 325–336, June 2009.