Data-Centric Execution Inspection

by

Andrew Quinn

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2021

Doctoral Committee:

Researcher Jason Flinn, Co-Chair
Assistant Professor Baris Kasikci, Co-Chair
Adjunct Associate Professor Michael Cafarella
Assistant Professor Jean-Baptiste Jeannin

Andrew Quinn

arquinn@umich.edu

ORCID iD: 0000-0002-0785-4119

ACKNOWLEDGEMENTS

I would like to thank my committee members, Jason, Mike, Baris, and Jean-Baptiste, for their help and dedication towards me throughout my time as a Ph.D. student. In different ways, each of you has served as an advisor and mentor and improved my research.

Jason was my first advisor and taught me most of what I know about computer systems research. I constantly strive to produce research at his caliber; if I get halfway there I will have had an amazing career. His ability to distill complex research arguments into simple succinct thoughts is inspiring (e.g., "current debugging tools are either interactive or useful"). Most importantly, Jason did his best to instill good research taste in me and taught me to never be afraid of projects that require putting your "nose-to-the-grindstone".

Mike taught me to think big-picture. During research meetings, Mike throws out ideas not for your current project, or even the next one, but for your student's student's project (this actually happened once!). He is also a fantastic collaborator and someone who was always willing and eager to discuss life outside of work.

Baris has shown me amazing kindness throughout my time as a Ph.D. student. Long before I was his advisee, he encouraged me to mentor a number of his students on research projects and allowed me to see what it is like to be on the other side. When things were crazy last January, Baris's first message to me was "what can I do?". Baris has played collaborator, mentor, therapist, advisor, and dissertation chair for me; I do not think it would be possible to do enough to repay him.

Thank you to the amazing University of Michigan Computer Science and Engineering professors, especially Manos Kapritsos and Mosharaf Chowdhury, were always there when I had a technical question, practice talk, or wanted to talk about life.

Thank you to my senior-graduate-student mentors when I was a junior student, David Devecsery and Mike Chow, who taught me how to build research systems.

Thank you to the Efeslab:Jiacheng Ma, Gefei Zou, Kevin Loughlin, Marina Minkin, Andrew Loveless, Tanvir Khan, and Sir Ian Neal, who graciously accepted me as one of their own over the last year; my 4929 lab-mates, Vaspol Ruamviboonsuk, HyunJong Lee, Ayush Goel, Muhammed Uluyol, and Chris Baek who were always up for a philosophical debate; and my friends from my cohort Ofir Weisse, Akshitha Sriraman, and Tim Tripple,

who were always there to celebrate triumphs, console defeats, and grab a cup of coffee.

Thank you to my parents, David and Bonnie, siblings, Eric and A.J., mother-in-law, JoAnn, and sister-in-law, Elizabeth, who provided me support and were always willing to hear me ramble about systems research, even when they were not sure what any of it meant.

Finally, I want to thank my family. My dogs, Huckleberry and Harley, have heard more about the `OmniTable` than any other living thing. My wife Tully is my rock. Six years ago when I started this process, I had no idea it would be this hard. You have consistently helped me through the whirlwind of emotions that comes with a Ph.D. You made this all possible. Thank you.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Modern software projects are incredible feats of engineering that manage dozens of concurrent execution tasks, are comprised of millions of lines of code, and written by hundreds of developers. Moreover, projects must meet an ever growing set of complex requirements, including correctness, performance, and security. Due in part to this complexity, developers make mistakes that lead to software failures that have devastating cost to society; in 2020 alone, operational failures caused by software bugs cost US companies 56 trillion dollars.

To understand the causes and effects of a software bug, a developer usually dynamically inspects the state of their execution by using an inspection tool such as gdb, Intel Pin, or logging. Debugging tools today support an inline inspection interface, which essentially requires iteratively constructing new inspection programs to understand a bug. The inline inspection interface imposes two limitations: (1) powerful inspection programs impose infeasible performance overhead and (2) inspection programs are needlessly complex to specify.

This dissertation proposes an alternative model for inspecting an execution called data-centric execution inspection. The framework takes a data-oriented view that considers an execution as a first-class data object and enables execution inspection as queries over the data objects. This dissertation shows that a data-centric framework enables the use of common data-centric approaches, namely cluster-scale parallelization and relational query models, to enable fundamentally more powerful inspection through three projects.

First, this dissertation shows how data-centric execution inspection enables cluster-scale parallelization of execution inspection to alleviate the performance limitations of existing tools. Data-centric execution inspection enables systems to inspect multiple regions of an execution simultaneously, so a system can parallelize inspection work across thousands of cores in a compute cluster. Alas, existing techniques and systems do not parallelize well, since they assume that inspection occurs inline with the original program. This thesis redesigns inspection techniques and tools by following scalability as a first class design constraint to facilitate cluster-scale parallelization of execution inspection. First, JetStream parallelizes the work of dynamic information flow tracking (DIFT) an order of magnitude better than prior approaches by partitioning DIFT across two phases and leveraging

a different from of parallelism for each phase. Second, Sledgehammer proposes a general vision for cluster-fueled debugging, which uses thousand-core computer cluster to enable debugging that is both powerful and interactive. Sledgehammer identifies cluster-fueled debugging as both a vehicle for accelerating existing debugging tools, such as retro-logging to enable after-the-fact execution inspection, and enables fundamentally more powerful tools, such as continuous-function evaluation to enable "always on" evaluation of complex global program invariants.

Second, this dissertation shows how a relational query model, called the `OmniTable` query model, transforms execution inspection from a low-level programming task into a high-level data science task to enable inspection that is both low-latency and simpler to specify. The model exposes a table abstraction of each execution and supports SQL queries for inspection. Results indicate that debugging using the `OmniTable` model is more succinct than in current state-of-the art tools. Moreover, our prototype, Steamdrill, optimizes inspection queries using a planning approach that seamlessly uses Sledgehammer-style cluster parallelization, standard relational optimizations, and a novel multi-replay approach to provide results an order-of-magnitude faster than prior general-purpose inspection tools.

CHAPTER I

Introduction

As software has increased in ubiquity, the costs of software bugs have become massive. The consortium for information and software quality estimate that software failures caused by software bugs cost companies in the United States 1.56 trillion dollars in the year 2020 [44]. In addition to financial costs, software bugs have interrupted key societal infrastructure, leading to massive power outages [62] and crippled health care organizations [88]. Finally, software bugs have even lead to loss-of-life due to bugs in the transportation industry [76] and medical devices.

To reduce the effects of software bugs, the community has devised a number of techniques, tools, and systems that improve software reliability. Of particular importance are techniques that help developers diagnose the causes and effects of a software failure. Developers use these techniques across many domains, including debugging, in which a developer identifies the root cause of a failure; security forensics, in which a developer identifies which secret data was leaked; and configuration management, in which a developer identifies which configuration parameters lead to software misbehaviors. Studies show that developers spend the majority of their time on these development tasks [57, 82].

To understand the cause and effects of a software failure, a developer traditionally inspects the runtime behavior of a program preceding and proceeding a failure; we refer to this process as execution inspection. Execution inspection is an iterative process: in each iteration, a developer gathers information about the execution state of a failing program until the developer can isolate the key causes and/or effects of their software's actions [25, 30, 32, 9]. Developers use tools, such as `gdb` or logging, to inspect program state: studies find wide-spread use of both ad-hoc tools (e.g., nearly all open-source software uses logging [91]) and sophisticated techniques (e.g., memory tools have altered the makeup of the bugs found in modern software [48]).

The power of the debugging tools at a developer's disposal has an outsize impact on their ability to debug their software. In this dissertation, we identify two fundamental

limitations inherent in current inspection tools. First, existing tools impose high latency, which is compounded by the iterative nature of execution inspection. Second, existing tools require that a developer construct complex specification in order to perform execution inspection. Then, we propose a novel data-oriented framework for program inspection that resolves the specification and performance problems in existing tools. we show that **A data-centric model enables fundamentally more powerful execution inspection by facilitating relational query models and cluster-scale parallelization.**

1.1 The Limitations of Existing Tools

Execution inspection is a programming task in which a developer specifies the program states that they wish to inspect by writing a new *inspection program*. For example, to track all values assigned to a variable, a developer might add a `set` to their program to track all assigned values, add instrumentation that updates the `set` after every assignment of the variable, and add code that prints all `set` elements at the end of the program execution. Different inspection tools offer different programming languages for this task. For example, if the developer used `gdb` to write the inspection program above, the `set` becomes paper-and-pencil notes, while the instrumentation becomes breakpoints with print statements. However, nearly all current tools support an *inline inspection model*, in which developers specify inspection logic as an imperative program that executes inline with the original buggy program. The inline inspection model directly leads to both poor performance and specification complexity.

Performance Inline inspection requires that a developer choose between power and interactivity when performing execution inspection. For simple problems, a developer may track program state infrequently, which imposes low performance overhead and allows the developer to interactively inspect their program. However, challenging issues usually require more heavyweight execution inspection. A developer will perform detailed logging, create functions that validate complex data-structures, and track the causality of execution state using tools like dynamic information flow tracking (DIFT). Heavyweight execution inspection is expensive: detailed logging, invariant, and DIFT can impose many order of magnitude slowdowns [23, 74]. High costs render heavyweight tools non-interactive, which is especially problematic given the iterative nature of execution inspection. As a result of this high cost, developers will even forgo the use of heavyweight tools altogether [25].

Specification Current inline inspection programming models make it difficult to correctly specify program inspection logic. Even basic questions, such as the variable value scenario presented above, require surprisingly complexity. The simple question requires the developer construct new data-structures and manually identify all program locations to injection their instrumentation. Depending upon the tool that the developer uses, they may need to contend with other software pitfalls including dynamic memory, fault handling, and concurrency. Inspection specification is made even more complex by the need for performance; achieving low-latency inspection requires implementing low-level optimizations, such as optimistic hybrid analysis [22] or path profiling [5], that are complex. In general, since the inline debugging model closely resembles traditional imperative programming, the developer will encounter many of the same bugs that they traditionally encounter when building software.

1.2 Data-Centric Execution Inspection

In this dissertation, we introduce a new data-centric model for execution inspection that alleviates the performance and specification limitations inherent in execution inspection tools. The key idea is to treat a program execution as a first-class data object, which developers query to perform execution inspection. Treating an execution as a data object enables the use of data-centric techniques to accelerate and simplify debugging. Data-centric execution inspection enables cluster-scale parallelization as a mechanism for better performance, since the execution data object model allows a developer to inspect multiple regions of an execution simultaneously. Additionally, data-centric execution inspection facilitates relational query models to simplify the specification of execution inspection.

Below, we elaborate on the three projects presented in this dissertation that provide evidence of the benefits of data-centric execution inspection.

1.2.1 Cluster-Scale Parallelization of Execution Inspection

In the first part of this dissertation, we show cluster-scale parallelization can alleviate the performance limitations of existing tools. Data-centric execution inspection allows multiple regions of an execution to be inspected simultaneously, so the work of execution inspection can be parallelized across thousands of cores in a compute cluster. Unfortunately, existing execution inspection tools and techniques assume that inspection occurs inline with execution, so prior approaches to scale inspection tools have me little success. Prior work even called one inspection technique, dynamic information flow, “embarrassingly sequential” [72].

Parallelizing execution inspection requires re-designing inspection techniques and tools to follow scalability as a first-class design constraint. In this dissertation, we first show how to accelerate DIFT using cluster scale parallelization (see chapter III). Then, we generalize the lessons learned in parallelizing DIFT to generally parallelize debugging-style questions (see chapter IV).

JetStream—Cluster-scale Parallelization of Information Flow Queries Dynamic information flow tracking (DIFT) is an important tool for understanding and troubleshooting program behavior. Originally proposed by the security community [64], DIFT has subsequently proven useful in a diverse set of domains including forensic analysis [42], information provenance [23], privacy [26], application debugging [72], and configuration troubleshooting [3, 4]. DIFT instruments an application binary to track data and/or control flow from global sources (e.g., program inputs) to global sinks (e.g., program outputs). Prior parallel DIFT techniques have failed to scale beyond a few cores in a single machine because of the fine-grain sequential dependencies between memory and register values: the set of dependencies at each step in the execution is a function of a large number of prior instructions, so partitioning DIFT into tasks that can be processed concurrently is difficult.

JetStream, my prototype, successfully parallelizes DIFT queries across many cores in a compute cluster by splitting the queries into two phases and using a different form of parallelism for each phase [69]. The first phase is the **local DIFT phase**, which uses epoch parallelism [84, 65]. It time slices an execution into sequential chunks, called epochs, and calculates the information flow dependencies within each epoch on a separate core in parallel. Epoch parallelism effectively breaks the sequential dependencies in DIFT and facilitates parallel processing. Next, the **aggregation phase** combines the information flow from the local DIFT phase to compute the final result. The aggregation phase organizes the cores into a chain in order of program execution and structures computation as a stream processing algorithm along the chain: Information about sources is passed *forward* along the chain. Information about sinks is passed *backward* along chain.

The structuring of the aggregation phase is the most important factor in enabling JetStream to scale much better than prior approaches at parallelizing DIFT. The insight is that a small amount of sequential information allows JetStream to avoid huge amounts of unnecessary work. Streaming this data along a sequential chain allows most processing to occur in parallel, with the sequential limitation being essentially the time to pass a single data value from one end of the chain to the other; this is much less than the total query time even for hundreds of processors. We used JetStream for DIFT queries over seven desktop and server applications and find that JetStream scales DIFT to hundreds of cores for these

applications. DIFT queries calculated using 128 cores are 12–48 times faster than sequential queries, and, in most cases, execute faster than the original execution of the program.

SledgeHammer—Cluster-fueled Debugging Cluster-fueled debugging generalizes the parallelization approach of JetStream to accelerate general-purpose debugging queries and enable execution inspection that is simultaneously powerful and interactive. Cluster-fueled debugging parallelizes heavyweight debugging tools across many cores in a compute cluster, which requires introducing scalability as a first-class design constraint. Sledgehammer, the first general-purpose cluster-fueled debugger, supports an interface designed to mimic logging, invariant function, and analysis approaches commonly used when debugging today. The system uses epoch parallelism to parallelize the calculation of *tracers*, functions that inspect the program state of an execution to produce output and enable logging and invariant functions. Critically, Sledgehammer preserves the behavior of the original execution by using a low-overhead, scalable, undo-log approach to isolate changes made by a *tracer*. Sledgehammer also supports two different interfaces for programmatically analyzing and aggregating tracer output in order to accelerate a wide variety of different debugging questions.

Sledgehammer uses the tracer and analyzer interface to support three parallel debugging tools. *Parallel Retro-logging* allows developers to modify the logging logic in their program and produces new logging output for a previously-recorded execution. *Continuous Function Evaluation* allows a developer to define a function over the state of their execution that will logically be executed after every program instruction. Sledgehammer returns a log that indicates all instances in the execution where the function's output changes. Finally, *Parallel Retro-Timing* allows a developer to retroactively measuring timing in a previously-recorded execution. Developers can use these tools in conjunction (i.e., a developer can use a continuous function alongside new logging); regardless of which tool a developer uses, they can aggregate output using a Sledgehammer analyzer interface.

We evaluate Sledgehammer in seven scenarios debugging common problems in complex applications (Memcached, MongoDB, and nginx). With 1024 compute nodes, Sledgehammer returns the same results as sequential debugging, but on average returns results 416 times faster due to parallelization.

1.2.2 Steamdrill & The OmniTable Query Model

Cluster-scale parallelization largely solves the performance tradeoff in the inline inspection programming model. Alas, the use of parallelism actually further exacerbates the specification issues with execution inspection, since developers must explicitly consider

parallelism when constructing inspection programs. In the final chapter of this thesis, we identify how relational query models can alleviate these concerns to enable execution inspection that is both low-latency and has requires simpler specification.

They key idea is an new relational query model for expressing execution inspection tasks that converts inspection from a low-level programming task into a data science task. At the center of the model is an `OmniTable`, a table representation of an execution that encompasses the complete view of all history program state reached during an execution. In essence, the `OmniTable` realizes the vision of treating an execution as a first-class data object. The `OmniTable` query model supports inspection of an execution through SQL queries over the `OmniTable` associated with the execution and leverages database concepts to create database-style views that contain more familiar debugging abstractions (e.g., the `Funcs` view identifies the functions executed in an execution). This is not the first work to propose new models for program inspection, but prior tools expose incomplete program history, only exposing data at specific points in time (e.g., function entry/return [27, 31]), only exposing some program state (e.g., global variables [55]), or only exposing data from necessarily incomplete software logs [54, 7, 77].

Unfortunately, an `OmniTable` for even a short execution is petabytes in size, which makes it infeasible to store or compute a fully materialized table. So, instead of materializing an `OmniTable` for each execution, our prototype, Steamdrill, lazily materializes the table for each developer query. The system uses deterministic record/replay to store the execution associated with each `OmniTable`. When a developer queries an `OmniTable`, Steamdrill generates instrumentation and re-executes the replay with the instruction to produce query output. Steamdrill uses a query optimization and planning approach to construct the instrumentation, and thus seamlessly leverages standard database optimizations (e.g., predicate push-downs) and cluster-scale parallelization (e.g., Sledgehammer and JetStream). Additionally, the system proposes a novel resolution strategy that partitions instrumentation across multiple replay executions. Intuitively, a multi-replay approach allows Steamdrill to use data that is inexpensive to observe (e.g., data about invoked functions) to reduce the amount of observation required of data that is expensive to observe (e.g., data about executed instructions).

We results show major advantages compared to prior tools. As a baseline, we use `gdb`'s python bindings, which offer a scripting interface over standard `gdb` primitives such as breakpoints and backtraces. Across 9 queries written to diagnose bugs in 4 case studies, we find that `OmniTable` queries require 3.465 times fewer lines, 1.95 times fewer operands and operators, and would require 3.775 times less time to construct (according to Halstead Complexity [34]) than their `gdb` counterparts. Moreover, we find that 3 representative

`OmniTable` queries, which require no modification for cluster-scale parallelism, are an average of 93.91 times faster than their `gdb` counterparts when executed on 64 cores.

1.3 Road-map

In the rest of this thesis, we elaborate on three projects that argue for a data-centric model for program inspection. First, we outline how my work relates to existing literature (chapter II). Then, we describe how a data-centric model enables cluster-scale parallelization, which alleviates the performance limitations of existing program inspection tools in the context of dynamic information flow (chapter III) and debugging queries (chapter IV). Next, we outline how a data-centric model encourages relational query models, which simplify the specification of inspection logic (chapter V). Finally, we describe conclude and identify the future of data-centric execution inspection (chapter VI).

CHAPTER II

Related Works

Deterministic record and replay is roughly as old as computing itself and has been used by many systems for program inspection. Deterministic record and replay reliably reproduces a program execution, which makes it possible to repeatedly inspect program behavior that would otherwise be difficult to reproduce due to non-determinism (e.g., concurrency errors and other "heisenbugs"). In addition, systems have used deterministic record and replay to enable reverse execution, which allows these tools to logically execute program statements in reverse order [6]. Prior work uses reverse execution to provide useful debugging features (e.g., reverse breakpoints and watchpoints), but fundamentally exposes an imperative inline programming model for specifying program inspection logic. In contrast, the execution data object framework explores how a data-centric approach to program inspection enables easier specification for program inspection.

Prior systems use deterministic record and replay to parallelize program inspection through epoch parallelism [65, 72, 84]. These systems are intended for online analysis (e.g., for security use-cases), where it only makes sense to scale to multi-core machines. In contrast, the execution data object framework models program inspection as a query over a prior program execution, which encourages using a massive shared-cluster of thousands of cores to accelerate inspection tasks. New design constraints and algorithms are necessary to scale to thousands of compute cores.

Execution mining [47] treats executions as data streams that can be dynamically analyzed. While execution mining offers a similar vision to that of an execution data object, it retains an imperative inline programming model. Thus, this work suffers from both the specification and performance problems of existing tools.

In the rest of this section, I identify the related work for each of the projects in this dissertation.

2.1 Jetstream

JetStream is the first system to parallelize DIFT across a cluster, and it is the first system to efficiently track millions of global sources, global sinks, and dependencies. Several prior systems have parallelized DIFT across the cores of a single machine. To achieve cluster-level scalability, JetStream's main contribution is parallelizing the aggregation of local DIFT data while minimizing the communication between cores.

Like JetStream, Speck [65] partitions an execution into epochs and performs local DIFT for each epoch. Speck tracks only a single label (tainted or untainted). Speck's local DIFT produces a log of sub-commands, which it then optimizes to achieve an up to 6x reduction in log size. Aggregation is done sequentially over the optimized log. This limits the speedup achieved by Speck to only 2x on a 8-core machine.

Ruwase et al. [72] partition an execution into epochs and perform local DIFT on each core using custom hardware [16]. JetStream's merge log optimization is derived from this work; thus, the local DIFT phases of the two systems are similar. However, Ruwase et al. perform aggregation sequentially, and this limits scalability. Like Speck, their system tracks only a single label. TaintPipe [61] partitions DIFT into epochs and tracks taint as symbolic formulas inside each epoch. TaintPipe also performs aggregation sequentially. It is unclear how symbolic tracking can scale efficiently to millions of labels and dependencies.

JetStream focuses on after-the-fact analysis, while prior DIFT parallelization has focused on live analysis during execution. Live analysis runs only a single pre-defined query, but it is suitable for security use cases in which sensitive actions such as sending network output need to be blocked based on the DIFT results (Speck and Ruwase et al. delay output to support this functionality, while TaintPipe does not). In contrast, after-the-fact analysis is suitable for tasks like forensics [42], debugging [72], configuration troubleshooting [3, 4], analysis of privacy leaks [26], and provenance [23]. No prior system has parallelized after-the-fact DIFT.

Many systems have explored how to make DIFT itself faster. One promising idea is decoupled execution, in which the DIFT work is split into an instrumentation thread and an analysis thread. ShadowReplica [36] combines decoupled execution with static analysis to reduce the amount of instrumentation that Pin must perform. TaintPipe combines decoupled execution with another form of static analysis: taint abstractions for commonly used function. libdft [40] provides several low-level optimizations for accelerating Pin-based DIFT. Profiling and/or static analysis can also reduce the cost of dynamic instrumentation [14, 37, 68].

These ideas are orthogonal to the speedups that JetStream provides through paralleliza-

tion. In fact, our evaluation shows that Pin dynamic instrumentation is often the scalability bottleneck after JetStream parallelization, so incorporating these optimizations into JetStream is a very promising direction for future work.

2.2 Sledgehammer

Sledgehammer is the first general-purpose framework for accelerating debugging tools by parallelizing them across a cluster. It has frequently been observed that deterministic replay [24] is a great help in debugging [15, 43, 66, 79, 85]. Sledgehammer leverages Arnold [23] replay both to ensure that results of successive queries are consistent and also to parallelize work via epoch parallelism [84]. JetStream [69] uses epoch parallelism for a different task: dynamic information flow tracking (DIFT). Sledgehammer' tracer isolation has less overhead and scales much better than the dynamic binary instrumentation used by JetStream, making it better suited for tasks like debugging that need not monitor every instruction executed.

Many tools aim to simplify and optimize the dynamic tracing of program execution. Dtrace and SystemTap reduce overhead when tracing is not being used, but are expensive when gathering large traces [13, 67]. Execution mining [47] treats executions as data streams that can be dynamically analyzed and supports iterative queries by indexing and caching streams. Other tools introspect distributed systems. Fay [27] lets users introspect at the start and end of functions but injected code must be side-effect free. Pivot tracing [54] lets users specify queries in an SQL-like language. These tools help debug parallel programs, but, unlike Sledgehammer, they are not themselves parallelized for performance.

Several prior systems support retro-logging. Most isolate all code added to an execution [17, 38]; this comes with high overhead. Sledgehammer reduces isolation overhead through compiler-based isolation and hides remaining overhead through parallelization. Like Sledgehammer, rdb [35] allows users to modify source code and executes the modifications during replay; however, rdb prohibits program state modifications instead of isolating them. Dora [85] allows the added code to perturb application state and uses mutable replay to make a best effort to keep replaying the application correctly after the perturbation. This eliminates isolation overhead, but there is no guarantee that the debugging output will be correct. Mutable replay is a good choice when output is simple and can be verified by inspection, but incorrect results could prove frustrating for complex debugging tasks.

As documented in the wild store scenario, developers commonly write debug functions to verify invariants. Researchers have advocated running similar functions at strategic code locations to repair structures [21] or detect likely invariants [28]. Continuous function

evaluation takes this to an extreme by logically running a function after every instruction. X-Ray [3] systematically measures timing during recording to support profiling of replayed executions; Sledgehammer's more general interface allows debuggers to define the events being measured and understand the uncertainty in timing results.

2.3 Steamdrill & The OmniTable Query Model

Prior systems have proposed high-level languages for debugging in which developers specify inspection logic in a declarative fashion. Many of these systems retain the challenges of inline debugging because they require the developer to manually instrument their software to specify interesting program state [77, 7, 54, 33]. Others limit the observations that a developer can make to only specific program state (e.g., the heap [31]) or only specific program events (e.g., function entry/exit [27, 55]). The Program Monitoring and measuring system [49] provides a relational interface for inspecting source-level events; a developer must manually instrument software to inspect events that occur below the source level (e.g., garbage collection). No existing system supports a non-inline programming model for inspecting arbitrary architectural program state.

In addition, prior high-level inspection languages allow a developer to instrument a future execution of their software. In these systems, execution state is ephemeral. So, while many of these systems propose relational optimizations to optimize queries [31, 27, 54, 49], these systems are unable to take advantage of the full range of optimizations afforded to Steamdrill.

Execution mining [47] models software at a similar granularity to the `OmniTable` and uses deterministic replay to facilitate repeated queries of the same execution. However, Tralfamadore, the system that supports the execution mining model, supports a high-level stream-based language which logically inspects a program inline with the original execution. Moreover, due to expensive materialization costs, Tralfamadore, has prohibitive space and computation overheads, making it more suitable for understanding programs rather than debugging them.

Many systems have noted that deterministic replay can be a great help when debugging software problems [43, 79, 83, 66, 29]. More recently, JetStream [69] and Sledgehammer [70] 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.

CHAPTER III

JetStream: Cluster-scale Parallelization of Information Flow Queries

Dynamic information flow tracking (DIFT) has emerged as an important tool for understanding and troubleshooting program behavior. Originally proposed by the security community [64], DIFT instruments an application binary to track data and/or control flow from global sources (e.g., program inputs) to global sinks (e.g., program outputs). Information flow analysis has proven to be helpful in a diverse set of domains that include forensic analysis [42], information provenance [23], privacy [26], application debugging [72], and troubleshooting of configurations [3, 4].

Unfortunately, dynamic information flow analysis can be painfully slow; depending on the granularity and amount of information tracked, execution slowdowns of up to one or two orders of magnitude are common. While this cost can be reduced by limiting analysis to managed languages such as Java or by restricting the types of queries that can be performed, *general-purpose* information flow analysis over binary code requires batch-style analysis for substantial programs. In other words, the user employing DIFT must run such analyses over the course of hours. DIFT would be much more powerful if analysis could be employed interactively; for instance, a user could refine a particular query by changing sources, sinks, the propagation function, the granularity of instrumentation, or the period of program execution over which the analysis is employed. The user could then narrow down the bug, misconfiguration, or privacy violation in a manner similar to traditional debugging techniques.

Our goal is to make DIFT queries interactive by parallelizing them across many cores in a compute cluster. With hundreds or thousands of cores, DIFT queries that previously took hours or days can complete in seconds or minutes, enabling refinement and iteration over multiple queries. Thus, a shared cluster can become a valuable resource for a large team of system operators or programmers who want to occasionally engage in an interactive

debugging or troubleshooting session using DIFT tools. Usage scenarios for DIFT include both live analysis (as the program runs) and after-the-fact analysis (executed on a replay of an execution). We target the latter scenario.

Previous efforts at parallelizing DIFT have met with only limited success. Information flow is inherently difficult to scale (Ruwase et al. call it "embarrassingly sequential" [72]) because it tracks many fine-grained sequential dependencies between memory and register values. The set of dependencies at each step is a function of a large number of prior instructions executed by the program. Consequently, prior efforts have produced parallel versions that scale to only a few cores on a single machine, and no current approach can effectively leverage a commodity cluster to scale DIFT.

Our solution, JetStream, provides cluster-level scalability by computing information flow in two phases, each using a different form of parallelization.

The first phase is the **local DIFT phase**; it divides program execution into time segments (epochs) and assigns a separate core to compute the information flow for each epoch. Each core determines the dependencies within its epoch that may be relevant to answering the overall query. It tracks sources and sinks that are identified explicitly in the query (e.g., network input and output); we call these *global sources/sinks*. It also tracks locations that may serve as links between global sources and sinks; we call these *local sources/sinks*. Local sources are all memory addresses and registers at the beginning of an epoch, and local sinks are all memory addresses and registers at the end of an epoch. The local DIFT phase parallelizes cleanly into separate partitions, with all dependencies between partitions resolved in the next phase.

Two challenges arise for the local DIFT phase. First, computing all dependencies that may be relevant to the query is too expensive. We address this challenge by deferring and avoiding work as much as possible. JetStream uses merge trees [72] to represent and manipulate dependency sets more efficiently. More importantly, it defers traversing these merge trees until the next (aggregation) phase; that phase avoids traversing the vast majority of tree nodes that do not lie on a dependency path between a global source and a global sink.

The second challenge is that each epoch must follow the same execution as the original execution, so that the aggregation of local DIFTs produces a result equivalent to a sequential DIFT. JetStream uses checkpointing and deterministic record/replay to divide an execution into epochs and perform the local DIFT for each epoch independently, yet consistently. JetStream uses lightweight statistics collected during the original execution of a program to partition the DIFT work equally, and it uses heavyweight statistics collected during the first query of an execution to better partition subsequent queries.

The second phase, called the **aggregation phase**, prunes and combines the information from the local DIFT phase to compute the final result, i.e., the relationship between global sources and global sinks across the entire execution. The cores in this phase are organized in a chain in order of program execution, and the computation is structured as a stream processing algorithm with pipeline-style parallelism. Each core resolves dependencies using information from one epoch's local DIFT phase, and the global query is answered via two streaming passes.

In the first streaming pass, the locations (registers and memory addresses) that are derived from global sources are passed *forward* along the chain (from the beginning to the end of execution). This information is used to prune deferred operations that do not depend on a global source. In the second streaming pass, the locations that propagate dependencies to a sink are passed *backward* along the chain. This lets JetStream prune deferred operations on which no sink depends.

The structuring of the aggregation is the most important factor in enabling JetStream to scale much better than prior approaches at parallelizing DIFT. Our insight is that a small amount of sequential information is necessary to avoid huge amounts of unnecessary work; this information is essentially the locations that depend on global sources (forward pass) and the locations on which global sinks will depend (backward pass). Streaming this data along a sequential chain allows most processing to occur in parallel, with the sequential limitation being essentially the time to pass a single data value from one end of the chain to the other; this is much less than the total query time even for hundreds of processors.

The contributions of this paper are:

- An algorithm for parallelizing DIFT that scales much better than prior approaches, enabling interactive (sub-minute) response times.
- Scalable and efficient support for tracking millions of distinct global sources and sinks at byte granularity, without restrictions on source-code availability, compute platform, or query type.
- A detailed evaluation of the remaining bottlenecks in accelerating DIFT through parallelization.

We have applied JetStream to run DIFT queries over seven desktop and server applications: Evince, Firefox, Ghostscript, Gzip, Mongodb, Nginx, and OpenOffice. Our results show that JetStream scales DIFT to at least 128 cores for these applications. It accelerates DIFT queries to run 12–48 times faster than sequential queries, and, in most cases, runs queries faster than the original execution of the program.

3.1 Motivation

DIFT is a fundamental analysis that is useful in diverse domains. For example, Arnold [23] uses DIFT for provenance queries that reveal how data values in files and application memory were derived. In forensics [42], DIFT has been used to answer questions such as: "How was my system compromised?" and "What data was leaked?" TaintDroid [26] and similar systems use DIFT to reveal whether an application execution leaks sensitive data. X-Ray [3] uses DIFT to identify misconfigurations that cause performance anomalies, and ConfAid [4] uses DIFT to identify misconfigurations that cause bugs. Poirot's [41] use of DIFT helps determine if a security vulnerability has been exploited.

Many of the above systems run complex DIFT queries on native binaries and can suffer from painfully slow DIFT query times. These systems are often forced to use batch-style computation, even though many would ideally be interactive in nature.

Consider a developer debugging an incorrect output value from a Web server. Using JetStream, they begin by running a DIFT query that shows all program inputs from which the faulty value was derived. This alone is not enough to reveal the bug, so they run an additional query tracking the inputs that led to a correct output. Comparing the results shows that inputs from a particular network connection led to the faulty output but not the correct one. Using this information, they discover a bug in the code which parses network inputs. To see if this bug has impacted any file system state, the developer runs another query specifying all values from the faulty parsing code as global sources and all file system outputs as global sinks. They detect that no permanent state has been affected by their bug. Next they consider other forms of external output such as network messages.

Debugging the problem and determining the impact of the bug both require multiple DIFT queries. Further, phrasing the correct queries may be non-trivial and require multiple iterations to get helpful results. If each query takes hours to complete, then this process only makes sense for the most difficult bugs. In contrast, low-latency DIFT enables information flow analysis to be an integral part of the debugging process.

3.2 Background

We first describe two technologies on which JetStream builds: dynamic information flow tracking and deterministic record and replay.

3.2.1 DIFT

Dynamic information flow tracking, sometimes referred to as taint tracking, instruments applications to monitor data flow as programs execute. In its most general form, DIFT reveals which *global sources* causally affect which *global sinks* according to a *propagation function*. Global sources are typically external program inputs, such as bytes read from a file or a network socket, and global sinks are typically external outputs.

The propagation function specifies what information flows to track during program execution. For example, a basic data flow propagation function for the instruction $x = y + z$ would state that the sources on which x depends are the union of the sources on which y and z depend. Usually, DIFT tracks data flow (as we do in this work), but some DIFT systems also track implicit flows propagated via control flow.

When an application executes, DIFT assigns a *taint identifier* to each unique global source. For each location, it maintains a set of taint identifiers that shows the global sources on which that location currently depends, and it updates taint sets as instructions execute. At each global sink, DIFT outputs the set of taint identifiers of all locations written to the sink (e.g., the bytes sent to a network socket). Thus, DIFT produces a set of $\langle globalsource, globalsink \rangle$ tuples that describe how particular global sources and global sinks are related.

JetStream tracks global sources, global sinks, and dependencies at byte granularity using binary instrumentation inserted by Pin [53]. A single JetStream query may look for relationships between millions of distinct global sources and sinks. In contrast, many prior DIFT systems require source code or the use of a managed language runtime. Others track only whether any global source data propagates to a global sink and cannot determine *which* sources affect each sink—such systems cannot answer questions such as: "Which inputs affected this program value?" or "What data did I leak?"

A JetStream query contains a program execution to monitor, a filter that specifies the global sources, a filter that specifies sinks, and a propagation function. For instance, a provenance query [23] might wish to determine the lineage of the data in a particular file. The source filter would match all external program inputs and the output filter would match writes to a particular file. This would reveal which bytes in the file were derived from which sources. Alternatively, a privacy query [26] might specify reads from sensitive files as sources and network outputs as sinks. This would reveal what data was leaked over the network and how it was leaked. JetStream provides an interface for supporting custom propagation functions and supplies Arnold's copy, data, and index propagation functions [23] as defaults.

For complex applications, mapping all global sources to all global sinks at byte gran-

	Instructions	Taint IDs	Merge Log	Live Set (forward pass)	Taint Tuples (backward pass)
Epoch 0	1. A = read() 2. B = read() 3. C = A + B	A: I_0 B: I_1 C: $M_{0,0}$	$M_{0,0} : \{I_0, I_1\}$	$\downarrow$ $\{A, B, C\}$	$\{< I_0, O_0 >,$ $< I_1, O_0 >\}$ $\uparrow$
Epoch 1	4. D = X + Y 5. E = C 6. B = 0 7. Z = A[D]	D: $M_{1,0}$ E: C_1 B: $\{\}$ Z: $M_{1,1}$	$M_{1,0} : \{X_1, Y_1\}$ $M_{1,1} : \{A_1, M_{1,0}\}$	$\downarrow$ $\{A, C, E, Z\}$	$\{< O_0, C >\}$ $\uparrow$
Epoch 2	8. F = E 9. write(F)	F: E_2 O_0: E_2		$\downarrow$	$\{< O_0, E >\}$ $\uparrow$

Table 3.1: JetStream Example. DIFT analysis of an example program. The n^{th} input to the execution is labeled I_n, the m^{th} output to the execution is labeled as O_m, the taint of variable X in epoch i is labeled X_i, and the j^{th} entry in the merge log for Epoch k is labeled $M_{k,j}$

ularity produces far too much information (e.g., terabytes of data for some benchmarks in Section 3.4). Thus, filters are needed to extract the right information succinctly. This leads to refinement through iteration. Our goal is to make DIFT fast enough to be interactive, so that a user can issue multiple queries to search for the right information.

3.2.2 Deterministic Replay

Deterministic replay allows the execution of a program to be recorded and reproduced faithfully. When a program first executes, all inputs from nondeterministic actions are logged; these values are supplied during subsequent replays in lieu of performing the nondeterministic operations again. Thus, the program starts in the same state, executes the same instructions on the same data values, and generates the same results.

JetStream derives several benefits from using deterministic replay. First, replay allows JetStream to partition a recorded execution into epochs and execute these epochs in parallel. Deterministic replay guarantees that the result of stitching together all epochs is equivalent to a sequential execution of the program. Second, replay allows an execution recorded on one machine to be replayed on a different machine. There are few external dependencies, since interactions with the operating system and other external entities are nondeterministic and replayed from the log. Thus, the only requirement for replay is that the replaying computer has the same hardware architecture as the recording computer and that it runs a kernel modified to support replay. Finally, replay allows iterative queries over the same

execution.

JetStream uses Arnold [23] to provide deterministic record and replay of multithreaded, multiprocess applications. Arnold's performance overhead is less than 10% for most workloads, and its storage overhead is reasonable even for continuous recording of a workstation.

3.3 Design and Implementation

To parallelize a DIFT query, JetStream divides an execution into epochs and assigns each epoch to a different core. JetStream then evaluates the query in two phases: a *local DIFT* phase and an *aggregation* phase.

In the local DIFT phase, each core concurrently computes the relationships between sources and sinks within its epoch. A core can directly observe global sources and global sinks that occur during its epoch. However, some locations at the start of an epoch may depend on global sources from preceding epochs, and the local DIFT cannot know the actual dependencies because the local DIFTs for those preceding epochs are being executed concurrently. Thus, for all epochs but the first one, the local DIFT phase conservatively tracks all locations at the start of the epoch as *local sources* and assigns a unique *local source identifier* to each location at the epoch start. Similarly, the local DIFT cannot determine which locations at the end of an epoch will ultimately propagate to global sinks in succeeding epochs, so the local DIFT treats all locations at the end of the epoch as *local sinks*. A local DIFT phase thus tracks and reports dependencies between all sources (both local and global) and all sinks (both local and global).

In the aggregation phase, the cores organize as a chain in program execution order and communicate local DIFT results forward and backward along the chain to produce the final set of $\langle globalsource, globalsink \rangle$ tuples. We next describe these two phases in more detail.

Table 3.1 shows an example query in which JetStream finds all dependencies from global sources to global sinks in a simple program. Program execution is divided into three epochs (shown by the horizontal partitioning). The local DIFT phase is the region to the left of the double vertical bar, and the aggregation phase is the region to the right of the bar. Instructions 1 and 2 read data from global sources, and instruction 9 writes to a global sink.

3.3.1 Local DIFT

JetStream implements the local DIFT phase as a Pin tool. Executing an application with this tool attached is quite slow (e.g., 14–75x slowdown for the benchmarks in Section 3.4). There are two reasons: DIFT may add several additional instructions to track taint for

each application instruction executed, and Pin dynamically adds the instrumentation to an application as it executes. The first is a fundamental cost of DIFT, while the second is a consequence of using a dynamic instrumentation tool.

The local DIFT phase for a given epoch first replays the application uninstrumented to advance its execution to the start of the epoch, a process we call *fast forwarding*. JetStream may start the replay from the beginning, or it may start from a checkpoint of application state taken during recording or during a previous query. Given the relative speed difference between instrumented and uninstrumented execution, starting from the beginning is reasonable for low numbers of epochs. As the number of epochs increases, fast forward time comes to dominate total query time, and checkpoints are quite beneficial.

Next, JetStream attaches the DIFT tool to the application and Pin starts instrumenting the application to track dependencies. JetStream assigns a unique *source identifier* to each location modified by a global source in the epoch and to each location at the start of the epoch.

JetStream runs all threads of a multithreaded application on a single core to realize an important performance benefit: the instrumentation code does not need to obtain locks to synchronize access to the DIFT data structures because only one thread runs at any given time [87]. JetStream still fully utilizes the processor because each core runs a different epoch in parallel.

For each location, JetStream stores an integer *taint identifier* that represents the set of global and local sources on which that location currently depends. A taint identifier may be: (1) a global source identifier, (2) a local source identifier, or (3) an identifier that maps to a set of global and local sources. In the Taint IDs column of Table 3.1, I_0 and I_1 are global source identifiers, and C_1 and E_2 are local source identifiers that represent the taint of address C at the start of epoch 1 and the taint of address E at the start of epoch 2.

For each x86 instruction, the local DIFT tool reads the taint identifiers of the instruction's inputs and updates the taint identifiers of the instruction's outputs. Taint identifiers for registers are stored in a per-thread array, and taint identifiers for memory addresses are stored in a two-level page table. The tool decomposes the work for each instruction into a sequence of four sub-commands: `set`, `clear`, `copy`, and `merge`. The first three sub-commands are straightforward—`set` assigns a taint identifier to a location, `clear` assigns the NULL identifier to a location, and `copy` sets the destination's taint identifier equal to the source's. Thus, each of these sub-commands are low overhead integer operations. Table 3.1's Taint IDs column shows how the local DIFT tool updates the taint data structures: a `set` for instruction 1, a `clear` for instruction 6, and a `copy` for instruction 5.

The `merge` sub-command is used for instructions that combine dependencies, e.g., in-

structions 3, 4, and 7. For these instructions, the set of sources on which the output depends is the union of the sets of sources on which the inputs depend. Our original implementation tracked such sets explicitly, but this worked poorly. For some complex applications, the DIFT did not finish after running for hours, or the size of the sets exceeded the 256GB memory of our server. Intuitively, the reason is that the set of tuples that relate all local sources to all local sinks can be as large as the size of the address space squared.

We therefore turned to an idea proposed by Ruwase et al. [72] in which sets of taint values are represented by a binary tree. Each `merge` operation generates a new taint identifier to represent the set union. JetStream writes an entry to a *merge log*, which contains the taint identifiers of the input to the `merge`. Thus, the merge log is a DAG sorted in temporal order, and each node (entry) in the merge log represents a binary tree of taint identifiers rooted at that node. Any merge node can be *resolved* to a set of source identifiers by performing a depth-first traversal of the tree rooted at that node.

In the example, instruction 4 creates a merge node $M_{1,0}$ (each epoch has a distinct merge log, with the particular log denoted by the subscript). The node states that address D depends on whatever X and Y depend on at the start of epoch 1. Instruction 7 creates a merge node that has $M_{1,0}$ as a child, so address X depends on whatever A, X, and Y depend on at the start of the epoch.

Using the merge log yields two benefits. First, it defers expensive set union operations until the aggregation phase; optimizations in that phase avoid the need to perform the vast majority of such unions. Second, the merge log uses much less memory than storing a set for each location. Memory usage is roughly proportional to the number of unique merge operations rather than the total size of all taint sets for every location. The cost of using a merge log is that JetStream must perform a tree traversal when it needs to resolve a root node to a set of source identifiers.

JetStream makes two enhancements to Ruwase et al's algorithm. First, it uses a hash table to cache recently-seen merge pairs and reuse merge nodes when duplicates are found. Second, whereas Ruwase et al. used the tree data structure only for abstract values (i.e., local source identifiers); JetStream also uses the tree structure for sets of global source identifiers, such as distinct bytes from different sources encountered during the local epoch (as for instruction 3 in the example).

At the end of an epoch, JetStream writes four datasets to a shared memory buffer: global source metadata, global sink metadata, the merge log, and the taint identifiers for all local sinks. The global source metadata describes each global source identifier (e.g., the system call that read the byte, the file the byte was read from, the offset within the file, etc.). Similarly, the global sink metadata describes each byte sent to a sink. Since application

execution typically modifies only a small percentage of locations during a given epoch, the local sink identifiers for most locations will be the local source identifier of those locations. To save space, the local DIFT only outputs those local sink identifies where this relationship does not hold. These optimizations allow the output of the local DIFT phase to fit in the memory of modern servers (though it is still large, e.g., a few GB per epoch).

3.3.2 Partitioning

The time to produce an answer to a query depends on the longest local DIFT time for any epoch. Thus, to achieve good speedups, JetStream must partition local DIFT so that each core does roughly the same amount of work. To accomplish this, JetStream estimates the amount of time it will take to run local DIFT for any given interval of execution and defines epoch boundaries so that the estimated local DIFT time for each epoch is the same.

We estimate the local DIFT time for an interval of execution as a linear combination of three factors:

- **Fast Forward Time:** JetStream replays the application without instrumentation to advance execution to the start of the epoch. We estimate that this component of work is proportional to the user-level CPU time used for this portion of execution by the recorded application.

- **Instructions executed:** To track information flow for an interval of execution, JetStream must execute the instructions in that interval, as well as the instrumentation code that propagates dependencies among locations. This component of work is proportional to the number of instructions executed, which we estimate from the user-level CPU time used to execute the interval in the recorded execution.

- **Unique instructions executed:** Pin instruments an instruction when it is executed for the first time. With Pin, instrumentation cost is a significant portion of the overall DIFT time, especially for short intervals in which each instruction may only be executed a few times. As JetStream parallelizes the DIFT work across more cores, each interval becomes shorter, and the relative cost of instrumenting instructions increases. This component of work is proportional to the number of *unique* instructions executed. During recording, we read processor performance counters via the `perf_events` API to estimate the number of unique instructions executed by sampling the instruction pointer (we sample every 32 L1 instruction cache read misses for user-level code). When executing the first query for an execution, we use dynamic instrumentation to measure the actual number of unique instructions executed

during an interval; this adds little overhead compared to DIFT instrumentation.

To avoid confounding testing and training in our evaluation, we choose the constants in the model for the first query of an execution by running a linear regression over data from the *other* benchmarks in our set. The coefficient of determination (R^2 value) for these regressions is 0.86–0.87. Due to the high overhead of instrumenting code with Pin, the cost of inserting instrumentation (proportional to unique instructions executed) usually dominates the cost of running the instrumented code (proportional to instructions executed), especially for small epochs.

For subsequent queries of a given execution, we run a linear regression over the performance data gathered during the first query. This produces a much better R^2 value of 0.985. We also add the number of merges that occurred during each interval to our model, and that change slightly increases the R^2 value to 0.989.

JetStream partitions the recorded execution into n epochs of roughly equal local DIFT time as estimated by the above model, where n is the number of available cores to run the query. This process is conceptually simple, but a complication is that the total local DIFT time depends on the particular partitioning chosen because an instruction that is executed in multiple epochs will incur an instrumentation cost in each of those epochs. We solve this problem by using a hill-climbing algorithm in which each iteration updates the estimate of the total local DIFT time for the query, and the new estimate is used to calculate a better partitioning in the next iteration. Usually, this process converges after a small number of iterations.

3.3.3 Aggregation

The aggregation phase produces the set of $\langle globalsource, globalsink \rangle$ tuples that are related by the propagation function. Within a single epoch, a global source and global sink are related if the global sink either has the global source's identifier or if it has the identifier of a merge node and that node resolves to a set that contains the global source's identifier. If the global source and sink are in adjacent epochs, then they are related if there exists a location L at the boundary of the two epochs such that, in the first epoch, the local sink identifier of L depends on the global source, and in the second epoch, the global sink depends on the local source identifier of L. If one or more epochs separate the epochs of the global source and sink, then there must be multiple such relationships forming a continuous path from source to sink.

In Table 3.1, such a path exists between the global sources of instruction 1 and 2 and the global sink of instruction 9. In epoch 0, resolving the merge tree for address C reveals

that it depends on both global source 0 and global source 1. In epoch 1, the final value of E depends on the value of C at the beginning of the epoch. In epoch 2, instruction 9 writes address F, which depends on location E at the beginning of the epoch. Determining the complete relationship between global sources and global sinks requires aggregating the data from the local DIFT phase of each epoch.

3.3.3.1 Parallelizing aggregation: A failed attempt

To meet our performance goals, both the local DIFT and aggregation phases must scale well with the number of cores. Our first approach to constructing a parallel aggregation phase was based on a tree-like merge of local DIFT information. First, each individual epoch produces a map of all $\langle source, sink \rangle$ tuples where sources and sinks may be either local or global. For each sink with a taint identifier that represents a merge node, the map is generated by a depth-first traversal of the tree rooted at that node to resolve the set of source identifiers. This step is performed in parallel for all epochs, and caching is used to avoid revisiting tree nodes. If both the source and sink in a tuple are global, then the tuple is immediately output and removed from the map. Such tuples represent dependencies that can be computed solely on the DIFT information in a local epoch.

Next, we merge maps for pairs of adjacent epochs. For all locations, L, if there exists a tuple $\langle source, L \rangle$ in the first epoch and a tuple $\langle L, sink \rangle$ in the second epoch, then this step adds the tuple $\langle source, sink \rangle$ to its map. Since two epochs are involved, this step is parallelized across two cores. As before, if the sources and sinks in a tuple are both global, the tuple is immediately output and removed from the map. Merges are performed in a binary tree, merging sets of 4, 8, 16, etc. epochs, using the same approach as above. The number of cores participating in each merge grows proportionally, and the number of merge steps is logarithmic in the number of epochs.

Unfortunately, this algorithm performed very poorly. Traversing the merge log for each end value and generating sets of start values was extremely time-consuming, even with caching and reuse of intermediate results. Even worse, for most of our applications, some of the merged maps failed to fit in 256GB of memory. Our analysis showed that the reason for this behavior was that we were doing far more work than we needed to: the vast majority of merge nodes visited and values in the merged maps were not actually on a path between a global source and a global sink. However, because no epoch knew the full set of global sources and sinks when creating or merging maps, each had to calculate all dependencies that could possibly be used.

We concluded from this failed attempt that a fully-parallel aggregation phase is infeasible because it vastly increases the total work done. To fix this, aggregation must use

data about global sources and sinks to generate less intermediate data and to traverse fewer merge nodes.

3.3.3.2 Backward pass

Our next approach to aggregation was to structure the computation as a stream processing algorithm that scales via pipeline-style parallelism. We arrange the epochs in an ordered chain. In parallel, each epoch processes any global sinks encountered during the epoch. The JetStream aggregator checks the taint identifier for each byte sent to a global sink. If the taint identifier is a global source (i.e., if the global source and sink are in the same epoch), the aggregator immediately outputs a $\langle globalsource, globalsink \rangle$ tuple. If the taint identifier is a local source identifier L, the aggregator sends a $\langle L, globalsink \rangle$ tuple to the *previous* epoch in the chain. Epochs on the same machine communicate via a shared memory buffer; epochs on different machines communicate via a TCP socket. If the taint identifier is a merge log node, the aggregator resolves the set with a depth-first traversal of the tree rooted at that node. For each unique source identifier in the set, it either outputs a $\langle globalsource, globalsink \rangle$ tuple or sends a $\langle L, globalsink \rangle$ tuple to the previous epoch.

When the aggregation phase for an epoch receives a $\langle L, globalsink \rangle$ tuple from the succeeding epoch, it checks the epoch's local sink taint identifier for L. This is either a local source identifier, a global source identifier, or a merge node identifier that resolves to a set of source identifiers. For each global source, the aggregator outputs a $\langle globalsource, globalsink \rangle$ tuple, and for each local source L', it sends a $\langle L', globalsink \rangle$ tuple to the preceding epoch.

The last epoch sends a sentinel value to its preceding epoch after it has finished processing its sinks; when an epoch reads the sentinel, its work is done as no more tuples will be forthcoming. It then sends the sentinel to its predecessor.

The last column of Table 3.1 shows the backward pass. Epoch 2 determines that O_0 depends on location E and passes that tuple to epoch 1. Epoch 1 determines that E depends on C, so passes the tuple $\langle O_0, C \rangle$ to epoch 0. Epoch 0 resolves the merge tree rooted at $M_{0,0}$ and outputs tuples relating O_0 with both I_0 and I_1.

The major advantage of this streaming algorithm is that no epoch will process a merge node or send a tuple to a proceeding epoch unless the node/tuple represents a location that propagates to some global sink according to the propagation function. In the example, no merge nodes in epoch 1 are visited. This vastly reduces the amount of aggregation work. The potential disadvantage of this algorithm is that we have added a sequential step; each tuple must flow from global sink to global source, passing through all intermediate epochs. Our results show that this has only a minor effect on overall query time since each core can

still process tuples in parallel. In other words, the latency of passing a tuple through all epochs in the chain is very small compared to the query time, just as the sequential time to execute a machine instruction in a pipelined CPU is trivial compared to the time spent operating with a full pipeline.

Our results show that this algorithm, which we will refer to as the *backward pass* produces reasonable aggregation costs for some simple applications/queries, but it still takes too long for complex applications/queries. The reason is that we are still visiting too many merge nodes and creating too many tuples that are not ultimately on the path between a source and a sink. Thus, we found it necessary to also add a streaming *forward pass* that propagates information about which locations are related to global sources along the chain of epochs.

3.3.3.3 Forward pass

The forward pass runs prior to the backward pass. For each epoch, the forward pass first calculates a *reverse index* that has the same vertexes as the merge log DAG but that has edges in the opposite direction. The reverse index is also a DAG; depth-first traversal from a given local or global source yields the set of sinks that depend on that source. Each epoch builds its reverse index by first visiting all merge log nodes in temporal order, then visiting all local sinks. This step is fully parallelized since the reverse index can be computed purely with local information for each epoch.

Next, for each byte read from a global source, the aggregator does a depth-first traversal of the reverse index to determine the set of local sinks that depend on any source. It passes these sink locations to the succeeding epoch in the chain (forward in time). Here, the aggregator is only determining that a given local sink is tainted by any global source; it is not identifying a particular global source that has tainted the local sink. Therefore, the aggregator passes a local sink to the succeeding epoch at most once, and it visits each node in the reverse index at most once. It sets a `visited` bit for each local sink and merge node to avoid duplicate work.

As the aggregator receives locations from the prior epoch, it does a depth-first traversal of the reverse index to determine which (if any) additional local sinks depend on that location. It sends the locations associated with those local sinks to the succeeding epoch. The aggregator also retains the complete set of locations obtained from the prior epoch; this *live set* is the set of all local sources that depend on any global source.

Similar to the backward pass, the first epoch sends a sentinel token as soon as it finishes processing global sources. Once an epoch receives the sentinel, its live set is complete; the epoch then sends the sentinel to its successor.

Benchmark	Replay Log Size (MB)	Replay Time (s)	Single Core DIFT Time (s)	Global Sources	Global Sinks	Dependencies
Gzip	0.03	2.98	109.23	64352941	48791393	36586765
Ghostscript	0.12	1.03	76.90	2514067	176009	14682254
Evince	2.90	13.47	234.30	10302852	104061604	346305
Nginx	30.65	4.75	196.51	10412627	35000000	5000000
MongoDB	37.02	22.79	309.99	8863855	116592809	76042962
OpenOffice	15.25	7.55	418.03	9946659	32110959	14599069
Firefox	24.80	67.42	1838.70	920029	1636119	131476

Table 3.2: JetStream Benchmarks.

In Table 3.1, the Live Set column shows the forward pass. At the end of the first epoch, locations A, B, and C depend on at least one global source. The second epoch adds Z to this set because it depends on A and adds E to this set because it depends on C. Additionally, the second epoch removes B from this set because its taint value was cleared.

Once an epoch knows its live set, it prunes its merge log. The aggregator processes merge log nodes sequentially. Any local source not in the live set for that epoch cannot depend on a global source. So, if a child of a merge node is a local source identifier, and the local source is not in the live set, the child is replaced by a NULL identifier. If a merge node has two NULL children, no members of its source set depend on a global source. Any identifier in the merge log that refers to such a node is also replaced with a NULL value. Essentially, this is a garbage collection in which any node known to be unrelated to a global source is removed. This garbage collection can substantially prune the merge log. Each epoch can run the prune in parallel once it knows its live set. Thus, the only sequential component of the forward pass is the propagation of live set values. In Table 3.1, epoch 1 prunes merge node $M_{1,0}$ because it does not depend on any global source. $M_{1,1}$ is updated to $\langle A_1, NULL \rangle$.

By inserting a forward pass, JetStream guarantees that all merge nodes processed and all tuples generated during the backward pass are on a path between a global source and global sink. This vastly reduces the number of nodes processed and tuples generated, making the backward pass more efficient. Note that although the forward pass itself must visit all nodes tainted by a global source (even those that do not lead to a global sink), the forward pass does much less work than the backward pass because it tracks only whether or not a location depends on a global source. It does not identify the specific source(s) on which the location depends.

3.3.3.4 Pre-pruning

JetStream uses one final optimization to improve aggregation performance. During an epoch, many values in memory or registers are overwritten before the epoch ends. If a

merge log node does not propagate to either a local or global sink, then it can be removed from the log based solely on information available from that epoch. We call this step *pre-pruning*. JetStream does pre-pruning via a mark-and-sweep garbage collection over the merge log. It iterates through all sinks; if a sink has the taint identifier of a merge log node, JetStream marks the merge node as referenced. Then, JetStream iterates backward through the merge log. For each child in a merge log entry that refers to a prior merge log node, JetStream marks the prior merge log node as referenced. It discards all unmarked nodes and compacts the merge log. This reduces the number of merge log nodes that need to be processed later during both the forward and backward passes.

3.3.3.5 Summary

For each epoch, JetStream performs the following operations: (1) It runs the program without instrumentation from the start or from the nearest checkpoint to the beginning of the epoch. (2) It attaches a Pin tool and performs a local DIFT until the end of the epoch. (3) It pre-prunes the resulting local DIFT output to eliminate merge log nodes that cannot lead to a global sink. (4) It performs a forward aggregation pass to further prune the merge log by excluding any node that does not depend on a global source. (5) It performs a backward aggregation pass to generate $\langle globalsource, globalsink \rangle$ tuples; only merge nodes and locations on the path between a source and a sink are visited during this pass. Almost all of these steps can be performed in parallel for each epoch. The exceptions are the propagation of source dependencies in the forward path and $\langle location, sink \rangle$ tuples in the backward pass. These sequential steps are structured as stream processing along the epoch chain to maximize the work done in parallel.

3.4 Evaluation

Our evaluation answers the following questions:

- How well does JetStream scale DIFT?
- What are the remaining scalability bottlenecks?
- What is the impact of query optimizations?

3.4.1 Experimental Setup

JetStream uses the Arnold record and replay system [23] and the Pin dynamic instrumentation framework [53]. We evaluated JetStream using a CloudLab [71] cluster of 32 r320 machines (8-core Xeon E5-2450 2.1GHz processors, 16GB RAM, 10Gb NIC). We

envision running JetStream on an even larger cluster, but we could only reliably get a 32 machine cluster from CloudLab. Since these machines have a relatively small amount of RAM (16GB) and DIFT queries are memory-intensive, we use only 4 cores per machine, leaving the experimental setup with 128 effective cores. For all experiments, we report the mean of 5 trials and show 95% confidence intervals.

3.4.2 Benchmarks

We evaluate JetStream with seven benchmarks chosen to represent common desktop and server workloads:

- **Gzip** – Zip a large file.
- **Ghostscript** – Convert a research poster from PostScript to PDF.
- **Evince** – Open and view a research paper.
- **Mongodb** – Yahoo cloud server benchmark [18].
- **Nginx** – Serve static content.
- **OpenOffice** – Edit a conference presentation.
- **Firefox** – A long Facebook browsing session.

For Gzip, Ghostscript, Evince, Mongodb, and Nginx, the query asks for dependencies between all command line, network, and file system inputs and all such outputs. Running the all-to-all query for OpenOffice and Firefox generated over 1TB of data before we stopped the query. Thus, the OpenOffice query only considers file system data from the user's home directory to be sources, and the Firefox query considers cookie data to be sources and network output to specific sites (about 10% of total output) to be sinks. We use Arnold's data flow propagation function for all queries.

Table 3.2 shows a summary of the benchmarks. These are complex queries: most consider millions of distinct source and sinks, and most generate millions of dependencies. We show the time to replay each benchmark without instrumentation and the sequential DIFT time on a replay as baselines. We do not show the time for the original benchmark to run because that time depends on user think-time (for interactive applications), network delays, idle time (for server applications) and external output. When the benchmark is not CPU bound, DIFT overheads can be underestimated. Replay time, against which we compare, can already be one or two orders of magnitude faster than the original execution time [23]. We also report the compressed replay log size for each benchmark.

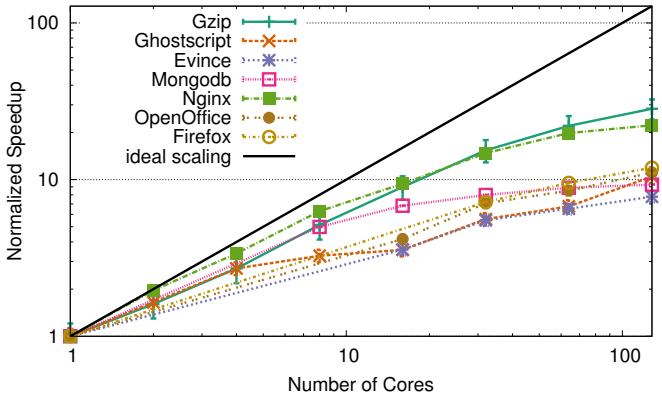


Figure 3.1: First Query Scalability - JetStream's scalability from 1 to 128 cores

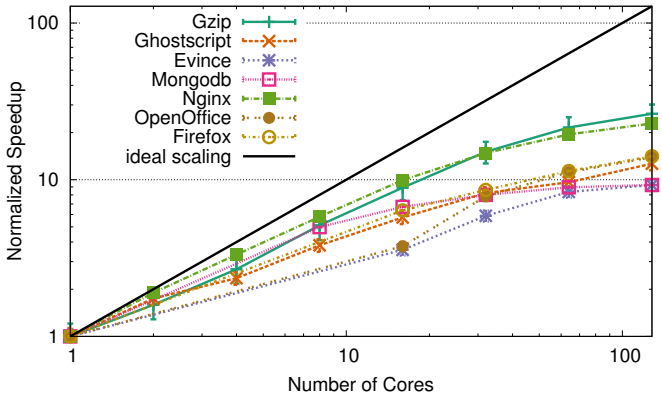


Figure 3.2: Scalability After Repartitioning - JetStream's scalability after incorporating its improved partitioning model

3.4.3 Scalability

We first evaluate the scalability of JetStream queries. Figure 3.1 shows the speedup of executing the first query for each benchmark on a log-log scale as we vary the number of cores from 1 to 128. Results are normalized to evaluating a query using a sequential algorithm on a single core; the black diagonal line shows ideal speedup, and a horizontal line would show no speedup. Overall, JetStream accelerates DIFT queries by 8–28x with a mean of 13x using 128 cores. All benchmarks continue to scale through 128 cores, but some (e.g., Evince) scale less well at high numbers of cores.

We find that the biggest bottlenecks to scalability of the first query are (1) the epoch partitioning is often imbalanced, resulting in delays due to tail latency, and (2) the fast forward time becomes a bottleneck as query time approaches the replay time (since we need to replay the application from the beginning to start an epoch).

We address the first bottleneck by gathering data about unique instructions executed during the first query and improving the partitioning. Figure 3.2 shows the impact of repar-

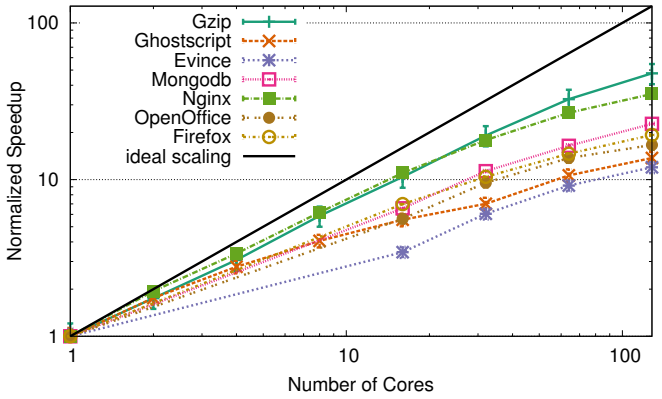


Figure 3.3: Second Query Scalability - JetStream's scalability after improving partitioning and checkpointing

titioning for the second query. With this optimization, JetStream scales the DIFT queries by 9–26x, with a mean of 14x. Repartitioning improves performance for all benchmarks except Gzip; in the case of Gzip, the model generated with less-detailed statistics is actually a better predictor of performance than the model generated with more-detailed statistics.

We address the second bottleneck by taking intermediate checkpoints during the first query. Figure 3.3 shows the scalability of the second query when using both repartitioning and checkpointing. JetStream scales the DIFT queries by 12–48x, with a mean of 21x. All benchmarks continue to scale up to 128 cores, though the pace of scaling diminishes with larger number of epochs. At 128 cores, the Gzip and Mongodb queries execute faster than their sequential replay times, and all benchmarks except Ghostscript execute faster than the original execution time of the application.

3.4.4 Analysis of First-Query Bottlenecks

Next, we examine results for individual benchmarks in more detail and identify scalability bottlenecks. Figures 3.4 and 3.5 show stacked bar graphs for each benchmark at 128 cores; results for the first query are in the left column, and results for the second query are in the right column.

Each stacked bar in a graph shows the time spent in different query stages for a single epoch; the epochs are ordered left to right by the order of the time slices in the application execution. The bottom region, labeled fast forward, shows the time for application execution to reach the start of the epoch. Instrumentation is the time required for Pin to instrument instructions, and analysis is the time to execute that instrumentation. The split between these two values is estimated by assuming that the instrumentation cost is equal to the unique instructions executed term from the model in Section 3.3.2 (which has an R^2

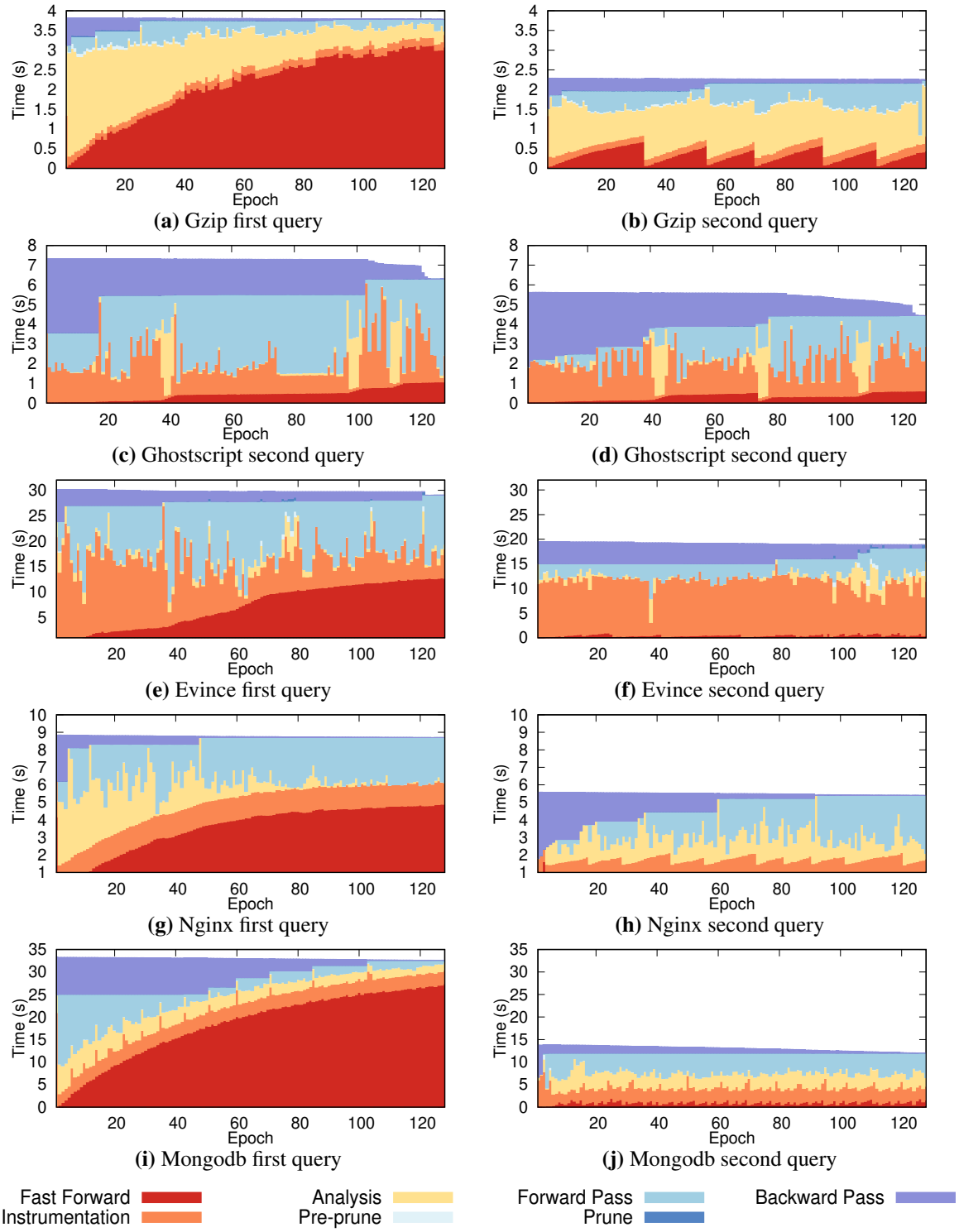
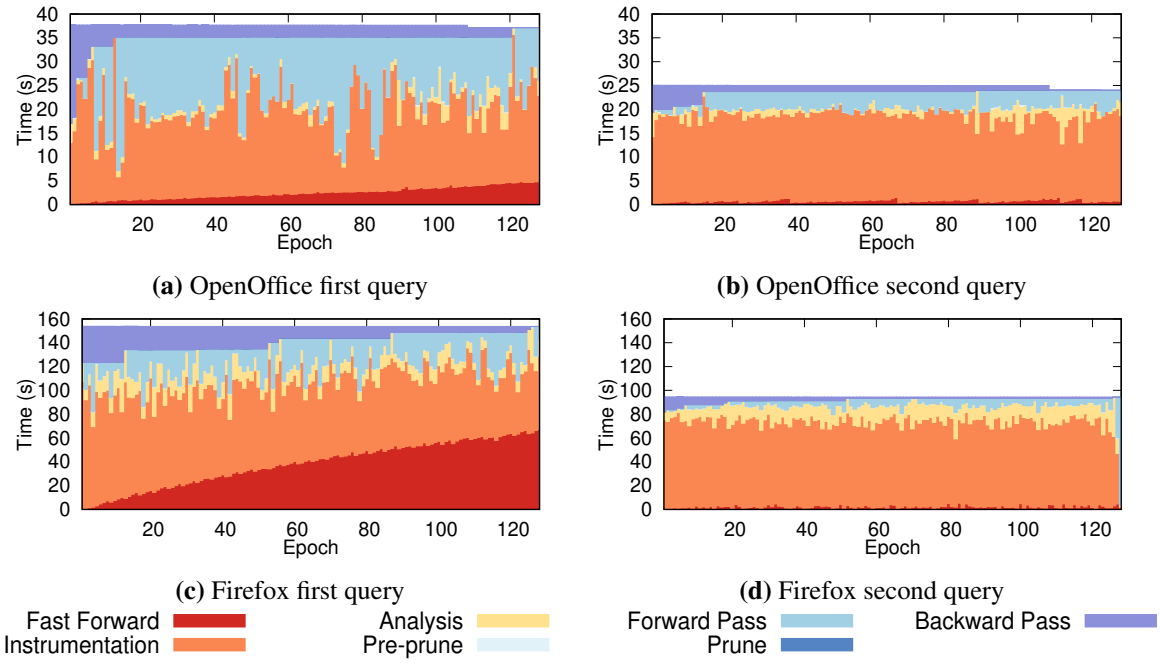


Figure 3.4: Breakdown of Query Processing Time for 128 Cores. Each stacked bar shows one core processing an epoch. From bottom to top, the shaded regions within each bar show the time spent fast forwarding, doing dynamic instrumentation, running DIFT, pre-pruning, performing the forward pass, pruning the merge log and performing the backward pass.



(a) OpenOffice first query

(b) OpenOffice second query

(c) Firefox first query

(d) Firefox second query

Figure 3.5: Breakdown of Query Processing Time for 128 Cores. Each stacked bar shows one core processing an epoch. From bottom to top, the shaded regions within each bar show the time spent fast forwarding, doing dynamic instrumentation, running DIFT analysis, pre-pruning, performing the forward pass, pruning the merge log and performing the backward pass.

value of 0.989) as we cannot directly distinguish these two values.

Pre-prune, forward pass, prune, and backward pass show the time spent in each aggregation stage. The sequential constraints of the forward pass and backward pass are shown by the gently sloping lines at the top of each region: one epoch's forward or backward pass cannot complete until the prior epoch in that pass has completed. The total time to complete the query is given by the height of the first stacked bar; the first epoch is the last to complete aggregation because of the sequential nature of the backward pass.

Outlier epochs caused by the result of poor partitioning can be detected by variance in the tops of the analysis regions (the combination of the fast forward, instrumentation, and analysis phases). All of our benchmarks except Gzip and Mongodb noticeably benefit from improved partitioning. For example, comparing the first and second queries of OpenOffice (Figures 3.5a and 3.5b) shows the benefit of improving the partitioning between the first and second queries. Reducing outliers leads to substantially faster second query times for these benchmarks.

The effects of checkpointing in JetStream can be seen in all of our benchmarks. For example, comparing the first and second queries of Gzip (Figures 3.4a and 3.4b) shows the dramatic effect that checkpointing can have on query latency. The primary reason that

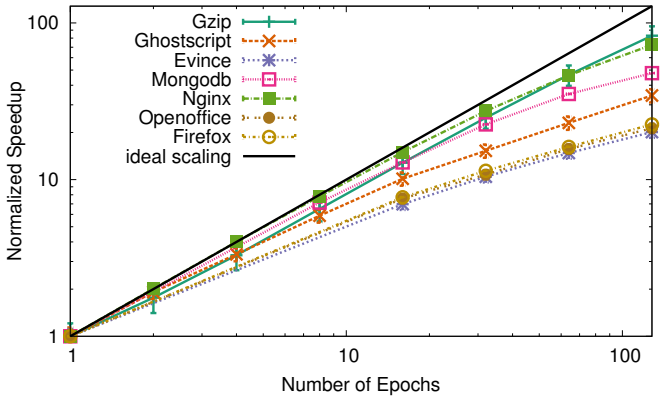


Figure 3.6: DIFT Scaling - Scalability of local DIFT (excluding fast-forward time) for different numbers of cores normalized to local DIFT (excluding fast-forward time) for one core.

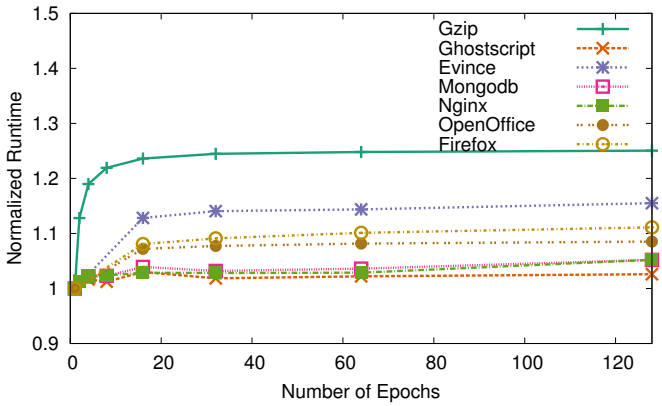


Figure 3.7: Retainting Overhead - Overhead of tainting local sources at the beginning of each epoch.

this benchmark does not scale well for the first query is that the query time approaches the replay time of the benchmark—this is shown by fast forward being a large component of the last epoch time for the first query. In contrast, the fast forward times in the second query are much smaller.

3.4.5 Analysis of Second-Query Bottlenecks

We next look at second query performance and bottlenecks. Interestingly, the specific bottlenecks vary from benchmark to benchmark.

For Evince (Figure 3.4f), OpenOffice (Figure 3.5b), and Firefox (Figure 3.5d), Pin instrumentation time dominates total query time. Pin instrumentation time also impacts Ghostscript (Figure 3.4d) to a lesser degree. To explore this issue in more detail, Figure 3.6 shows the speedup for just instrumentation and analysis. All benchmarks scale up to 128 cores, but not ideally.

There are two main factors that limit instrumentation and analysis scalability: (1) JetStream must taint all local sources at each epoch boundary, and (2) Pin instruments an instruction in every epoch in which that instruction occurs, so dividing the program into smaller epochs increases the total instructions instrumented. We isolated the cost of (1) by running a sequential query on one core that retaints each address at epoch boundaries. This does the exact same work as the parallel version, and it produces the same results; however, Pin instruments each instruction only once across all epochs. As Figure 3.7 shows, the overhead added by tainting local sources is relatively small (3–25% of the sequential DIFT query). When this overhead is parallelized over 128 cores, it should have little effect on query time. Additionally, our model from Section 3.3.2 shows that unique instructions correlate very highly with instrumentation and analysis time.

Switching from a dynamic to static instrumentation tool, using techniques that reduce the amount of dynamic instrumentation [36], or employing a low-overhead binary instrumentation platform (e.g., Protean code [46]) could reduce instrumentation time. Alternatively, we could take checkpoints that include already-instrumented code, as is done by Speck [65].

Poor partitioning is a significant component of overall query time for Ghostscript (Figure 3.4d), Nginx (Figure 3.4h), and Mongodb (Figure 3.4j). There are two separate reasons for poor partitioning in these benchmarks.

JetStream gathers statistics about query execution after each system call is executed. Ghostscript contains long regions of computation without a system call. At 128 epochs, JetStream must split some of these regions into multiple epochs. It divides these regions crudely based on the number of entries in the replay log; this crude metric often mispredicts the actual query execution time for the split epochs. Gathering statistics at finer-grained intervals would reduce outliers.

Outlier epochs in Nginx and Mongodb occur due to variance in the amount of taint processing done in each epoch. Outliers are correlated with large numbers of tainted sources and/or sinks in the epoch. Currently, our partitioning tool cannot determine which sources and sinks will be tainted in advance, but JetStream could potentially gather more statistics about sources and sinks during the first query, which could help the partitioning tool make such a determination for subsequent queries.

Aggregation plays a minor role in query time for most benchmarks. The speed of the forward pass is seen in the slope of the top of this region across epochs. Similarly, the speed of the backward pass is given by the slope of the top of that region. The total area for these two regions is less relevant since the sequential constraints mean that epochs will sometimes be idle waiting for data to arrive from predecessor epochs. We see negligible forward

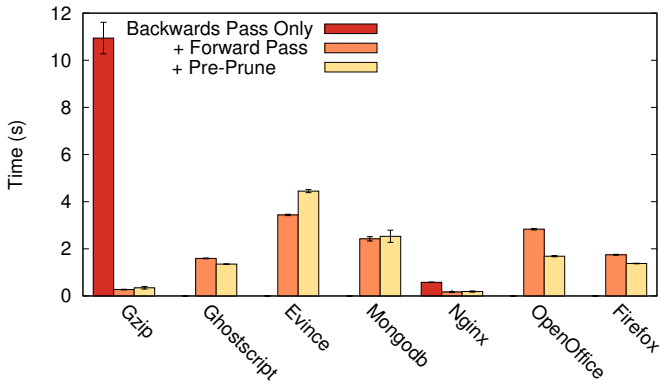


Figure 3.8: Optimization Effectiveness - Effect of aggregation optimizations at 128 cores. Experiments that did not complete are left blank.

pass time across all benchmarks. Backward pass time is most noticeable for Ghostscript (Figure 3.4d), Mongodb (Figure 3.4j) and OpenOffice (Figure 3.5b), as shown by the noticeable slope of the top region in each graph. Better caching heuristics may improve the backward pass for these benchmarks.

3.4.6 Optimizations

We next evaluate the costs and benefits of optimizations employed by JetStream. We first measure the benefit of two aggregation optimizations: the forward pass and pre-pruning. To isolate aggregation cost from outliers in the local DIFT stage, we let all epochs finish local DIFT before beginning aggregation. This is the worst case for aggregation costs since individual epochs cannot pre-prune or construct the reverse index while waiting for prior epochs to finish local DIFT.

Figure 3.8 shows the isolated cost of aggregation for 128 cores. If aggregation performs only the backward pass (omitting the forward pass and pre-pruning), then only Gzip and Nginx complete; aggregation runs out of memory on all other benchmarks. For Gzip and Nginx, adding the forward pass improves isolated aggregation time by 98% and 71%, respectively.

The pre-prune optimization appears less effective. It decreases isolated aggregation time for Firefox, OpenOffice, and Ghostscript, but increases it slightly for Mongodb, Evince, and Gzip. We conclude that JetStream's policy of always pre-pruning is likely suboptimal; an adaptive policy that only pre-prunes when spare CPU cycles are available would be better.

We also measured the extra costs to optimize partitioning. We measured the time to profile L1 instruction cache misses for CPU-intensive benchmarks (Gzip and Ghostscript);

the average overhead was 3.1%. The average overhead imposed by taking checkpoints during the first query was only 0.7% since each epoch takes at most one checkpoint. Finally, the average overhead of tracing unique instructions during the first query was 1.5%.

3.5 Conclusion

JetStream enables interactive DIFT over past executions by parallelizing queries across a cluster. It uses deterministic record and replay to divide an execution into epochs and execute a local DIFT for each epoch on a separate core. It aggregates results from local DIFTs by arranging epochs in a sequential chain according to the order of program execution and using a pipeline-like stream processing algorithm to pass information about global sources and sinks along the chain. For future work, we plan to explore novel debugging and forensics applications enabled by JetStream.

CHAPTER IV

Sledgehammer: Cluster-Fueled Debugging

In this chapter, we show how to generalize the lessons and architecture from JetStream to accelerate general-purpose debugging questions. Debugging is onerous and time-consuming, comprising roughly half of all development time [57]. It involves detective work: using the tools at their disposal, a developer searches a program execution for clues about the root cause of correctness or performance problems.

Current debugging tools force developers to choose between power and interactivity. Tools such as gdb are interactive: developers can inspect program values, follow execution flow, and use watchpoints to monitor changes to specific locations. For many simple bugs, interactive debuggers like gdb allow developers to quickly identify root causes by asking and answering many low-level questions about a particular program execution.

Yet, complex bugs such as wild stores, synchronization errors, and other heisenbugs are notoriously hard to find. Consider a developer trying to uncover the root cause of nondeterministic data corruption in a Web server. They cannot use gdb because they do not yet know which values to inspect or which part of the server execution to monitor. So, they employ more heavyweight tools. They add logging message and sprinkles functions to verify invariants or check data structures at various points in the server code.

Custom tools like logging and invariant checks are powerful, but they are definitely not interactive. First, the developer must execute a program long enough for a bug to occur. Complex bugs may not be evinced with a simple test case; e.g., rare heisenbugs may require lengthy stress testing before a single occurrence. Second, detailed logging and custom predicates slow down program execution, sometimes by an order of magnitude. This means that each new question requires a long wait until an answer is delivered, and diagnosing a root cause often requires asking many questions.

Ideally, our developer would have tools that are both powerful and interactive. Then, they could ask complex questions about their server execution and receive an answer

in a few seconds. Yet, the tradeoff seems fundamental: these powerful tools are time-consuming precisely because they require substantial computation to answer complex questions about long program executions.

Cluster-fueled debugging solves this dilemma: it provides interactivity for complex tools by parallelizing their work across many cores in a compute cluster. With sufficient scale, developers see answers to even detailed queries in a few seconds, so they can quickly iterate to gather clues and identify a root cause.

Sledgehammer is the first general cluster-fueled debugger. It is designed to mirror current debugging workflows: i.e., adding logging [91] or invariant checks, re-compiling, re-executing to reproduce the problem, and analyzing the output of the additional instrumentation. However, Sledgehammer produces results much faster through parallelization of instrumentation and analysis. Like prior academic [43, 79, 83] and commercial [66] tools, Sledgehammer is replay-based; i.e., it can deterministically reproduce any previously-recorded execution on demand for debugging. Replay facilitates iterative debugging because each question is answered by observing the same execution, ensuring consistent answers.

Sledgehammer uses deterministic replay for another purpose: it time-slices a recorded program execution into distinct chunks called *epochs*, and it runs each epoch on a different core. It uses `ptrace` to inject debugging code, called *tracers* into program execution. Vitally, Sledgehammer provides isolation so that tracers do not modify program behavior, guaranteeing that each replayed execution is consistent with the original recording. Because tracers are associated with specific points in the program execution and the execution is split across many cores, the overhead of both tracer execution and isolation is mitigated through parallelization.

Tracers may produce large amounts of data for complex debugging tasks, and processing this data could become a bottleneck. So, Sledgehammer also provides several options to parallelize data analysis. First, local analysis of each epoch can be performed on each core. Second, stream-based analysis allows information to be propagated from preceding epochs to subsequent epochs, allowing further refinement on each core. Finally, tree-based aggregation, terminating in a global analysis step, produces the final result.

Cluster-fueled debugging makes existing tools faster. Retro-logging [17, 38, 85] lets developers change logging in their code and see the output that would have been produced if the logging had been used with a previously-recorded execution. Retro-logging requires isolating modified logging code from the application to guarantee correct results. Both isolation and voluminous logging add considerable overhead. We introduce *parallel retro-logging*, which hides this overhead through cluster-fueled debugging to make retro-logging

interactive.

Cluster-fueled debugging enables new, powerful debugging tools that were previously infeasible due to performance overhead. To demonstrate this, we have created *continuous function evaluation*, which lets developers define a function over the state of their execution that is logically evaluated after every instruction. The tool returns each line of code where the function return value changes. Continuous function evaluation mirrors the common debugging technique of adding functions that verify invariants or check data structure integrity at strategic locations in application code [21], but it frees developers from having to carefully identify such locations to balance performance overhead and the quality of information returned.

We have also created *parallel retro-timing*, which lets developers retroactively measure timing in a previously-recorded execution (a feature not available in prior replay-based debugging tools). Sledgehammer returns timing measurements as a range that specifies minimum and maximum values that could have been returned during the original execution.

This paper makes the following contributions:

- We present a general framework for parallelizing complex debugging tasks across a compute cluster to make them interactive.
- Parallelization makes scalability a first-class design constraint for debugging tools, and we explore the implications of this constraint.
- We introduce continuous function evaluation as a new, powerful debugging tool made feasible by Sledgehammer parallelization and careful use of compiler instrumentation and memory protections.
- We explore the fundamental limits of parallelization and show how to alleviate the bottlenecks experienced when trying to scale debugging.

We evaluate Sledgehammer with seven scenarios debugging common problems in memcached, MongoDB, nginx, and Apache. With 1024 compute nodes, Sledgehammer returns the same results as sequential debugging, but parallelization lets it return answers 416 times faster on average. This makes very complex debugging tasks interactive.

4.1 Usage

To use Sledgehammer, a developer records the execution of a program with suspect behavior for later deterministic replay. Recording could occur during testing or while reproducing a customer problem in-house. Deterministic replay enables parallelization. It also makes results from successive replays consistent, since each replay of the application executes the same instructions and produces the same values on every replay.

Next, the developer specifies a debugging query by adding *tracers* to the application source code. A tracer can be any function that observes execution state and produces output. Examples of tracers are logging functions, functions that check invariants, and functions for measuring timing. A tracer can be inserted at a single code location, inserted at multiple locations, or evaluated continuously. Thus, tracers are added in much the same way that developers currently add logging messages or invariant checks to their code.

A developer can also add *analyzers* to aggregate tracer output and produce the final result; e.g., an analyzer could filter log messages or correlate events to identify use-after-free bugs. Sledgehammer provides several ways to parallelize analysis. Developers can write local analyzers that operate only on output from one epoch of program execution, stream analyzers that propagate data between epochs in the order of program execution, and tree-based analyzers that combine per-epoch results to generate the final result over the entire execution.

In summary, the interface to Sledgehammer is designed to be equivalent to the current practice of adding logging/tracing code and writing analysis code to process that output. However, Sledgehammer uses a compute cluster to parallelize application execution, instrumentation, and analysis, and, in our setup, produces answers in a few seconds, instead of minutes or hours.

4.2 Debugging Tools

We have created three new parallel debugging tools.

4.2.1 Parallel Retro-Logging

Retro-logging [17, 38, 85] lets developers modify application logging code and observe what output would have been generated had that logging been used during a previously-recorded execution. We implement parallel retro-logging by adding tracers to the application code that insert new log messages; often tracers use the existing logging code in the application with new variables. Log messages are deleted via filtering during analysis, and log messages are modified by both inserting a new log message and filtering out old logging.

Parallelizing retro-logging has several benefits. First, the application being logged may run for a long time, and verbose logging causes substantial performance overhead. Second, even carefully-written logging code perturbs the state of the application in subtle ways, e.g., by modifying memory buffers and advancing file pointers. If left unchecked, these subtle differences cause the replayed execution to diverge from the original, which can

prevent the replayed execution from completing or silently corrupt the log output with incorrect values. Isolation is required for correctness, and the cost of isolation is high. This cost is not unique to Sledgehammer: tools such as Pin [53] and Valgrind [63] that also isolate debugging code from the application have high overhead. Sledgehammer hides this overhead via parallelization.

4.2.2 Continuous Function Evaluation

Continuous function evaluation logically evaluates the output of a specified function after every instruction. It reports the output of the function each time the output changes and the associated instruction that caused the change. Continuous function evaluation can be used to check data structure invariants or other program properties throughout a recorded execution.

Actually evaluating the function after each instruction would be prohibitively expensive, even with parallelization. Sledgehammer uses static analysis to detect values read by the function that may affect its output and memory page protections to detect when those values change. This reduces performance overhead to the point where parallelization can make this debugging tool interactive.

4.2.3 Retro-Timing

Many debugging tasks require developers to understand the timing of events within an execution. Replay debugging recreates the order of events, but not event timing. Thus, a recorded execution is often useless for understanding timing bugs.

Sledgehammer systematically captures timing data while recording an execution. To reduce overhead of frequent time measurements, it integrates time recording with the existing functionality for recording non-deterministic program events. When debugging, developers call `RetroTime`, a Sledgehammer provided function that returns bounds on the clock value that would have been read during the original execution. These bounds are determined by finding the closest time measurements in the replay log.

4.3 Scenarios

We next describe seven scenarios that show how Sledgehammer aids debugging. We use these scenarios as running examples throughout the paper and measure them in our evaluation.

```
1  bool is_corrupt (item *head, item *tail, int size) {
2   int count = 0;
3
4   while (tail->prev != NULL)
5    tail = tail->prev;
6   if (tail != head) return true;
7
8   while (head != NULL) {
9    head = head->next;
10   count++;
11  }
12  return (count != size);
13 }
14
15 char *check_ all_lists () {
16  for (int i = 0; i < SIZE; ++i)
17   if (is_corrupt (heads[i], tails[i], sizes[i]))
18    CFE_RETURN (``1'');
19  CFE_RETURN (``0'');
20 }
```

Figure 4.1: Tracer for the First Memcached Query.

4.3.1 Atomicity Violation

Concurrency errors such as atomicity violations are notoriously difficult to find and debug [51]. In this scenario, a memcached developer finds an error message in memcached's production log indicating an inconsistency in an internal cache. Memcached uses parallel arrays, `heads`, `tails` and `sizes`, to manage items within the cache. For each index, `heads[i]` and `tails[i]` point to the head and tail of a doubly-linked list, and `size[i]` holds the number of list items.

To use Sledgehammer, the developer first records an execution of memcached that exhibits the bug. Next, they decide to use continuous function evaluation and writes tracers to identify the root cause of the bug. To illustrate this process, we used existing assert statements in the memcached code to write the sample tracer in Figure 4.1. The `is_corrupt` function validates the correctness of a single list. The `check_all_lists` function returns "1" if any list is corrupt and "0" otherwise.

By adding `SH_Continuous(check_all_lists)` to the memcached source, the developer specifies that `check_all_lists` should be evaluated continuously. This outputs a line whenever the state of the lists transitions from valid to invalid, or vice versa. The `CFE_RETURN` macro prepends to each line the thread id and instruction pointer where the transition occurred.

The developer then writes an analysis function; we show the function they would write

```
1  void analyze (int in, int out) {
2   FILE *inf = fdopen(in), outf = fdopen(out);
3   map<int, int> invalid_count;
4   char line[128];
5   int location, tid, count;
6
7   while (getline(&line, NULL) > 0) {
8    sscanf(``%x:%x:%x\n'', &location, &tid, &count);
9    if (count) invalid[location] += count;
10  }
11  for (auto &it : invalid)
12   fprintf(outf, ``%x:%x\n'', it.first, it.second);
13 }
```

Figure 4.2: Analyzer for the First Memcached Query.

```
1  char* check_all_lists () {
2   for (int i = 0; i < SIZE; ++i)
3    if (check(heads[i], tails[i], sizes[i]))
4     CFE_APPEND (``invalid:%x\n'', locks[i]);
5    else
6     CFE_APPEND (``valid:%x\n'', locks[i]);
7   CFE_RETURN();
8  }
9
10 void hook_lock (pthread_mutex_t *mutex) {
11  tracerLog(``0:%x:lock:%x\n'', tracerGettid(), mutex);
12 }
13
14 void hook_unlock (pthread_mutex_t *mutex) {
15  tracerLog(``0:%x:unlock:%x\n'', tracerGettid(), mutex);
16 }
```

Figure 4.3: Tracer for the Second Memcached Query.

in Figure 4.2. This function reports all code locations where a transition to invalid occurs. We wrote this function so that the same code can be used for local and tree analysis.

Running this query doesn't reveal the root cause of the bug, as each transition to invalid occurs at a code location that is supposed to update the cache data structures. So, the developer next suspects a concurrency bug. The cache is updated in parallel; for each index `i`, a lock, `locks[i]`, should be held when updating the parallel arrays at index `i`. Thus, there are two invariants that should be upheld: whenever the arrays at index i become invalid, `locks[i]` should be held, and whenever `locks[i]` is released, the arrays should be valid.

Figure 4.3 shows how the developer would modify the tracer for a second query. The `check_all_lists` function now appends the validity and lock for each item in the list to a

string and returns the result. The developer also adds two functions that report when cache locks are acquired and released. They add two more statements to the memcached source code to specify that these functions should run on each call to `pthread_mutex_lock` and `pthread_mutex_unlock`.

Figure 4.4 shows the new analysis routine that the developer would write. The analyzer is structured like a state-machine; each line of input is a transition from one state to the next. `lockset` tracks the locks currently held and `needed_locks` tracks which locks must be held until lists are made valid again. Line 14 checks the first invariant mentioned above, and line 28 checks the second.

We again use the same analyzer for both local and tree-based analysis. Since local analysis occurs in parallel, a needed lock may have been acquired in a prior epoch, and locks held at the end of an epoch may be needed in a future epoch. Thus, the analyzer outputs all transitions that it can not prove to be correct based on local information, as well as information that may be needed to prove transitions in subsequent epochs correct. The global analyzer at the root of the tree has all information, so any transition it outputs is incorrect.

In our setup, the query returns in a few seconds and identifies two instructions where an array becomes invalid while the lock is not held. One occurs during initialization (and is correct because the data structure is not yet shared). The other is the atomicity bug.

4.3.2 Apache 45605

In this previously reported bug [11], a Apache developer noted that an assertion failed during stress testing. The assertion indicated that a thread pushed too many items onto a shared queue. Without Sledgehammer, developers spent more than two months resolving the bug. They even proposed an incorrect patch, suggesting that they struggled to understand the root cause.

Four unsigned integers, `nelts`, `bounds`, `idlers` and `max_idlers`, control when items are pushed onto the queue. By design, `nelts` should always be less than `bounds`, and `idlers` should always be less than `max_idlers`. We emulated a developer using Sledgehammer to debug this problem by writing a tracer that uses continuous function evaluation to check these relationships and an analyzer that lists instructions that cause a relationship to no longer hold.

The query returns a single instruction that decrements `idlers`. As this result is surprising, we modified the query to also output the value of each integer when the transition occurs. This shows that the faulty instruction causes an underflow by decrementing `idlers` from 0 to `UINT_MAX`. From this information, the developer can realize that `idlers` should

```
1  void analyzer (int in, int out) {
2   FILE *inf = fdopen (in), outf = fdopen (out);
3   map<int, set<int>> lockset;
4   map<int, set<int>> needed_locks;
5   char line[128], type[8];
6   int thread, ip, lock;
7
8   while (getline (&line, NULL) > 0) {
9    sscanf (``%x:%x:%s:%x'', ip, thread, type, lock);
10
11   if (!strcmp (type, ``lock''))
12    lockset[thread].insert (lock);
13   } else if (!strcmp(type, ``invalid'')) {
14    if (lockset[thread].contains (lock))
15     needed_locks[thread].insert (lock);
16    else
17     fprintf (outf, line);
18   } else if (!strcmp (type, ``valid'')) {
19    if (needed_locks[thread].contains (lock))
20     needed_locks[thread].remove (lock);
21    else
22     fprintf (out, line);
23   } else if (!strcmp (type, ``unlock'') {
24    if (lockset[thread].contains (lock))
25     lockset[thread].remove (lock);
26    else
27     fprintf (outf, ``%s'', line);
28    if (needed_locks[thread].contains (lock))
29     fprintf (outf, ``BUG: atomicity violation: %x\n'', ip);
30   } else {
31    fprintf(outf, ``%s'', line);
32   }
33  }
34
35  for (const auto &l_set : lockset)
36   for (lock : l_set.second)
37    fprintf (outf, ``lock:%x:%x\n'', l_set.first, lock);
38  for (const auto &l_set : needed_locks)
39   for (lock : l_set.second)
40    fprintf (outf, ``invalid:%x:0:%x\n'', l_set.first, lock);
41 }
```

Figure 4.4: Analyzer for the Second Memcached Query.

never be 0 when the instruction is executed and that the root cause is that the preceding `if` statement should be a `while` statement.

4.3.3 Apache 25520

In another previously-reported bug [10], an Apache developer found that the server log was corrupted after stress testing. Apache uses an in-memory buffer to store log messages and flushes it to disk when full. With Sledgehammer, a developer could debug this issue by writing a tracer that uses continuous function evaluation to validate the format of log messages in the buffer. The analyzer identifies instructions that transition from a correctly-formatted buffer to an incorrectly-formatted one. Running this query returns only an instruction that updates the size of the buffer after data has been copied into the buffer. This indicates that the buffer corruption occurs during the data copy before the size is updated.

Thus, the developer next writes a query to detect such corruption by validating the following invariant: each byte in the buffer should be written no more than once between flushes of the buffer to disk. The continuous function evaluation tracer returns a checksum of the entire buffer region (so that all writes to the buffer region are detected irrespective of the value of the size variable) and the memory address triggering the tracer (see `tracerTriggerMemory()` in Section 4.4.2). The developer also writes a tracer that hooks calls to the buffer flush function. The analyzer outputs when multiple writes to the same address occur between flushes. The output shows that such writes come from different threads, identifying a concurrency issue in which the instructions write to the buffer without synchronization.

4.3.4 Data Corruption

Memory corruption is a common source of software bugs [48] that are complex to troubleshoot; often, the first step in debugging is reproducing the problem with more verbose logging enabled. In this scenario, an nginx developer learns that the server very infrequently reports corrupt HTTP headers during stress testing, even though no incoming requests have corrupt headers. Without Sledgehammer, he would enable verbose logging and run the server for a long time to try to produce a similar error. Reproduction is painful; verbose logging adds considerable slowdown and produces gigabytes of data.

With Sledgehammer, the developer uses parallel retro-logging to enable the most verbose existing nginx logging level over the failed execution recorded during testing. In nginx, this requires adding tracepoints in two dedicated logging functions. Each tracer calls a low-level nginx log function after specifying the desired level of verbosity. The de-

veloper also filters by regular expression to only collect log messages pertaining to HTML header processing. The same filter code can be run as a local analyzer without modification, so parallelization is trivial. In our setup, the Sledgehammer query returns results in a few seconds, and the developer notes that corruption occurs between two log messages. This provides a valuable clue, but the developer must iteratively add more logging to narrow down the problem. Fortunately, these messages can be added retroactively to the same execution; the resulting output is seen in a few seconds.

4.3.5 Wild Store

Wild stores, i.e., stores to invalid addresses that corrupt memory, are another common class of errors that are reportedly hard to debug [48]. In this scenario, Mongodb crashes and reports an error due to a corruption in its key B-tree data structure. Mongodb has an existing debugging function that walks the B-tree and checks its validity. Without Sledgehammer, the developer must sprinkle calls to this function throughout the code, re-run the application to reproduce the rare error, and try to catch the corruption as it happens. Unfortunately, the corruption was introduced during processing of a much earlier request and lay dormant for over 10 seconds. Further, the wild store was performed by an unrelated thread, so it takes numerous guesses and many iterations of running the program to find the bug.

With Sledgehammer, the developer specifies that the existing Mongodb debugging function should be evaluated continuously. Since the B-tree is constantly being modified, its validity changes often in the code that adds and deletes elements. The developer therefore writes a simple analyzer that counts the number of transitions that occur at each static instruction address. The same code is used for both local and tree-based analyzers. The query returns in under a minute. It reports three code locations where the data structure becomes invalid exactly once: two are initialization and the third is the wild store.

4.3.6 Memory Leak

Memory leaks, double frees, and use-after-free bugs require reasoning about an execution's pattern of allocations and deallocations. In this scenario, an nginx developer notes that a large code change has introduced very infrequent memory leaks that lead to excessive memory usage for long-running servers. One option for tracking down this bug is to run a tool like Valgrind [63] over a long execution with varied requests. Due to the overhead of Valgrind instrumentation, this takes many minutes to return a result over even a relatively short execution.

With Sledgehammer, the developer adds three tracers and hooks the entry and exit of

routines such as `malloc` and `free`. The analyzer matches allocations and deallocations and reports remaining unallocated memory. Parallelizing the analyzer is straightforward: the sequential analyzer can be used for tree-based and local analysis without modification. Stream analysis requires adding 10 lines of code to pass a list of allocated memory regions that have not yet been deallocated from epoch to epoch. In our setup, a Sledgehammer query identifies leaked memory in nginx in a few seconds.

4.3.7 Lock Contention

Rare performance anomalies are hard to debug. One common source of performance anomalies is lock contention [81]. Low-level timing data is informative, but gathering such data has high overhead and may prevent the anomaly from occurring. In this scenario, a memcached developer sees infrequent requests that have much longer latencies than expected. They run a profiler, but the tool reports only average behavior, which obscures the occasional outlier. So, they sprinkle timing measurements at key points in their code and re-runs the application many times to drill down to the root cause: lock contention with a background thread. Each run requires a long time to exhibit an anomaly and difficult analysis to determine which requests are outliers in each new execution.

With Sledgehammer, the developer runs a query that gathers `RetroTime` data at key points in request parsing, starting with existing timing code originally disabled during recording. Because queries are fast, they retroactively add even more timing code and can iterate quickly to drill down to the suspect lock acquisition. Their analysis function tracks time taken in each specified request phase, and compares breakdowns for the five longest requests with average behavior. A final query identifies the thread holding the contended lock by combining retro-timing with tracers that hook mutex acquisition and release. Analysis of the tracelog identifies the background thread that holds the lock on which the anomalous requests wait.

4.4 Design and Implementation

Figure 4.5 shows how Sledgehammer parallelizes debugging. The developer specifies (1) a previously-recorded execution to debug, (2) tracers that run during a replay of that execution, and optionally, (3) analysis functions that aggregate tracer output to produce a final result. Sledgehammer parallelizes the replayed execution, tracers, and analysis across many cores in a cluster.

Section 4.4.2 discusses how developers specify tracers by annotating their source code to add logging and instrumentation. Sledgehammer parses the source code to extract the

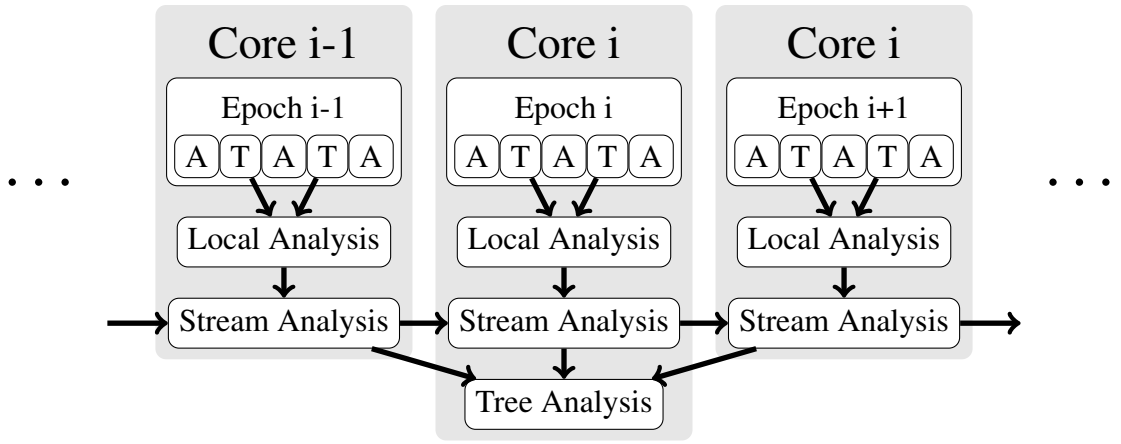


Figure 4.5: Sledgehammer Architecture Overview. A replay is divided into n epochs. Each core runs an epoch by executing the original application (A boxes) and injecting tracers (T boxes). Local analysis runs on each core with output from a single epoch, stream analysis takes input from previous epochs and sends output to subsequent ones. Tree analyzers combine output from multiple epochs.

tracers, the arguments passed to each tracer, and the locations where tracers should be invoked.

Section 4.4.3 describes how Sledgehammer prepares for query execution by distributing generic (non-query-specific) information needed to replay the execution to available compute nodes. It divides the execution into epochs of roughly-equal duration, where the number of epochs is determined by the number of cores available. Read-only data shared across epochs, e.g., the replay log, application binaries, and shared libraries, are read from a distributed file system. Sledgehammer caches these files on local-disk for improved performance on subsequent queries. The per-epoch state, e.g., application checkpoints at the beginning of each epoch, is generated in parallel, with each core generating its own state.

As discussed in Section 4.4.4, Sledgehammer runs a query by executing each epoch in parallel on a separate core. Epoch execution starts from a checkpoint and replays non-deterministic operations from the replay log to reproduce the recorded execution. Sledgehammer uses `ptrace` to insert software breakpoints at code locations where tracers should run. When a breakpoint is triggered, it runs the tracer in an isolated environment that rolls back any perturbation to application state after the tracer finishes. To support continuous function evaluation, Sledgehammer uses page protections to monitor memory addresses that may affect the return value of the function; it triggers a tracer when one of those addresses is updated.

Section 4.4.5 discusses how analyzers process the stream of output from tracers. As shown in Figure 4.5, Sledgehammer supports three types of analysis routines: local, stream,

and tree. Local analysis (e.g., filtering) operates on tracer output from a single epoch. Stream analysis allows information to be propagated from epochs earlier in the application execution to epochs later in the execution. Sledgehammer runs a stream analyzer on each compute node; each analyzer has sockets for reading data from its predecessor epoch and sending data to its successor. A tree analyzer combines input from many epochs and writes its output to `stdout`. For a large number of cores, these analyzers are structured as a tree with the root of the tree producing the final answer to the query. Thus, a purely sequential analysis routine can always run as the root tree analyzer.

4.4.1 Background: Deterministic Record and Replay

Sledgehammer uses deterministic record and replay both to parallelize the execution of a program for debugging, and also to ensure that successive queries made by a developer return consistent results. Deterministic replay [24] allows the execution of a program to be recorded and later reproduced faithfully on demand. During recording, all inputs from nondeterministic actions are written to a *replay log*; these values are supplied during subsequent replays instead of performing the nondeterministic operations again. Thus, the program starts in the same state, executes the same instructions on the same data values, and generates the same results.

Epoch parallelism [84] is a general technique for using deterministic replay to partition a fundamentally sequential execution into distinct epochs and then execute each epoch in parallel, typically on a different core or machine. Determinism guarantees that the result of stitching together all epochs is equivalent to a sequential execution of the program. Replay also allows an execution recorded on one machine to be replayed on a different machine. There are few external dependencies, since interactions with the operating system and other external entities are nondeterministic and replayed from the log.

Sledgehammer uses Arnold [23] for deterministic record and replay due to its low overhead (less than 10% for most workloads) and because Arnold supports epoch parallelism [69]. We modified Arnold to support ptrace-aware replay, in which Sledgehammer sets breakpoints and catches signals. We also modified Arnold to run tracers in an isolated environment where they can allocate memory, open files, generate output, etc. Our modifications roll back the effects of these actions after the tracer finishes to guarantee that the replay of the original execution is not perturbed, similar to prior systems that support inspection of replayed executions [17, 38, 41]. In other words, the same application instructions are executed on the same program values, but Sledgehammer inserts additional tracer execution into replay and the instrumentation needed to support that execution. We also modified Arnold to capture additional timing data during recording to support retro-timing.

4.4.2 Sledgehammer API

Developers debug a replayed execution by specifying tracers that observe the program execution, defining when those tracers should execute, and supplying analysis functions that aggregate tracer output. This is analogous to placing log functions in source code and writing programs to process log output.

4.4.2.1 Tracers

Tracers are functions that execute in the address space of the program being debugged, allowing them to observe the state of the execution. Tracers are compiled into a shared library that is loaded dynamically during query execution. Tracers write output to a logging stream called the *tracelog*; this output is sent to analysis routines for aggregation. Tracers can write to the tracelog directly by calling a Sledgehammer-supplied function or they can specify that all output from a specific set of file descriptors should be sent to the tracelog.

Isolation Tracers must not perturb program state. Even a subtle change to application memory or kernel state can cause the replay to diverge from the recording, leading to replay failure, or even worse, silent errors introduced into the debugging output. None of the queries in our scenarios run without isolation. As Section 4.4.4.1 describes, Sledgehammer isolates tracers in a sandbox during execution; any changes to application state are rolled back on tracer completion. Sledgehammer has two methods of isolation with different tradeoffs between performance and code-generality: fork-based and compiler-based.

With fork-based isolation, tracers run as separate processes. Developers have great flexibility. A tracer can make arbitrary modifications to the program address space, and it can make system calls that write to the tracelog or that affect only child process state. A tracer may link to any application code or libraries and invoke arbitrary functionality within that code, provided it does not make system calls that externalize state. However, we found that fork-based isolation was very slow to use with frequently-executed tracers.

Thus, we added support for compiler-based isolation, in which tracers can execute more limited functionality. This isolation is enabled by compiling tracers with a custom LLVM [45] pass. Tracers can modify any application memory or register. However, they must use a Sledgehammer-provided library to make system calls. This library prevents these calls from perturbing application state. A tracer may call functions in application code or libraries only if that code is linked into the tracer and compiled with LLVM. Since LLVM cannot compile glibc by default, Sledgehammer provides many low-level functions for tracer usage. Our compiler pass verifies that all functions linked into a tracer call only

other functions compiled with the tracer or Sledgehammer library functions. Our results in Section 4.5.5 show that compiler-based isolation executes queries 1–2 orders of magnitude faster than fork-based isolation.

The tracestore Tracers must execute independently. Since tracers run in parallel in different epochs, a tracer cannot rely on state or output produced by any tracer executed earlier in the program execution. Yet, there are often many tracers executed during a single epoch, and sharing data between them can be a useful optimization. For instance, it is wasteful for each tracer to independently determine the file descriptor used for logging by an application.

Sledgehammer provides a *tracestore* for opportunistic sharing of state within an epoch. If data in the tracestore is available, a tracer uses it; if not available, it obtains the data elsewhere. Sledgehammer allocates the tracestore by scanning the replay log to find an address region never allocated by the execution being debugged; it maps the tracestore into this region. This prevents tracestore data from perturbing application execution.

Sledgehammer initializes the tracestore at the beginning of each epoch, prior to executing any application instructions. The developer can supply an initialization routine that inspects application state and sets variables to initial values. If compiler-based isolation is being used, the LLVM compiler pass automatically places all static tracer function variables in the trace store and initializes them at the start of each epoch.

Tracers read and write tracestore values, and updates are propagated to all subsequent tracer executions until the end of the epoch. Tracers may dynamically allocate and deallocate memory in the tracestore; the memory remains allocated until the end of the epoch. All of our scenarios use the tracestore to cache file descriptors, which avoids the overhead of opening and closing files in each tracer. The continuous function evaluation scenarios also cache lists of memory addresses accessed by the function.

Tracer Library Sledgehammer provides several functions that implement common low-level tasks, including:

- `tracerTriggerAddress()`, which returns the instruction pointer that triggered the tracer.
- `tracerStack()`, which returns the stack pointer when the tracer was called.
- `tracerTriggerMemory()`, which returns the memory address that triggered a continuous function evaluation tracer.

4.4.2.2 Tracepoints

Sledgehammer inserts tracers at *tracepoints*, which are user-defined locations in the application being debugged. There are several ways to add tracepoints. First, a location-based tracepoint executes a tracer each time the program execution reaches a given location. Our data corruption scenario uses this method to add tracers to nginx log routines. These tracepoints are specified by adding annotations to the application source code at the desired locations.

Second, a user can *hook* a specific function to invoke a tracer each time a given function is called or whenever a function exits. The tracer receives all arguments passed to the function by default. For example, the memory leak scenario hooks the entry and exit of `malloc` and `free` to track memory usage.

Third, a continuous function evaluation logically inserts a tracepoint to evaluate the function after every program instruction. In practice, Sledgehammer tracks the values read by the function and uses memory page protection to detect when those values change. It only runs the function at these instances. Hooks and continuous functions can be specified by annotations anywhere in the application source code since their effects are global to the entire execution.

When running a query, developers specify which C/C++ source files contain their modified source code. The Sledgehammer parser scans these files and extracts all tracepoint annotations. It correlates each tracer with a line or function name in the application source code as appropriate. Next, it uses the same method as gdb to convert source code lines and function symbols to instruction addresses. For each parameter passed to a tracepoint, the parser determines the location of the symbol, i.e., its memory address or register.

Developers who lack source code or use other programming languages can instead use gdb-like syntax to specify tracepoints, or they can specify all functions residing in a particular binary, or matching some regex. In this case, Sledgehammer leverages UNIX command-line utilities and gdb scripts to associate tracepoints with instruction pointers and symbols.

4.4.3 Preparing for Debugging Queries

Much of the work required to run a parallel debugging tool is query-independent: it can be done once, before running the first query, and reused for future queries. To prepare a recorded execution for debugging, a *master node* parses the replay log and splits the execution into distinct epochs, where the number of epochs is set to the number of cores available. Each core is assigned a distinct epoch. Currently, Sledgehammer requires

each epoch to start and end on a system call. The master divides epochs so that each has approximately the same number of system calls in the replay log.

Next, the master distributes or creates the data needed to replay execution. Arnold replay requires a deterministic replay log, application binaries and libraries, and snapshots of any read-only files [23]. These files are read-only and accessed by many epochs, so the master places them in a distributed file system and sends a message to compute cores informing them of the location.

Each epoch starts at a different point in the program execution. Prior to instrumenting and running the epoch, Sledgehammer must re-create the application state at the beginning of the epoch. A simple approach would replay the application up to the beginning of the epoch. However, for the last epoch, this process takes roughly as long as the original execution of the program. To avoid this performance overhead, Sledgehammer starts each epoch from a unique checkpoint.

During recording, Sledgehammer takes periodic checkpoints every few seconds. This creates a relatively small set of checkpoints that are distributed to compute nodes by storing them in the distributed file system. Prior to running the query, the master asks each compute core to create an epoch-specific checkpoint. Each core starts executing the application from the closest previous recording checkpoint, pauses at the beginning of its epoch, and takes a new checkpoint. This process effectively parallelizes the work of creating hundreds or thousands of epoch-specific checkpoints, and it avoids having to store and transfer many large checkpoints.

Sledgehammer hides the cost of checkpoint creation in two ways. First, it overlaps per-epoch checkpoint creation with parsing of source code. Second, it caches checkpoints on each core so that they can be reused by subsequent queries over the same execution.

4.4.4 Running a Parallel Debugging Tool

To run a query, the master sends a message to each compute core specifying the shared libraries that contain the compiled tracers and analysis functions. It also sends a list of tracepoints, each of which consists of an instruction address in the application being debugged, a tracer function, and arguments to pass to that function.

Upon receiving the query start message, a compute core restores its per-epoch checkpoint and loads the tracer dynamic library into the program address space via `dlopen`. Sledgehammer uses `dlsym` to get pointers to tracer functions. Unfortunately, the dynamic loader modifies program state and causes divergences in replay. Sledgehammer therefore checkpoints regions that will be modified before invoking the loader and restores the checkpointed values after the loader executes.

Prior to starting an epoch, each core also maps the tracestore into the application address space and calls the tracestore initialization routine. Each compute core starts a control process that uses the `ptrace` interface to manage the execution and isolation of tracer code. For each location-based tracepoint or function hook, the control process sets a corresponding software breakpoint at the specified instruction address by rewriting the binary code at that address with the `int 3` instruction.

Each core replays execution from the beginning of its epoch. When a software breakpoint is triggered, replay stops and the control process receives a `ptrace` signal. The control process rewrites the application binary to call the specified tracer with the given arguments. It uses one of the isolation mechanisms described next to ensure that the tracer does not perturb application state. After the tracer executes, the control process rewrites the binary to restore the software breakpoint.

4.4.4.1 Isolation

Tracer execution must be side-effect free: any perturbations to the state of the original execution due to tracer execution can cause the replay to diverge and fail to complete, or such perturbation can lead to incorrect debugging output. Sledgehammer supports fork-based and compiler-based isolation.

Fork-based isolation When a tracepoint is triggered, the control process forks the application process to clone its state. The parent waits until the child finishes executing. The control process rewrites the child's binary to call the tracer. As the tracer executes, it may call arbitrary code in the application and its libraries, but it must be single-threaded. The kernel sandboxes the system calls called by the child process. It allows system calls that are read-only or perturb only state local to the child process (e.g., its address space). To avoid deadlocks, Sledgehammer ignores synchronization operations made by the tracer; this is safe only because the tracer itself is single-threaded. The kernel also redirects output from any file descriptors specified by the developer to the tracelog; this is convenient for capturing unmodified log messages. Tracelog output can also be generated by system calls made by the Sledgehammer library. System calls that modify state external to the process (e.g., writing to sockets or sending signals) are disallowed. System calls that observe process state, e.g., getpid(), return results consistent with the original recording.

At the end of tracer execution, the child process exits and the tracer restarts application execution. If a tracer fails, the control process receives the signal via *ptrace* and resumes application execution.

Compiler-based isolation Our early results showed that fork-based isolation was often too slow for frequently-executed tracers. So, we created compiler-based isolation, which improves performance at the cost of losing some developer flexibility. With compiler-based isolation, tracer libraries must be self-contained; i.e., rather than calling application or library code from a tracer, that code must be copied or compiled into the tracer library. This means that tracers must use a set of standard library functions provided by Sledgehammer instead of calling those functions directly. Tracers must also be single-threaded and written in C/C++.

Sledgehammer compiles tracers with LLVM. A custom compiler pass inserts code into the tracer that instruments all store instructions and dynamically logs the memory locations modified by tracer execution and the original values at those locations to an undo log. The compiler pass inserts code before the tracer returns that restores the original values from the undo log. It also checkpoints register state before executing a tracer and restores that state on return. To avoid deadlocks, the compiler pass omits any synchronization instructions in the tracer; this is safe only because the tracer itself is single-threaded and all its effects are rolled back. The compiler pass verifies that the tracer is self-contained; e.g., that it does not make any system calls.

We noticed that most addresses in tracer undo logs were stack locations. Rather than log all of these stores, Sledgehammer allocates a separate stack for tracer execution and switches the stack pointer at the beginning and end of tracer execution. The compiler pass statically determines instructions that write to the stack via an intra-procedural points-to analysis, and it omits these stores from the undo log. Some variables are passed to the tracer on the stack, so Sledgehammer explicitly copies this data when switching stacks.

If a tracer fails, the control process catches the signal, runs the code to undo memory modifications, restores register state, and continues the application execution.

4.4.4.2 Support for Continuous Function Evaluation

Continuous function evaluation must use compiler-based isolation. When a tracer runs, the compiler pass tracks the set of memory addresses read. The tracer is guaranteed to be deterministic because it cannot call non-deterministic system calls and must be single-threaded. Therefore, the value produced by the tracer cannot change unless one of the values that it has read changes.

Sledgehammer uses memory page protections to detect if any value read by a tracer changes. The control process causes the continuous function to be evaluated at the beginning of the epoch, before any application instruction executes. Tracer execution generates an initial set of addresses to monitor; the compiler adds instrumentation to record this *mon-*

itor set in the tracestore. Sledgehammer executes the tracer only to initialize the monitor set, so tracer output is not logged to the tracelog. The control process asks the kernel to mark all pages containing at least one address in the monitor set as read-only.

When a page fault occurs due to the application writing to one of these pages, the kernel alerts the control process. The control process unprotects the page and single-steps the application. Then, the control process checks if the the faulting address is in the monitor set. If the address is not in the set, the page fault is due to false sharing, so Sledgehammer re-protects the page and continues execution.

If the address is in the monitor set, Sledgehammer runs the tracer again, and records its output in the tracelog. If the tracer faults on a page in the monitor set, the control process unprotects the page and resumes execution of the tracer. Since tracers do not write to many of the pages in the monitor set, unprotecting on demand is much more efficient than unprotecting all pages before tracer execution.

After the tracer completes, Sledgehammer updates the monitor set. If a page is added to the monitor set, Sledgehammer protects it. However, if a page is removed from the monitor set, Sledgehammer does not unprotect the page until the next page fault; this optimization improves performance by deferring work.

The stack switching optimization used for compiler-based isolation is also useful for continuous function evaluation. Reads of addresses on the stack are detected via an intra-procedural points-to analysis and not instrumented. Any remaining stack reads are detected dynamically from their addresses.

4.4.4.3 Support for Retro-Timing

To support retro-timing, we modified Arnold to query the system time when a replay event occurs: such events include all system calls, signals delivered, and synchronization operations, including low-level synchronization in glibc. The timing information is written into the replay log for efficiency. Since Arnold is already paying the cost of interrupting the application and logging its activity, the additional performance cost of querying the system time is minimal (1%, as measured in Section 4.5.6).

A typical replay log will have tens of millions of events even for a few seconds of execution. Logging all this data would introduce substantial slowdown, so we compress the timing data by only logging the time if the difference from the last logged time is greater than 1us.

Sledgehammer provides a library function to query time retroactively. Starting from the application's current location in the replay log, Sledgehammer finds and returns the immediately preceding and succeeding time recorded in the log. Reading the clock at this

point in the execution would have returned a value in this range.

4.4.5 Aggregating Results

Tracelog output can be quite large, so Sledgehammer allows developers to write analysis routines that aggregate the tracelog data. It provides several options for parallelizing analysis to improve performance.

There are three types of analysis routines. A *local analyzer* runs on each compute core and operates only over the tracelog data produced by a single epoch. For example, the data corruption scenario uses a local analyzer to filter undesired messages from verbose logging. If a local analyzer is specified, Sledgehammer creates an analysis process that loads and executes the local analyzer from a dynamic library. Local analyzers receive tracelog data on an input file descriptor and write to an output file descriptor. Sledgehammer uses shared memory to implement high-performance data sharing.

A *stream analyzer* passes information from epoch to epoch along the direction of program execution. The memory leak scenario passes allocated chunks of memory to succeeding epochs so that they can be matched with corresponding frees. This allows each core to reduce the amount of output data it produces.

Each epoch's stream analyzer has an input file descriptor on which it receives the output of the local analyzer (or the tracelog data if no local analyzer is being used). The stream analyzer has an additional file descriptor on which it receives data from its predecessor epoch. It has two output file descriptors: one to which it writes analysis output and another by which it passes data to its successor epoch. Data is passed between epochs via TCP/IP sockets. Each stream analyzer closes the output socket when it is done passing data to its successor, and each learns that no more data will be forthcoming by observing that the input socket has been closed.

A *tree analyzer* combines the output of many epochs. Each compute core sends its output to the node running the tree analyzer via a TCP/IP socket. Sledgehammer receives the data, buffers and reorders the data, then passes the output of the prior stage to the tree analyzer in the order of program execution. The tree analyzer aggregates the data and writes its output to a file descriptor.

By default, Sledgehammer allows a tree analyzer to combine up to 64 input streams. Therefore, if there are less than 64 epochs, a single tree analyzer performs a global aggregation.

Since use of these analyzers is optional, the simplest form of aggregation is NULL tree aggregation, in which Sledgehammer concatenates all tracelog output into a file in order of application execution. Alternatively, a developer may take any existing sequential

Benchmark	Application	Replay time (s)	Tracer calls (millions)
Data corruption	nginx	2.0	7.7
Wild store	MongoDB	30.1	3.4
Atomicity violation	memcached	98.5	42.8
Memory leak	nginx	76.0	3.6
Lock contention	memcached	93.4	75.5
Apache 45605	Apache	50.7	1.9
Apache 25520	Apache	60.1	3.7

Table 4.1: Sledgehammer Benchmarks. This table shows the benchmarks used to evaluate Sledgehammer.

analysis routine and run it as a tree analyzer at the root of the tree. However, Section 4.5.4 reports substantial performance benefits for many queries from using local, stream, and tree aggregation to parallelize analysis.

4.5 Evaluation

Our evaluation answers the following questions:

- How much does Sledgehammer reduce the time to get debugging results?
- What are the challenges for further scaling?
- What is the benefit of parallelizing analysis?
- Does compiler-based isolation reduce overhead?

4.5.1 Experimental Setup

We evaluated Sledgehammer using a CloudLab [71] cluster of 16 r320 machines (8-core Xeon E5-2450 2.1GHz processors, 16GB RAM, and 10Gb NIC). Since several applications we evaluate use at least 2GB of RAM, we only use 4 cores on each machine, yielding 64 total cores for parallelization. To investigate scaling, we emulate more cores by splitting the execution into 64-epoch subtrees, each with their own tree analyzer, and running the subtrees iteratively. We calculate the time for the final tree aggregation by distributing subtree outputs across the cores and measuring the time to send all outputs to a root node and run the global analyzer. We add this time to the maximum subtree execution time. This estimate is pessimistic since no output is sent until the last byte has been generated by the last tree analyzer. We also do not run stream analyzers beyond 64 cores.

Our results assume that the query-independent preparation of Section 4.4.3 (e.g., parallel checkpoint generation) is already completed. Preparation is only done once for each execution and can be done in the background as the developer constructs a query. We mea-

sured this time to be proportional to the recording checkpoint frequency; e.g., preparation takes an average of 2.1 seconds when the record checkpoint interval is every 2 seconds.

4.5.2 Benchmarks

We reproduce the 7 scenarios described in Section 4.3 by injecting the described bug into each application and running the specified Sledgehammer query. In each scenario, our query correctly identifies the bug. All reported results are the mean of 5 trials; we show 95% confidence intervals. Queries use compiler-based isolation and parallelize analysis as described in each scenario. Table 4.1 identifies the time required to replay of each benchmark as well as the number of tracer calls performed in the debugging scenario. We use the following workloads:

- **Data corruption** We send nginx 100,000 static Web requests.
- **Wild store** We send MongoDB workload A from the YCSB benchmarking tool [18].
- **Atomicity violation** We use memtier [60] to send memcached 10,000 requests and execute the final query described in the scenario.
- **Memory leak** We send nginx 2 million static Web requests. By default, nginx leaks memory with this workload, so we did not inject a bug.
- **Lock contention** We use memtier [60] to send memcached 10,000 requests and execute the final query that hooks `pthread functions` and measures timing at 5 tracepoints.
- **Apache 45605** We recreate the bug by stress testing using scripts from a collection of concurrency bugs [90] and run the final query.
- **Apache 25520** We recreate the bug by stress testing Apache and run the final query.

4.5.3 Scalability

Table 4.2 shows the query lancency for 1 core, 64 cores, and 1024 cores. The wild store and atomicity violation scenarios take over 2 hours to return a result with sequential execution. The simplest scenario, Apache 45605, still takes over 4 minutes when executed sequentially. With 64 cores, Sledgehammer speeds up these queries by a factor of 42–55 (with a geometric mean of 48). With 1024 cores, the speedup is 255–601 with a mean of 416. Queries that take hours when executed sequentially return in less than 20 seconds. The data corruption and Apache 45605 queries returns results in one second. At 1024 cores, the result is returned faster than the time to replay the execution sequentially without debugging in all cases.

Benchmark	1 Core	64 Cores		1024 Cores	
	latency (s)	latency (s)	speedup	latency (s)	speedup
Data corruption	324.8 (±2.2)	7.2 (±0.2)	45 (±1.0)	1.0 (±0.0)	330 (±13)
Wild store	7688.4 (±13.5)	181.3 (±6.0)	42 (±2.0)	17.2 (±1.0)	446 (±15)
Atomicity violation	7852.4 (±20.1)	173.1 (±1.2)	45 (±0.3)	13.7 (±0.7)	573 (±15)
Memory leak	1575.2 (±8.1)	30.3 (±0.2)	52 (±0.3)	2.8 (±0.2)	559 (±25)
Lock contention	3281.8 (±17.5)	68.3 (±0.6)	48 (±0.5)	10.9 (±1.2)	301 (±16)
Apache 45605	249.9 (±1.1)	5.2 (±0.5)	48 (±0.5)	1.0 (±0.6)	255 (±15)
Apache 25520	717.3 (±1.6)	12.9 (±0.0)	55 (±0.2)	1.2 (±0.0)	601 (±4.4)

Table 4.2: Sledgehammer Performance. This table shows how Sledgehammer speeds up the time to run a debug query with 64 and 1024 cores, as compared to sequential (1 core) execution. For reference, we also show the time to replay the application without debugging and the number of tracers executed during each query. Figures in parentheses are 95% confidence intervals.

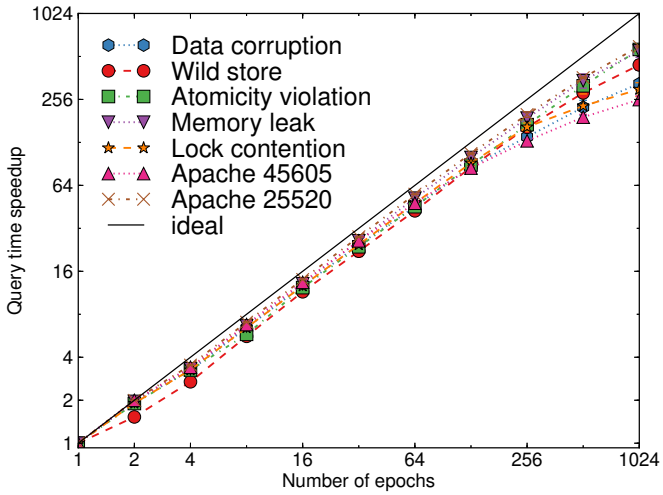


Figure 4.6: Sledgehammer Scalability. This figure shows how query time improves as the number of cores increases.

Figure 4.6 shows how Sledgehammer performance scales as the number of cores increases from 1 to 1024. The diagonal line through the origin shows ideal scaling. Most queries approach ideal scaling, and all continue to scale up to 1024 cores. However, some start to scale less well as the number of cores approaches 1024.

4.5.3.1 Scaling bottlenecks

We next investigated which factors hinder Sledgehammer scaling. One minor factor is disk contention. Arnold stores replay logs on local disk, which leads to contention when 4 large server applications each read their logs during epoch execution on separate cores. We measured this overhead as ranging from 0 to 41% at 4 cores per node, with an average of 15%. This accounts for some of the dip in scalability from 1 to 4 cores.

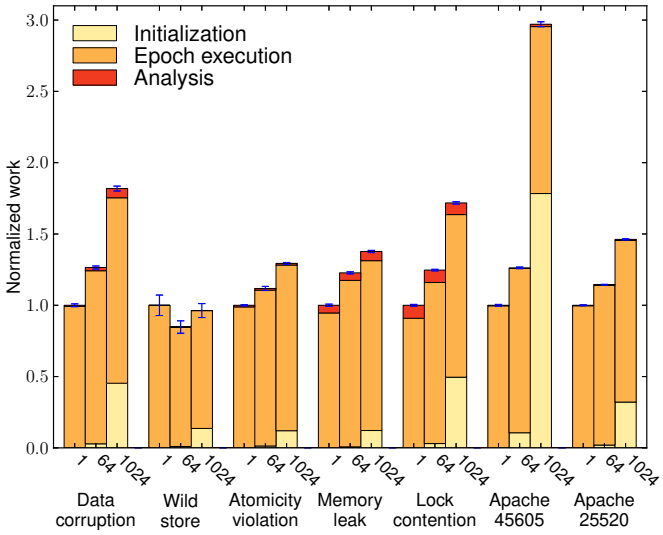


Figure 4.7: Total Work. Each bar sums initialization time, epoch execution time, and analysis time overs all epochs. This shows how much extra work is created by parallelization.

The last step in tree analysis is sequential; its performance does not improve with the number of cores. At 1024 cores, this step is only 0.1–7% of total query time in our benchmarks. While it could be a factor for higher numbers of cores, it has little impact at 1024 cores.

For each query, Figure 4.7 totals individual execution time over all cores when using 64 and 1024 cores, normalized to execution time for single-core execution. Initialization includes restoring checkpoints and mapping tracers into the application address space. Epoch execution is the time to run the application and its tracers, Analysis includes all local, stream, and tree-based analysis. As expected, the cost of per-node initialization increases as the number of cores increases; this is especially noticeable in the Apache 45605, data corruption and lock contention scenarios. Initialization overhead is the primary factor inhibiting the scalability of Apache 45605. Initialization will eventually bound Sledgehammer scalability in other scenarios as well, but it is not the most important factor at 1024 cores.

Interestingly, the total work for the wild store scenario actually decreases slightly as we increase the number of cores. Continuous function evaluation defers work when pages are deleted from the monitor set. For shorter epochs, deleted pages are more likely to never be accessed again; work deferred is never done. At 1024 cores, this effect is dwarfed by increasing per-node initialization work, so total work increases again.

In most scenarios, the most significant barrier to scalability is workload skew. As Sledgehammer partitions epochs into smaller chunks, we see more imbalance in the work

Benchmark	Skew	
	64 cores	1024 cores
Data corruption	1.13 (±0.04)	1.72 (±0.07)
Wild store	1.78 (±0.07)	2.38 (±0.14)
Atomicity violation	1.28 (±0.02)	1.40 (±0.08)
Memory leak	1.06 (±0.01)	1.40 (±0.14)
Lock contention	1.17 (±0.01)	2.16 (±0.24)
Apache 45605	1.07 (±0.03)	1.27 (±0.03)
Apache 25520	1.01 (±0.00)	1.17 (±0.00)

Table 4.3: Skew. The reported values are the longest epoch execution time divided by the average execution time.

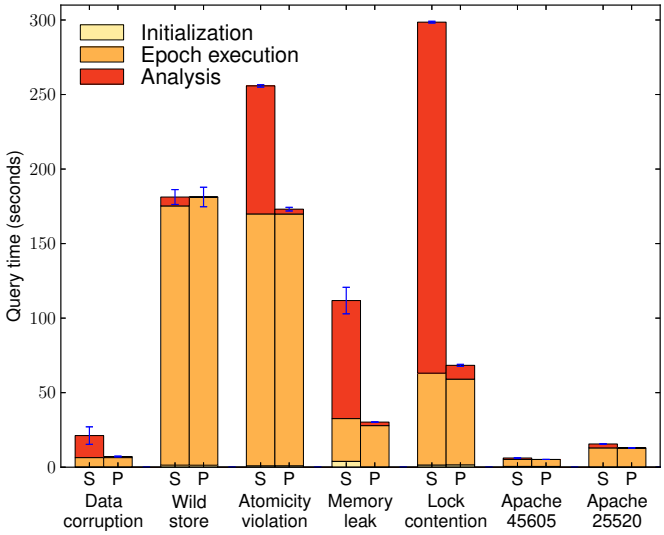


Figure 4.8: Analysis. We compare query time with sequential (S) and parallel (P) analysis using 64 cores. The regions in each bar show how much time is spent in each phase along the critical path of query processing.

done by different epochs. Outlier epochs lead to high tail latency [20]. We quantify skew in Table 4.3 as the ratio of maximum epoch execution time over mean epoch execution time. Perfect partitioning would yield a skew of 1, but Sledgehammer sees average skew of 1.19 at 64 scores and 1.60 at 1024 cores. Skew is the most important factor in the decreased scaling seen in Figure 4.6.

4.5.4 Benefit of Parallel Analysis

We next quantify how much benefit is achieved by parallelizing analysis. Figure 4.8 compares query response time for sequential analysis and parallel analysis using the analyzers for each query described in Section 4.3. We show results with 64 cores, i.e., the largest number of cores we can support without emulation.

Benchmark	Fork-based query time(s)	Compiler-based query time(s)	Speedup
Data corruption	7.5 (±0.9)	.79 (±0.1)	9.6 (±1.4)
Memory leak	32.5 (±0.2)	2.30 (±0.1)	14.2 (±0.5)
Lock contention	632.2 (±5.0)	6.01 (±0.0)	105.3 (±1.0)

Table 4.4: Isolation Performance. We compare the time to execute the first 64 out of 1024 epochs using fork-based and compiler-based isolation.

All scenarios except the wild store and Apache scenarios achieve substantial speedup by parallelizing analysis. The atomicity violation, lock contention, and memory leak analyses traverse large tracelogs and track complex interactions across log messages. Many of these interactions are contained within a single epoch, so local analysis can resolve them. Using parallel analysis speeds up analysis by up to a factor of 96, with a mean improvement of 31. Overall, parallel analysis accelerates total query time by up to a factor of 4, with a mean improvement of 2. Sequential analysis does not scale, so we expect this speedup to increase as the cluster size grows.

4.5.5 Isolation

Table 4.4 compares the performance of compiler-based and fork-based isolation for all queries that do not use continuous function evaluation (which requires compiler-based isolation). On average, compiler-based isolation speeds up epoch execution by a factor of 24, making it the best choice unless its restrictions on what can be included in a tracer become too onerous.

4.5.6 Recording Overhead

We measured recording overhead on a server with an 8-core Xeon E5620 2.4 GHz processor, 6 GB memory, and two 1 TB 7200 RPM hard drives. The average recording overhead for our application benchmarks was 6%. Checkpointing every two seconds increases the average overhead to 8%, and adding additional logging for retro-timing increases average overhead to 9%. The additional space overhead for retro-timing is 17% compared to the base Arnold logging.

4.6 Conclusion

In this chapter, we introduce cluster-fueled debugging: a technique that enables interactivity for powerful debugging tools by leveraging thousands of cores in a compute

cluster. Then, we describe Sledgehammer, the first cluster-fueled debugger. Sledgehammer introduces scalability as a new design constraint for debugging tools in order to achieve cluster-scale parallelization of debugging tools. We leverage massive-scale parallelism to make parallel retro-logging, continuous function evaluation, and retro-timing practical by running them an average of 416 times faster than sequential execution on a 1024-core cluster.

CHAPTER V

The OmniTable Relational Query Model

JetStream and Sledgehammer show how data-centric execution inspection enables cluster-scale parallelization as a mechanism to nearly eliminate the performance issues of existing execution inspection tools. Alas, these approaches amplify the complexity required to perform execution inspection, since developers need to carefully consider parallelism when constructing parallel queries. In this chapter, we identify relational query models as an approach to not only simplify execution inspection, but also to seamlessly leverage performance optimizations, such as cluster-scale parallelization. We focus particularly on debugging use-cases, but the model and artifacts presented in this chapter could be useful across a variety of domains including configuration management and security forensics.

Our new debugging model is the `OmniTable` query model. At the center of the model is a new abstraction, called an `OmniTable`, which is a table representation of an execution that contains all architectural state (e.g., all bytes of memory and all registers) before every instruction executed during an execution. The `OmniTable` query model supports debugging queries written in SQL, which both simplifies debugging logic and enables the underlying query processing engine to automatically optimize and accelerate debugging queries. The model re-purposes existing database concepts (e.g., high-level views) and creates new operators (e.g., traversal functions) to tailor relational models for debugging purposes.

The `OmniTable` query model is not the first declarative debugging model, but prior tools expose incomplete program history that is insufficient for debugging. For example, existing systems only expose data at specific points in time (e.g., function begin/end [27, 31]), only expose some program state (e.g., global variables [55]), or only expose data from inherently incomplete software logs [54, 7, 77]. In contrast, an `OmniTable` provides unrestricted access to all user-level state in an execution.

While our results show that the `OmniTable` query model provides simplicity compared to existing tools (see section 5.5), the model comes with a large feasibility problem: nearly all `OmniTables` are petabytes or exabytes in size, making it impossible to store or compute

a fully materialized table. So, our prototype, Steamdrill, lazily materialized an `OmniTable` on-demand. The system uses deterministic record/replay to store the execution associated with each `OmniTable`, which is much smaller than storing the entire table. When a developer queries an `OmniTable`, Steamdrill generates instrumentation that it executes over the replay to produce the data required for the query. In essence, this approach filters `OmniTable` data *before* it is calculated, rather than afterwards.

Steamdrill uses a query optimization and planning approach to generate instrumentation, which facilitates automated performance optimizations on arbitrary debugging queries. Query planning simplifies the adoption of database optimizations, such as predicate push-downs and caching, to optimize instrumentation. In addition, Steamdrill uses a novel multi-replay approach to resolves debugging queries. The system splits query logic across multiple invocations of a replay so that it can use data that is inexpensive to observe (e.g., data about invoked functions) to reduce the cost of observing data that is expensive to observe (e.g., data about executed instructions).

After query planning, Steamdrill turns to cluster-fueled debugging (see chapter IV) to parallelize execution inspection. Using relational logic simplifies the adoption of cluster-fueled debugging, since queries can be decomposed into numerous stateless operators that are easy to parallelize. Steamdrill partitions the work of resolving a query into portions that it resolves using existing database systems (e.g., spark SQL [1]) and portions that it resolves using instrumentation.

We evaluated the `OmniTable` query model by writing 9 queries to diagnose the root cause of 4 bugs from open-source software projects. We first evaluate the complexity of these queries by comparing them to inspection programs written using `gdb`'s python library, which supports a scriptable interface for familiar `gdb` features (e.g., breakpoints and backtraces). We find that `OmniTable` queries require 3.465 times fewer lines, 1.95 times fewer operands and operators, and would require 3.775 times less time to construct, according to Halstead Complexity [34], than their `gdb` counterparts. Additionally, we find that the average latency of 3 representative `OmniTable` queries is 93.91 times faster than `gdb` scripts.

We make the following contributions:

- The `OmniTable` query model, which facilitates debugging queries over the entire history of execution state by providing the detailed `OmniTable` abstraction and an extended SQL interface.
- Steamdrill, which automatically optimizes `OmniTable` queries using query planning and cluster-scale parallelization.
- An evaluation of 4 queries, which shows that `OmniTable` queries are succinct and that

Metadata				Registers				Memory			
clock	**thread**	**stack**	...	**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 5.1: Omnitable. An example `OmniTable` for a short execution.

Steamdrill can evaluate queries with low-latency.

5.1 Query Model

The `OmniTable` query model helps a developer succinctly reason about the entire history of execution's state. The core abstraction is an `OmniTable`, a table abstraction containing all user-level architectural state of an execution immediately before every instruction in the execution. 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. Developer queries can inspect any state of an execution at any instruction in the execution using the `OmniTable`.

However, an `OmniTable` on its own offers an incomplete debugging tool, since all execution state would need to be accessed in terms of architectural state: variables would need to be accessed directly by their memory addresses, functions by the address of the function entry and return, etc. Moreover, traditional relational logic cannot express the reasoning required for debugging, since it does not support stacks and unbounded traversals.

To solve these limitations, the `OmniTable` query model adopts and extends a variety of database concepts. The model supports `Generators`, user-defined-table-functions, to enable queries over non-architectural state (e.g., debugging symbols). It adds new operators to support common logic used when debugging, such as traversal functions and new `Join` operators for modelings stacks and event ordering.

The `OmniTable` query model makes heavy use of derived views. Derived views label `OmniTable` state according to a high-level abstraction, such as the functions executed in an `OmniTable` or the value of the variables in scope at each instruction in an `OmniTable`. This organization simplifies reasoning across programming abstractions (e.g., reasoning about both executed instructions and function invocations). Derived views are written using the same version of SQL used to query an `OmniTable`; we have created 3views to date.

Below, we describe the relational tables (subsection 5.1.1), SQL operators (subsection 5.1.2), and column operators (subsection 5.1.3) supported by the model. Finally, we

describe how these concepts are used to create a high-level derived view over an `OmniTable` (subsection 5.1.4).

5.1.1 Relations

The `OmniTable` query model supports two types of relational base tables, an `OmniTable` and `Generators`. The model supports columns with primitive types (e.g., Long, String, etc.), structs, maps, and arrays. Finally, the model supports variant columns (see Figure 5.3a), which can contain any data type.

OmniTable The `OmniTable` is a database-style table that logically exposes all state in an execution at all instructions in the execution; Figure 5.1 shows an example of the table for a short execution. For each instruction in the execution, the `OmniTable` contains a `clock`, a monotonically increasing logical-time, the `thread` that executed the row's instruction, and the value of all registers and memory addresses. The `clock` column provides the ordering of the instructions from the original execution (in multi-threaded executions, `clock` reflects a total ordering consistent with the original partial order) and facilitates querying the order of events within an execution. Moreover, the field serves as a primary key across the derived views implemented over an `OmniTable`. To query the `OmniTable` for an execution named `execId`, a developer would specify `ot(execId)` in their query.

Generators An `OmniTable` alone is insufficient for debugging since developers would need to reference all execution state in terms of architectural structures. `Generators` provide a user-defined-trace-function interface for referencing non-execution state in a query, such as debugging symbols, that allows developers to bridge the semantic gap between traditional programming abstractions (e.g., functions, lines of code) and the state of an `OmniTable`. A developer specifies a `Generator` by writing a program that generates relational output to standard out; Steamdrill provides an API to simplify this process. A developer queries `Generator` data by calling it like a function in their query. For example, `FuncDefs` is a generator that produces information about all of the functions defined in a binary; a developer can query all such definitions for an executable, "a.out", by specifying `Select * From FuncDefs("a.out")`. We have written three generators, `FuncDefs`, `InstDefs`, and `VarDefs`; Figure 5.2 describe the schema of the first two.

The input to a `Generator` can depend upon query data. For example, the `Generator Binaries(execId)` identifies the binaries mapped into the address space of the execution named `execId` (this information is contain in an `OmniTable`, but useful for bootstrapping queries). To identify all functions defined in all binaries mapped into the address

InstDefs

addr	`Long`	The address of the instruction definition
operation	`String`	The name of the instruction
operand1	`Variant`	The first operand to the instruction
operand2	`Variant`	The second operand to the instruction
memread	`Stored Procedure`	A procedure that produces the memory address read by the instruction
memwrite	`Stored Procedure`	A procedure that produces the memory address written by the instruction

(a) A `Generator` that identifies all instructions defined in all executables used in an `OmniTable`

FuncDefs

entrEip	`Long`	The address of the instruction definition
exitEip	`Array[Long]`	The address of return instructions in the function
name	`String`	The name of the function
def	`String`	The source location of the function definition
args	`Stored Procedure`	A procedure that produces a map of all inputs and their value
ret	`Stored Procedure`	A procedure that produces the return value of the function when executed at an exitEip

(b) A `Generator` that identifies all instructions defined in all executables used in an `OmniTable`

Figure 5.2: Generators. The schema two `Generators` that we have implemented in the `OmniTable` query model

space of the `execId`, a developer specifies: **`Select`** `*` **`From`** `FuncDefs(`**`Select`** `*` **`From`** `Binaries(execId))`

5.1.2 Relational Operators

To query execution state, a developer specifies SQL-style Select, From, Where queries. Steamdrill supports standard relational operators including join operators, groupby, orderby, and pivot statements. Additionally, Steamdrill provides two types of new join operators to handle debugging semantics which are inconvenient or impossible to express in native SQL.

StackJoin SQL does not support unbounded traversals, so it is not possible to model a function stack using standard SQL. Existing relational debugging tools do not handle this logic and do not allow a developer to reason about recursive invocations of functions. Specifying `r1` **`StackJoin`** `r2` **`on`** `ord1, ord2, equal` in a query joins relations `r1` and `r2` using stack semantics. In particular, the operator groups rows from `r1` and `r2` that are equal based upon the predicate `equal`. It pushes rows in `r1`, ordered by `ord1`, onto a stack until the next row occurs after `r2`, as ordered by `ord2`. The operator then matches the top element on the stack with the current row from the second table.

OrderedJoins It is common to reason about next, or previous, event that matches some criteria. SQL supports this for supports this form of reasoning, but requires complex subqueries and aggregates. So, the `OmniTable` query model adds two new ordered join operators. Specifying `r1` **`NextJoin`** `r2` **`on`** `ord1, ord2, equal` operator matches each row in `r1`, as ordered by `ord1`, to the next row in `r2`, as ordered by `ord2`, that matches the predicate `equal`. **`PrevJoin`** computes the opposite relationship. We find that these relations useful for identifying which function call executed an instruction, or identifying the next access to a shared variable.

5.1.3 Column Operators

The `OmniTable` query model supports a rich set of column types and column operators. The model supports primitives (e.g., long, string, pointer, etc.), structs, maps, arrays, dynamically typed *variant* columns, and postgress-style stored procedure columns. It supports arithmetic and conditional operators, field expressions (`a.b`), subscript expressions (`a[b]`), and traversal functions that iterate through a data structure. The model supports

pointer dereferences by converting them into expressions over an `OmniTableMemory` column (e.g., `a->b` becomes `Memory[a].b`), provided that the variables the deference derive from a single `OmniTable`. The `OmniTable` query model supports grouped aggregations (e.g., **`Count`**, **`Max`**, **`Min`**, etc.), user-defined functions, and user-defined-aggregation functions. Below, we elaborate on traversal function and stored procedures.

Traversal Functions When debugging, developers often traverse the elements in a data structure. However, standard SQL does not support unbounded traversals, so the `OmniTable` query model builds now 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, starting at `column` and following `field` pointers until the value is `NULL`. For example, `traverse(head, "next")` generates a row for all elements in a linked-list starting at `head`. Steamdrill also supports an `explode` operator that converts each value from an array-typed or map-typed column into a row.

Stored Procedures Debuggers perform computation that depends upon execution context. For example, the location of the arguments to a function will depends upon the particular function. Stored procedures [80] are relational logic stored in relation and offer a mechanism to decide the relational logic to perform at query resolution. Stored procedures are called like functions in a query; for example, `Loc(esp)` calls the stored procedure in column `Loc` with the column `esp` as input.

5.1.4 Derived Views

We used the `OmniTable` query model to implement three derived views according to three different abstractions of execution behavior; Figure 5.3) details their schema. The `Vars(ot)` view contains information about all source-level variables that are in scope at every instruction in the `OmniTable` named `ot`. The `eip` column identifies the executed instruction, the `name` and `def` columns identify the variable, and the `clock` column encodes a logical time that makes it possible to compare the execution order of rows in `Vars`. The `Value` column includes the value of the variable; the table uses a polymorphic column type, `Variant`, to allow different rows to have different types of data stored in the `Value` column.

Similarly, `Funcs(ot)` contains information about every function that is executed in the `OmniTable`, `ot` (see Figure 5.3b). The `name` and def columns identify the function that was executed, and the `entr` and `exit` columns identify logical times of function entry and exit, respectively. If the function never returns (e.g., due to a crash), then `exit` will be `NULL`.

Funcs

entr	`Long`	The clock value of the function entry
exit	`Long`	The clock value of the function exit (or null)
name	`String`	The function name
def	`String`	The location of the source-code definition
thread	`Long`	The thread that called the function
args	`Map (String->Variant)`	The argument values of the function call

(a) A derived view identifying all function calls in an `OmniTable`

Vars

clock	`Long`	The clock value of the instruction
thread	`Long`	The thread that executed the instruction
eip	`Long`	The instruction pointer at which the variable is in scope
name	`String`	The variable name
def	`String`	The location of the source-code definition
value	`Variant`	The value of the variable

(b) A derived view identifying the value of all variables in scope at each instruction in an `OmniTable`

Insts

clock	`Long`	The clock value of the instruction
thread	`Long`	The thread that executed the instruction
eip	`Long`	The instruction pointer
read	`Long`	The memory address read by the instruction (or null)
write	`Long`	The memory address written by the instruction (or null)
op	`String`	The name of the type of operation for this instruction

(c) A derived view identifying data about all instructions executed in an `OmniTable`

Figure 5.3: Derived View. The derived views implemented using the `OmniTable` query model.

```
Create View Funcs(ExecId):
Select c.clock as entr, name, def, thread, args, r.clock as exit, rtn
From (Select clock, name, def, thread, args(esp) as args
      From ot(ExecId) Join FuncDefs(ExecId)
      Where eip = entrEip) c
Left StackJoin
     (Select clock, rtn(esp, eax) as rtn, name, thread
      From ot(ExecId) Join FuncDefs(ExecId)
      Where eip in return) r
on clock, clock, c.name=r.name And c.thread=r.thread
```

Figure 5.4: Functs Definition. The SQL code that creates the `Funcs` view.

`Funcs` exposes the arguments to the functions through the `args` column. `args` contains an entry for each argument passed to the function. For example, a developer cal specify `args["i"]` to get the value of the argument `i` passed to the function.

Finally, the `Insts(ot)` view contains information about each instruction contained in the `OmniTable`, `ot` (see Figure 5.3c). The `thread` column identifies the thread that executes the instruction associated with each row, the `eip` column identifies the instruction, the `clock` column identifies the logical time of the operation, and the `op` column identifies the mnemonic for the assembly instruction.

We next describe how we construct the `Funcs` view using the `OmniTable` query model. We use two subqueries to construct the view, one that joins the `OmniTable` to `FuncDefs` when `eip = startIP` to identify all function calls, and one that joins the `OmniTable` to `FuncDefs` when `eip in returns` to identify all function returns. The view uses a `Left StackJoin` operator to join each call with it's corresponding return, or `null` if the function never returns. Figure 5.4 shows the code to construct the view.

5.2 Case Studdies

In this section, I describe four case studies and nine queries that illustrate the use and benefits of the `OmniTable` query model. The queries presented here use the three derived views and two `Generators` presented in section 5.1. Below, I model the behavior of a developer confronted with each bug and describe how they can use the `OmniTable` query model to deduce the root cause of their issue.

```
Select Count(*), args.key, args.value, args.incr
From Funcs(ot)
Where Name = "process_command"
Group By args.key, args.value, args.incr
```

Figure 5.5: Atomicity Q1. The first atomicity violation query

5.2.1 Memcached Atomicity Violation

I describe how the `OmniTable` query model helps a developer debug an atomicity violation in memcached [58], an open-source key-value caching system. In this scenario, a developer notices that an integer from their memcached cache has an incorrect value, i.e., they expect the value of the integer to be 1000 after executing commands against the cache, but the value is only 995. Lost update bugs, such as this one, are difficult to diagnose because the developer does not know what program state to track or when to track it. An atomicity violation detection tool [52] could help identify the cause of the lost update, but, these tools rely on heuristics and can thus misdiagnose the cause of a bug.

Using the `OmniTable` query model, the developer first isolates the module of their program where the bug occurs. Memcached processes requests in two steps, parsing and processing; the developer's first query identifies which of these two steps is buggy. They inspect all calls to a function, `process_command`, that is the boundary between processing and parsing by using the `Funcs` view. The developer groups the calls by three of the function arguments, the key that is updated, the value to use when updating, and whether the value should be incremented and decremented. They count the number of calls in each group. Figure 5.5 shows the query.

The output of the query shows that the failing execution calls the processing function with the correct parameters the expected number of times. So, the developer knows that the bug lies in the `process_command` function. `process_command` operates by updating a local variable, `it`, of type `item`, which represents a key-value pair. Each `item` struct has an associated `linked` boolean field that tracks whether it resides in the cache; a lost update could arise if the function were operating on an unlinked item. To identify if this is the cause of the bug, the developer uses the `Vars` view to track the value of `it` at every instruction where it is in scope (see Figure 5.6). The developer counts the number of times `it` is linked (i.e., `it.linked = True`) and how many times it is unlinked at each instruction in the program. They use a pivot so that each row shows the counts for both linked and unlinked states at a given instruction side-by-side.

The output of the second query identifies a few instructions in the execution in which `it` usually is `linked`, but very rarely is `unlinked`. These anomalous states arise in instructions

```
Select eip, True as "linked", False as "unlinked"
From Vars(ot)
Where name = "it" and def = "memcached.c:123"
Pivot Count() for val.linked in (True, False)
```

Figure 5.6: Atomicity Q2. The second atomicity violation query

after `process_command` assigns `it`, but before the function modifies the the key-value pair. Through code inspection, the developer notices that the function does not correctly lock `it` throughout the function, and so another thread invalidates the item while it is processed. The fix is to add proper locking to the `process_command` function.

5.2.2 Apache Http Performance Anomaly

The next scenario highlights how the `OmniTable` query model allows a developer to simultaneously reason about multiple abstractions over an execution. In this scenario, an apache httpd developer notices that all requests issued to their web server between two requests, the 150^{th} and 160^{th}, experience a large spike in latency [12]. Standard performance debugging tools, such as perf, use sampling to identify frequently executed code. They work well for fixing end-to-end application performance, but are not suited for fixing outlier performance bugs.

The first developer query identifies which source module in httpd is buggy (see Figure 5.7). Internally, apache httpd uses two types of threads, a listener thread, which accepts new requests, and worker threads, which process requests. When worker threads are idle, the system adds them to a worker queue. The developer's query identifies whether workers are idle during the outlier performance to identify which type of thread impedes progress; if all workers are idle, then the listener thread is buggy, otherwise the workers are to blame. The developer identifies the clock values of the start and end of the outlier performance by using a window aggregation function to order all calls to the `request` function, which is called for each request. The developer uses the `Vars` view to identify all instructions where the `worker_queue` is full. They joins these two relations by clock value to determine if the queue is full during the poor performance window

The output shows that the `worker_queue` is full during the poor performance window and suggests that the listener thread causes the poor performance. One possible cause of the poor performance by the listener thread is a blocking system call, so the developer checks if the execution calls any blocking system calls (i.e., system calls that operate on a blocking file descriptor or are called with a long timeout) during the poor performance window (see Figure 5.8). The developer's query uses two subqueries. First, they iden-

```
Select *
From Vars(ot) Join
 (Select row_number() Over (Order By entr) rnum, min(entr), max(exit)
  From Funcs(ot)
  Where name = "request"
  Having rnum>=150 And rnum<=160) on clock>=min And clock<=max
Where name="worker_queue" And val->idlers=val->max_idlers
```

Figure 5.7: Performance Q1. The first Apache Httpd performance anomaly query

```
Select f.name
From Funcs(ot) f Left Join
 (Select Args["fd"], entr
  From Funcs(ot)
  Where Args["arg"]=O_NONBLOCK And Args["cmd"]=F_SETFL And Name="fcnt") b
  on f.entr<b.entr And b.args["fd"]=f.args["fd"]
 Join
 (Select row_number() Over (Order By entr) rnum, min(entr), max(exit)
  From Funcs(ot)
  Where name = "request"
  Having rnum >= 150 And rnum <= 160) on f.entr >= min And f.exit <= max
Where (b.start = NULL And "fd" in f.args) Or (f.args["timeout"] >= 1000)
```

Figure 5.8: Performance Q2. The second Apache Httpd performance anomaly query

tify all calls to `fcntl` that modify a file descriptor to be non-blocking using `Funcs` view. They use a left join to identify all functions that are called on the same file descriptor and occur after the `fcntl`. However, the developer only selects functions that take a file descriptor parameter without a matching non-blocking call (i.e., `b.start = NULL` And `"fd" in f.Arguments`) or those that have a large timeout parameter (i.e., `f.Args["timeout"] > 1000`). Lastly, to limit to search to those functions called during the poor performance window, the developer reuses the window aggregation subquery from their first query.

The query result shows a single call to `poll` that is called with a minute-long timeout when shutting down a socket. The solution implemented by the developers redesigned the system so that shutdown logic is performed by a worker thread and will not stall the system.

5.2.3 Memcached Livelock

In this scenario, a developer notices livelock when testing memcached, i.e., the server stops accepting requests and one thread, `thread=12478`, has high cpu utilization while the other have no cpu utilization [59]. The developer's first query locates the location in the program where the livelock occurs by counting the number of instructions that occur

```
Select count(*), Name
From Insts(ot) i PrevJoin Funcs(ot) f
  on i.order, f.entry, i.order<=f.exit And i.thrd=f.thrd
Where f.exit=NULL And f.thrd=12478 And "memcached" in f.def
```

Figure 5.9: Livelock Q1. The first memcached livelock query

```
Select Name, True as "linked", False as "unlinked"
From Funcs(ot)
Where "item" in args
Pivot Count() for args["item"].linked in (True, False)
```

Figure 5.10: Livelock Q2. The second memcached livelock query

during each of the functions on the call stack of the thread with high cpu utilization (see Figure 5.9). The livelock is most likely in the function that contains an unexpectedly large number of instructions. The developer identifies functions on the call stack at the end of the execution by finding functions in the `Funcs` view that do not return (i.e., `exit = NULL`). They identify all of the instructions that execute during a function by using the `PrevJoin` operator to join each instruction (from `Insts`) to the previous function that operates on the same thread.

The output indicates that a single function has a larger than expected number of instructions, likely from an infinite list. The function contains a single loop that iterates over a linked-list of `item` structs (see subsection 5.2.1), so the developer suspects that the list is corrupted and contains a cycle. The developer identifies any operations over `item` structs that are unexpectedly linked (unlinked) by counting the number of times each `item` parameter is in a linked (unlinked) state when functions that take `item` parameters are called (see Figure 5.10).

The query identifies a single unexpected function call; in exactly one case, `item_free`, the call to free an item, is called on a linked `item`. However, memcached uses reference counting and `item_free` should only be called when the refcount field reaches 0. A final query tracks all updates to the reference count of the erroneous item and determines if it there is an underflow or overflow (see Figure 5.11). The output identifies a single overflow; the developer fixes the issue based upon the call stack of the erroneous addition.

5.2.4 SQLite Semantic Bug

I describe how a developer can use the `OmniTable` query model to debug an occasional segmentation fault in their webserver [78]. The webserver processes each request by querying an internal SQLite in-memory database; the segmentation fault indicates that there is a

```
Select Name, args["recount"] as ref
From Funcs(ot)
Where Name = "refcnt_incr" And ref = 65535
   Or Name = "refcnt_decr" And ref = 0
```

Figure 5.11: Livelock Q3. The third memcached livelock query

```
with (Select explode((Oper[args["nOp"]])args["p"]), entr, exit)
From Funcs(ot)
Where Name="sqlite3VdbeExec") as Opers
Select o1.Oper.opcode, True, False
From Opers o1 Join Opers o2 on entr And o1.val.cursor = o2.val.cursor
Where o2.opcode = "Rewind" And o1.index < o2.index
Pivot Count() for o1.exit=NULL in (True, False)
```

Figure 5.12: Semantic Q1. The first query for the SQLite bug

bug in the SQLite logic. In particular, SQLite convert each query into a list of operations that act upon `cursors`; the fault shows that one type of operation, a `Rewind` operation, occasionally acts upon a null `cursor`.

The developer's first query identifies whether the bug is located in the logic that creates a list of operations or the logic that executes one by comparing the list for the failing query with the list for successful queries (see Figure 5.12). They create a view, called `Opers`, that identifies each operation that is passed to `"sqlite3VdbeExec"`, the function that executes each list of operation. In particular, they cast a pointer argument to the function to an array of `Oper` structs with size `nOp` (another argument to the function) and use the `explode` function to produce a row in the output for each element in the array. Additionally, the view identifies whether the operation belongs to the failing query by checking to see if the call to `"sqlite3VdbeExec"` completed. The developer joins `Opers` with itself to count how many times each operation occurs before a `Rewind` to the same cursor in both failing and successful queries. They use a pivot so that each row shows the successful and failing counts side-by-side.

The query identifies that an `Open` operation precedes every `Rewind` except for in the failing case. The developer next aims to understand why an `Open` operation was not added to the list of operations in the failing case (see Figure 5.13). So, the developer identifies "near-misses", where the program almost created an `Open` operation. In particular, they identify instructions that were executed from functions which *could* have added an `Open` operation to the list. They first find all functions that contain a call to `OpenCreate`, the function that creates `Open` operations, using the `FuncDefs` and `InstDefs Generators`. They then join two `Funcs` views and one `Insts` view to the above generators to identify

```
Select Unique eip
From Funcs(ot) c
Join Funcs(ot) e on c.args["p"]=e.args["p"]
Join Insts(ot) i on i.clock>=c.entr And i.clock<=c.exit
Join FuncDefs(ot) f on f.entrEip<=i.eip And f.exitEip<=i.eip
Join InstDefs(ot) id on f.entrEip<=id.addr And f.exitEip<=id.addr
Where c.name="create_list" And c.name="Sqlite3VdbeExec" And p.exit=Null
  And opcode="call" And operand="OpenCreate"
```

Figure 5.13: Semantic Q2. The first query for the SQLite bug

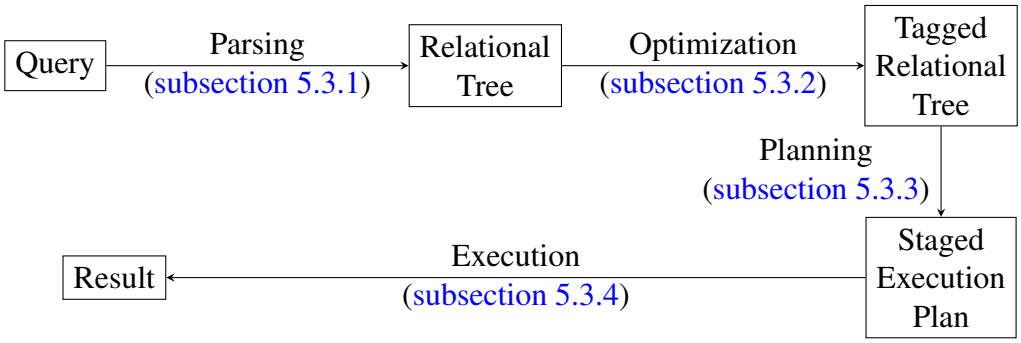


Figure 5.14: SteamDrill Design. Depicts each step in resolving a query together with their input and output data-structures.

all instructions that execute during the function that creates the list of operations which ultimately fail.

The output of the query identifies that all of the ”near-misses” originate from a single function. Moreover, by mapping the instructions back to the source code, the developer identifies a boolean flag that, if reversed, would create an `Open` operation. The bug fix involved setting the boolean to `False` when processing certain types of queries.

5.3 Design

In this section, we describe how Steamdrill resolves debugging queries. From the developer perspective, Steamdrill executes queries to filter and select data from a fully materialized `OmniTable`. However, fully realizing an `OmniTable` is infeasible due to high storage and compute costs: the size of an `OmniTable` is roughly equal to the size of addressable memory times the number of executed instructions and thus reaches petabytes of data after even a few seconds of execution.

Instead, Steamdrill lazily materializes `OmniTable` data, eliding realization of any `OmniTable` state that is unneeded to answer a developer query. Rather than extract an `OmniTable` during execution, Steamdrill uses deterministic record and replay [23] to extract a log of

```
Select eip, read, write
From Insts(e)
Left Join
 (Select name, m.entr, f.exit, m.rtn
  From Funcs(e) m NextJoin Funcs(e) f
  on exit, entr, m.rtn = f.args["ptr"] And f.name = "free"
  Where m.name = "malloc")
on entr<clock And clock<exit And (read=rtn Or write = rtn)
Where name=NULL
```

Figure 5.15: Undefined Use Query. An undefined use query over the execution identified by e.

non-deterministic inputs to the program execution. For each developer query, the system uses the logs of nondeterministic inputs to generate `OmniTable` state on-demand by generating instrumentation for the specific query and re-executing the original program with the instrumentation to extract execution data. In essence, the approach filters `OmniTable` data *before* the table is generated instead of afterwards.

Steamdrill's design mirrors that of conventional database engines [2], which allows the system to seamlessly employ known debugging optimizations, such as cluster-scale debugging chapter V, and employ novel ones. In particular, since deterministic replay guarantees consistent program state across executions of the same replay, Steamdrill treats each `OmniTable` referenced in a query as an independently materializable object. The system assigns materialization tasks across multiple replay executions, which enables it to use inexpensive-to-materialize relations (e.g., `Funcs`) to reduce the calculation of expensive-to-materialize relations (e.g., `Insts`)

In the rest of this section, we describe how Steamdrill resolves queries in more detail (see Figure 5.14). First, Steamdrill translates a query into a tree of relational operators (internal nodes) over data tables (leaves) (subsection 5.3.1). These trees help the system more easily employ relational optimizations (subsection 5.3.2) and decomposes logic into easy to parallelize operators. The system then constructs a staged execution plan, which assigns each operator in the relational tree to a particular replay stage (subsection 5.3.3). Finally, the system executes each stage by instrumenting a replay execution and leveraging existing database engines (subsection 5.3.4), constructing the `clock`column using a performance counter optimization (subsection 5.3.5).

We use the query in Figure 5.15 as an example. The query is a simplified query for identifying undefined use bugs in an execution. It identifies all memory instructions that read or write to memory that was not dynamically allocated. It uses a subquery to identify the valid region of time for dynamically allocated memory by using a **`NextJoin`** to find

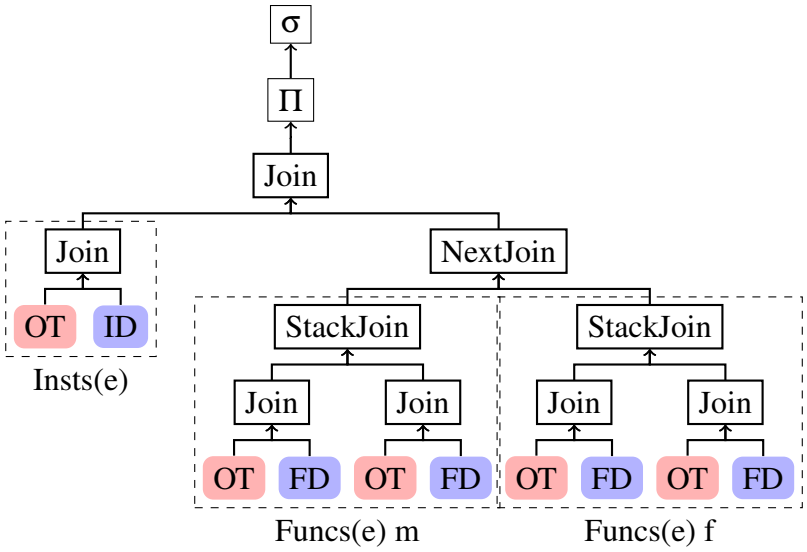


Figure 5.16: Undefined Use Parsed Tree. The tree of relational operators after parsing. Each `Generator` is a blue oval, each `OmniTable` is a red oval, and each relational operator is shown as a white rectangle. Dotted rectangles identify the derived views used in the query.

match each call to `malloc` with the subsequent call to `free`. The query uses a **`Left Join`** to join each instruction that dereferences memory to the dynamically allocated region of time, but only produces output when no match is found.

5.3.1 Parsing

Steamdrill begins processing a query by parsing the query and 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. Steamdrill follows the standard database approach: selection statements are converted into 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 [73]. Tables in the relational tree are decoupled and may appear multiple times. Figure 5.16 shows the tree of relational operators for the undefined used query.

5.3.2 Optimization

After converting the query into a tree of relational operators, Steamdrill optimizes the query by manipulating the tree of relational operators. The system uses the rule-based pattern-matching approach of the catalyst optimizer [1]. Steamdrill allows developers to specify rules as simple patterns that manipulate the tree of relational operators. These rules encode traditional database optimizations (e.g., operator push-down and caching), debug-

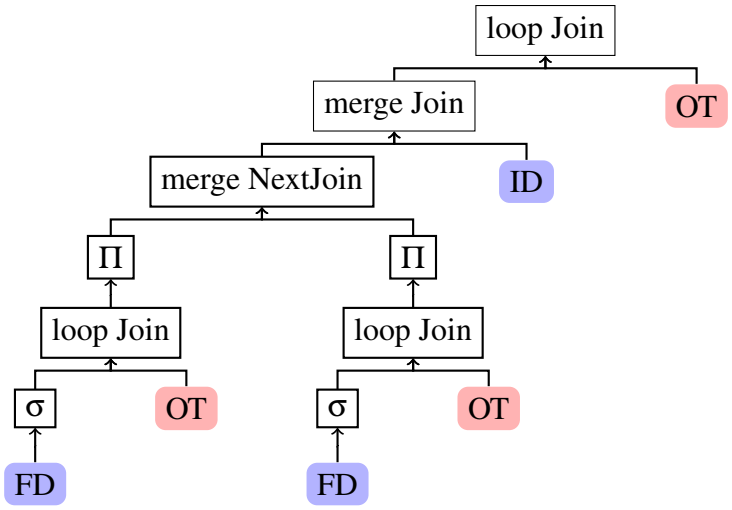


Figure 5.17: Undefined Use Annotated Tree. The tree of relational operators after optimization. Steamdrill rules choose a join order that evaluates `Funcs` subqueries before `Insts`subtrees.

ging specific optimizations, (e.g., converting logic over instructions to logic over basic blocks), and decide upon the join order and join algorithm for each join in a query. The output of optimization is a relational tree in which all join nodes have been annotated with their join algorithm. During planning, Steamdrill will use the join order and algorithms to separate the tree into stages.

The most critical task in optimization is determining the join order and algorithm. Steamdrill supports two execution strategies for all `Join` operators other than `StackJoin`: merge-based joins, which operate over fully realized relations, and block nested loop joins, which calculate one of the input relations first and use the data in that relation to limit the data scanned over the second relation. We describe the specific rules that we have implemented in section 5.4.

Figure 5.17 shows the output of optimization for the undefined use query. Steamdrill first removes the subtrees for the call to `malloc` and the return from `free` since no data is required from either subtree. Steamdrill rules then reorder join operators to minimize the work of materializing the `Insts`subtree by computing the `Funcs` subtrees first. The system determines the type of join (merge vs. loop) to use for each join operator.

5.3.3 Planning

Next, Steamdrill converts the tagged relational tree into an execution plan. The execution plan groups each relational operator into a *stage*, where each `OmniTable` node in a stage will be executed in the same replay execution. The system uses as few stages as possible (since each additional stage will require a replay execution) while correctly following

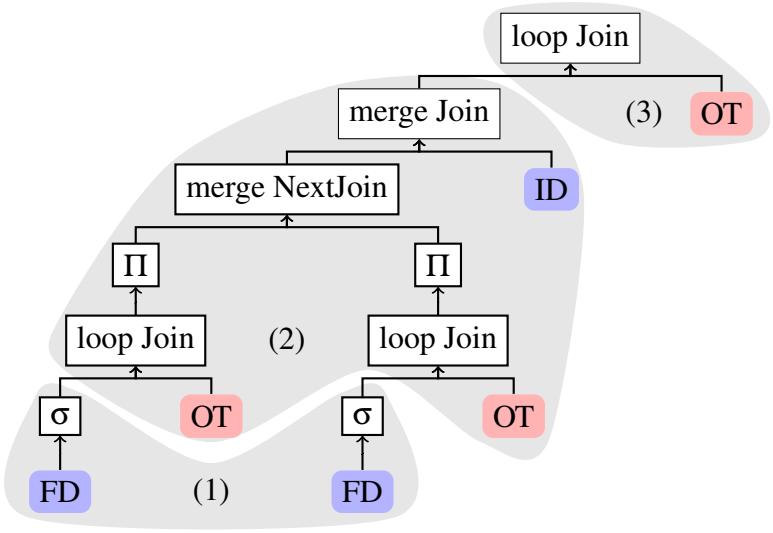


Figure 5.18: Undefined Use Execution Plan. The tree of relational operators after planning. Each stage is shown in a gray background and labeled (1), (2), or (3).

the join algorithms and join order in the tagged relational operator tree. This approach is similar to database engines that attempt to reuse table scans[92].

The system performs a depth-first traversal of the tree starting at the root node and tracks the current stage assignment, starting at 1. Leaf nodes (`OmniTable`, `Generator`) are assigned to the current stage value. Unary nodes (i.e., all non-join operators) inherit the stage of their child. Merge join operators inherit the maximum stage assignment from their child node. To assign loop join operators to a stage, Steamdrill first traverses the inexpensive relation (by convention, the left), adds one to the stage of the root of the inexpensive tree and uses the incremented value to traverse the expensive relation. The system assigns the loop join operator to the stage of the root of the expensive relation.

Figure 5.18 shows the staged execution plan for the undefined use query. Each of the gray regions represents to a separate execution stage; by convention the relation shown on the left-hand side of each loop join is calculated before the relation on the right-hand side. When manipulating tree operations, Steamdrill pushes the selection and projection operators from the root of the tree towards the leaf nodes. This process allows Steamdrill to prune subtrees that do not produce data in the query output (e.g., execution data from the entry to `malloc` and the return from `free` functions are not used by the query).

5.3.4 Execution

Next, Steamdrill executes each stage in staged execution plan to resolve the query. To resolve a stage, Steamdrill first identifies the operators that it can use to create instrumen-

tation for the next replay execution. For each `OmniTable` node in the stage, Steamdrill gathers the subtree of operators that consume row-by-row data from that `OmniTable` node. For example, in the second stage of Figure 5.18, the Merge Join and Merge NextJoin nodes are multi-row relational operators, the ID node is a row-by-row `Generator` subtree, and the two subtrees root at Π nodes are row-by-row `OmniTable` subtrees. To produce `Generator` data, the system executes the `Generator`'s program with the inputs from the query. Steamdrill uses existing implementations and algorithms from the database community to implement other relational operators in the query (e.g., merge joins, aggregations) [1].

To materialize data for each row-by-row `OmniTable` subtree, the system instruments the replay execution. The system converts each row-by-row subtree into a stateless *cursor*. Logically, each cursor single-steps through the execution, conditionally producing the execution data from an output clause when a filter clause evaluates to true. Since each subtree consumes data from a single row, cursors are embarrassingly parallel (i.e., the system can execute the same cursor over multiple time regions of the program simultaneous using cluster-fueled debugging [70]). For each subtree, the system converts the clauses in projection nodes into the output clause and converts the clauses in all join and selection nodes to into the filter clause.

An early prototype collected all of the cursors in a stage and passed them as `tracers` to execute in a parallel Sledgehammer query. Unfortunately, this approach is far too expensive, even with cluster-fueled debugging, since single-stepping the execution can impose slowdowns of 4 orders of magnitude. So, Steamdrill analyzes the filter clause of a cursor to determine when the cursor might produce output without having to execute the filter clause after every instruction.

The system analyzes the filter clauses and the executables in the recorded execution to identify program locations at which the filter might evaluate to true; it calls the set of program locations the *breakpoint set*. Second, the system identifies any regions of memory upon which the filter clause depends and places these into *watchpoint set*. Third, Steamdrill identifies time-based dependencies (i.e., conditions on `clock`) in the filter clause which it places in *progresspoint set*. Steamdrill creates two versions of the replay executables, the native executables and one that includes cursor code at their corresponding breakpoint sets. The system switches back and forth between the executables depending upon the value of `clock`(for progresspoints) and memory (for watchpoints). It uses Sledgehammer continuous function evaluation to track watchpoints and uses a kernel module to alert the system of progresspoints.

While these optimizations limit the number of instructions that require cursor logic, the frequency of instrumentation poses a problem for existing instrumentation tools. Dy-

namic instrumentation tools, such as PIN [53], guarantee accurate instrumentation, but do not scale as Steamdrill parallelizes queries (see chapter III) because they duplicate instrumentation effort across each epoch. Software breakpoint tools, like Sledgehammer, scale to large numbers of cores, but impose massive slowdowns for frequently executed code because each breakpoint imposes two context switches. Static binary instrumentation would allow shared instrumentation and would scale, but existing systems rely on inaccurate heuristics [89] because x86 ELF binaries cannot be statically disassembled.

Steamdrill uses an approach similar to that taken by MULTIVERSE [8] in which it soundly disassembles the executables by over-approximating the basic-blocks in the program. However, where as xxx treats all bytes in a binary as a potential starting point for a basic-block, Steamdrill leverages Intel Control Flow Enforcement Technology (CET) [75] to drastically improve its precision. CET requires that binaries include special `ENDBR` instructions at all indirect jump and indirect call destinations. Steamdrill scans all regions of ELF executables in the replay that were mapped with executable permission to create a work-list of basic block starting addresses. For each address in the work-list, the system disassembles the binary starting at the address and adds the destination of any direct jump or direct call instructions to the work-list. After disassembling all elements in the work-list, the system treats all discovered targets as the starting point of a basic block and uses the next aligned basic block start address as the ending point for the block.

Then, Steamdrill instruments the executables in a replay basic-block by basic-block. As long as the distance between two subsequent basic-blocks is longer than a `jump` instruction, the system uses a `jump` instruction instead of a software breakpoint. When two basic-blocks are too close together for a `jump` instruction, the system uses a software breakpoint approach that context switches to a daemon process to execute instrumented code.

5.3.5 Clock Optimization

The `clock`column is a critical primitive in the `OmniTable` query model since it both provides ordering and acts as a primary key. Alas, actually calculating an instruction count would be too expensive, since each instruction in the program would need to increment a counter. However, instructions will execute from low program counter to high program counter within a basic block. So counting the number of `jump`, `call`, and `return` instructions executed is sufficient, with ties broken by the program counter value.

Unfortunately, tracking all jump instructions is expensive because of both the additional work for each query and the additional work of instrumenting all `jump` instructions. Steamdrill turns to program counters to solve this issue. In particular, Intel provides two deterministic counters, the number of call instructions executed and the number of condi-

tional jump instructions executed [86]. With the instruction pointer tie-break, Steamdrill thus only needs to instrument unconditional backwards `jump` instructions and all `return` instructions.

5.4 Implementation

We built a Steamdrill prototype by extending spark SQL [1] and Sledgehammer. Relational operations (joins, aggregations, etc.) execute within Spark, while the instrumentation and execution code execute in Sledgehammer. We used Spark's DataSource API to expose the `OmniTable` as a relational interface and add new operators for interfacing with an `OmniTable`. Significant implementation work is involved with adding stored procedures and nested-block loop-joins to spark, because they require a restructuring of an `OmniTable` scan.

We implement rules for the catalyst optimizer. In general, the rules aim for left-deep join structures in which `OmniTable` nodes are isolated on the right-hand side of a join node. Intuitively, these structures allow the system to leverage as much filtering as possible when extracting data from an `OmniTable`. The rules favor ordering joins so that the system calculates `Funcs` before `Vars`and `Vars`before `Insts`. Each rule required roughly 25 lines of Scala code, making it an extendable approach as developers add derived views. We continue to investigate cost-based approaches that could help infer or simplify rule construction.

Our instrumentation component takes cursors as input and produces instrumented basic-blocks, watchpoints and progresspoints. Our implementation currently only identifies breakpoints by inspecting bounds on the instruction pointer in the filter clause of a cursor (i.e., it does not statically analyze the replay executables). The instrumentation produces an c file, which includes the output and filter clauses of each `cursor` object, and an inline assembly component, which includes the original logic of the program instrumented to call into the `cursor` c code. The tool compiles these components before injecting them into a replay execution.

Finally, we built a library for constructing `Generators` that formats output to `stdout` in a format that the Steamdrill spark component can parse.

5.5 Evaluation

In this section, we evaluate the `OmniTable` query model and Steamdrill. We answer three questions: first, "Does the `OmniTable` query model improve upon existing debugging

interfaces?", second, "Does Steamdrill accelerate debugging queries?", and third, "How do Steamdrill design decisions impact query performance?".

We use the four case studies and nine queries presented in section 5.2 for evaluation criteria. As a baseline for our study, we implement the queries from section 5.2 using `gdb`'s python bindings. gdb's python bindings enable scripting on top of gdb abstractions (e.g., breakpoints and backtraces). We present an example python script in subsection 5.5.1 and perform a qualitative and quantitative comparison of `gdb` scripts with equivalent `OmniTable` queries.

We then implement and execute 3 queries from our case studies using the `OmniTable` and `gdb`. We present a comparison between the performance of the `OmniTable` and `gdb` in subsection 5.5.2. We investigate the scalability of Steamdrill (subsection 5.5.3), and show the performance impact of using rounds of replay and the performance counter optimization (subsection 5.5.4).

We gather performance numbers using a CloudLab [71] cluster of 8 r320 machines (8-core Xeon E5-2450 2.1GHz processors, 16GB RAM, and 10Gbps NIC).

5.5.1 OmniTable and GDB Comparison

In this section, we compare the complexity of queries written in the `OmniTable` query model with scripts written for `gdb`'s python bindings. We find that all of the debugging queries for the case studies are expressible in both the `OmniTable` query model and in `gdb`'s python bindings. We first explain the complexity qualitatively by providing a representative `gdb` query. Then, we use software engineering metrics to compare the complexity of these queries qualitatively.

5.5.1.1 Qualitative Comparison

Figure 5.9 shows the `gdb` script for the first memcached livelock query. The script creates a custom class, `operation`, that extends the `gdb Breakpoint` class. Each time the execution reaches a breakpoint, `gdb` executes the `stop` function from the associated class. The script manipulates a multi-dimensional data structure, `counts`, which stores a stack for each function executed in the program. For all functions, the script creates a breakpoint that adds a new counter for the function invocation. For all instructions in a function, the script increments the counter at the top of the stack. For all return instructions in a function, the script pops the counter at the top of the stack. The script uses regular expressions to identify each type of breakpoint. Finally, it prints out all of the counts on all of the stacks for each function in `counts`.

```
counts = defaultdict(list)
class operation(Breakpoint):
  def __init__(self, location, l, o):
    Breakpoint.__init__(self, location)
    self.label = l
    self.oper = o
  def stop(self):
    if get_record_pid() == 12478:
      self.oper(self.label)
    return False
def add_one(l):
  counts[l][-1] += 1
def pop(l):
  counts[l].pop()
def push(l):
  counts[l].append(0)

for line in execute("info functions").split("\n"):
  match = search(r"\A[^\(\)]+ ([^ \t\n\r\f\v\(\)]+)\(.*\).*", line)
  if match:
    func = match.group(1).strip("*")
    operation(func, func, push)
    for inst in execute("disas %s".format(func)).split("\n"):
      i = search(r"\s+(0x[0-9A-Fa-f]+).*", inst)
      r = search(r"\s+(0x[0-9A-Fa-f]+).*ret.*", inst)
      if i:
        operation(c.group(1), func, add_one)
      if r:
        operation(r.group(1), func, pop)

def print_counts():
  for func in counts:
    for count in counts[func]:
      print("%s:%d".format(func, counts[func][k]))
events.exited.connect(print_counts)
```

Figure 5.19: The gdb script for the first memcached livelock query

Metric	Lines		Nodes		Halstead	
	gdb	OT	gdb	OT	gdb	OT
Atomicity Q1	7	4	48	39	147	82
Atomicity Q2	11	4	74	26	518	38
Performance Q1	20	6	94	52	518	227
Performance Q2	30	9	113	120	989	1243
Livelock Q1	35	3	149	26	1471	62
Livelock Q2	12	4	69	23	397	34
Livelock Q3	10	3	45	26	140	39
Semantic Q1	22	10	110	77	911	520
Semantic Q2	41	8	151	96	1489	684
Average	20.89	5.67	94.78	53.89	731.66	326.16

Table 5.1: OmniTable Complexity. Complexity metrics for `OmniTable` (OT) queries and `gdb` scripts for each query in our case study.

In sum, we see complexity in this query in its use of (1) a multidimensional data-structure, (2) complex control flow operations, and (3) complex regular expressions.

5.5.1.2 Quantitative Comparison

We use three software engineering metrics to compare complexity of `gdb` scripts and `OmniTable` queries: the number of lines of code, the number of terms in the abstract syntax tree associated with the query or script, and the Halstead Complexity, which uses properties of an abstract syntax tree to estimate the amount of time it will take to correctly produce the program [34]. Table 5.1 shows the evaluation metrics for each query in our case studies.

We see that `OmniTable` queries are usually substantially less complex than their `gdb` counterparts. By geometric mean, `OmniTable` queries require 3.465 times fewer lines, 1.95 fewer nodes in the abstract syntax tree, and require 3.775 less time to construct according to Halstead Complexity. The only query that is more complex than its `gdb` counterpart is the second query for the performance anomaly use case. Additional complexity arises in the second performance query because the `OmniTable` does not include kernel state, so identifying the blocking file descriptors requires substantial logic. In contrast, `gdb` scripts directly execute `fcntl` to identify whether a file descriptor is blocking or non-blocking. Extending the `OmniTable` query model to include kernel state would eliminate this complexity.

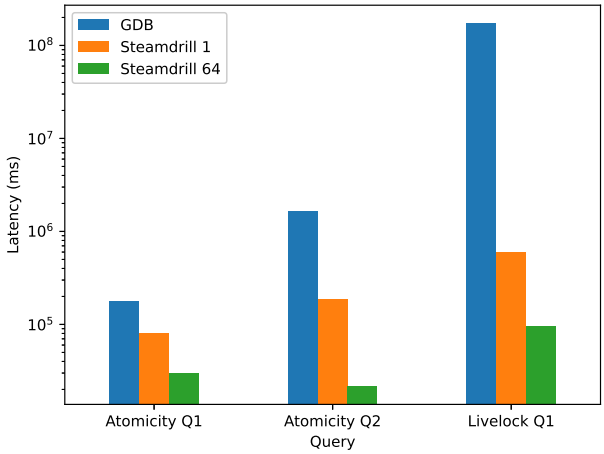


Figure 5.20: SteamDrill and GDB Performance Comparison. Compares the latency of `gdb` scripts and Steamdrill queries. The y-axis is in log scale

5.5.2 SteamDrill and GDB Comparison

We compare the latency of 3 queries using Steamdrill with that of the corresponding scripts executed using `gdb`. Figure 5.20 shows the latency of each tested query when using `gdb`, Steamdrill with a single core, and Steamdrill with 64 cores. We plot latency on a log scale.

In order to provide a reasonable performance comparison, we manually perform the multi-replay optimization for the Livelock Q1 `gdb` script. The first script calculates the functions on the stack at the end of the execution and second tracks instruction counts in only those functions. The first script finished in about 2 hours, while the second was less than halfway completed after a full 24 hours. We kill the script at this time and use 48 hours as an estimate of the total run-time.

We find that Steamdrill accelerates queries relative to `gdb` by a geometric mean of 17.84 when using one core and a geometric mean of 93.91 when using 64 cores.

5.5.3 Steamdrill Scalability

We next evaluate the scalability of Steamdrill. We measure the latency of Steamdrill queries when using between 1–64 cores. Figure 5.21 provides a log-log plot with number of cores on the x axis and speedup on the y-axis. Based on geometric mean, Steamdrill queries at 64 cores are a 9.83 times faster than queries when running on a single core.

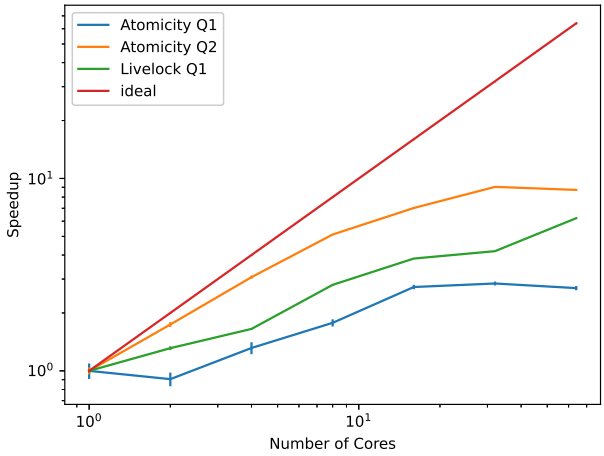


Figure 5.21: SteamDrill Scalability. Compares the latency of `gdb` scripts and Steamdrill queries.

5.5.4 Optimization Impact

We evaluate the impact of two optimizations on Steamdrill performance. First, we evaluate the impact of using rounds of replay on query latency. We first evaluate Livelock Q1 using Steamdrill's default optimizer, which uses two rounds of replay to calculate the query, and using an optimizer that schedules all materialization tasks in a single replay. We execute both queries using 64 cores. We find that Steamdrill is 3.48 times faster when using multiple rounds of replay instead of a single round.

Next, we evaluate the impact of the performance counter optimization. We execute the 3 evaluation queries using the performance counter optimization and then with the optimization disabled (when disabled, Steamdrill instruments all `jump`, `call`, and `return` instructions). We execute all queries using 64 cores. We find that queries using the performance counter optimization are 1.44 times faster than those using the naive instrumentation.

5.6 Conclusion

In conclusion, we presented the `OmniTable` query model as a solution for writing more succinct execution inspection queries, and Steamdrill as a system that realizes the model. We showed that `OmniTable` queries are more succinct that existing debugging tools, and that Steamdrill resolves them faster as well.

CHAPTER VI

Conclusion and Future Work

In this dissertation, we proposed data-centric execution inspection as an alternative framework for thinking about execution inspection. We argued that treating an execution as a first-class data object allows us to consider execution inspection from a query-based perspective, instead of from the traditional low-level programming perspective. We showed that this framework enables the use of data-centric approaches, namely cluster-scale parallelization and relational query models, to build fundamentally more powerful inspection tools.

More precisely, we presented two tools, JetStream and SledgeHammer, which use cluster-scale parallelization to accelerate our debugging queries. JetStream and SledgeHammer provide evidence of how thousands of cores in a compute cluster can accelerate the latency of inspection tasks from hours or minutes to seconds. Then, we presented the OmniTable query model, which turns to relational query models to not only simplify our debugging tools, but also to seamlessly adopt performance optimizations, such as cluster-scale parallelization.

In the rest of this section, we propose future work that extends the ideas presented in this dissertation. We first describe follow up work that expands the `OmniTable` query model beyond single-execution inspection into multi-execution inspection. Then, we identify the challenges and opportunities that surround adoption of data-centric execution inspection. Finally, we propose a future where data-oriented approaches percolate into other aspects and areas of software reliability.

6.1 The Future of OmniTable Queries

In chapter V, we described how a developer can use the `OmniTable` relational query model to inspect a single execution of their software. In essence, the use-cases presented

in the chapter present the `OmniTable` query model as a "narrow-waist" for performing program inspection, whether for sophisticated dynamic analyses or one-off debugging questions. My evaluation (see section 5.5) shows that the model is superior to state-of-the-art tools; on average, the `OmniTable` decreases the complexity of inspection by a factor of 3.775. Moreover, the design of the `OmniTable` query model allows Steamdrill to use the same query resolution strategy and optimizations across an extremely diverse set of inspection tasks; a feature absent from existing state-of-the-art inspection techniques. Within the context of debugging single-executions, we see follow-up work investigating extensions to the model to enable transitive queries for inspection such as information flow.

Additionally, while these results show the benefits of the model, the use-cases and scenarios presented in chapter V only scratch the surface of the tasks that are possible with an `OmniTable`. SQL queries can use `Join` operators to extract data from multiple tables; the use-cases in chapter V join diverse views of the *same* `OmniTable` when debugging an execution. However, it is easy to imagine using the `OmniTable` query model to join inspection *across* diverse executions. There are at least three directions facilitated by cross-execution inspection.

First, a developer can inspect multiple executions of the same (or nearly the same) program. They can perform statistical bug isolation [50], in which they inspect failing and succeeding executions to gather predicates over executions that are highly correlated with failure. Moreover, whereas prior tools were application independent and difficult to customize, the `OmniTable` query models simplifies the customization of these tools. Additionally, this model allows a developer to implement custom performance debugging queries. For example, the `OmniTable` model would allow a developer to identify if the performance of their application is correlated with the size of a custom data-structure in their program. Finally, developers can perform "evolutionary debugging", in which they debug regression performance or correctness bugs by tracking how the behavior of the program changed across versions.

Second, a developer can use the `OmniTable` to inspect the execution of separate nodes in a distributed system. This abstraction enables the `OmniTable` query model to subsume prior work in the space of high-level inspection tools for distributed systems, such as pivot tracing. There is interesting work to be done in this space, primarily for efficient record-and-replay in distributed settings, since the categorization of inputs into deterministic and non-deterministic is completely upended in a distributed setting where multiple nodes can cooperatively replay a distributed execution.

Third, a developer can use the `OmniTable` query model to inspections multiple executions of different programs. Cross-program execution inspection might be useful for

learning the "unwritten rules" of a programming library. These inspection queries might enable the sharing of bug detection predicates across applications. Further, statistical bug isolation is potentially possible across applications that have some similarities, such as using the same libraries.

Cross-execution inspection queries present a number of systems challenges. How can a system feasible support queries that require re-executing hundreds or thousands of executions? Solutions from the database community stand out as possibilities for enable these queries. For example, caching and indexing strategies could precompute predicates that identify equivalent or related executions. The intellectual challenges are vast, but so too are the payoffs.

6.2 On the Adoption of Data-Centric Execution Inspection

This dissertation shows the benefits of data-centric execution inspection, and we am fascinated with how we can make these ideas and tools amenable to typical software system deployments.

Record-and-replay is an enabling technology for data-centric execution inspection; it is difficult to imagine completing the work in this thesis without building on these ideas. Nevertheless, inn my opinion, the biggest challenge with the adoption of data-centric execution inspection is the current reliance on record-and-replay. Record-and-replay has been known for decades and had immense impact the research community. However, deployed software has been extremely slow to adopt record-and-replay technology. There are a variety of reasons why, including maintainability limitations (e.g., Arnold [23] requires tens-of-thousands of kernel source code lines, Castor [56] requires new compiler tool-chains), performance issues (e.g., Mozilla RR [66] serializes threads during recording and imposes order of magnitude slowdowns), and limited effectiveness (e.g., Arnold [23] cannot support racy program).

We see two possible approaches to resolve the limitations of record-and-replay. First, we could construct record-and-replay systems with more appropriate tradeoffs. REPT [19] and ER [93] alleviate many of the maintainability, performance, and effectiveness challenges of early systems and may pave the way for adoption. New deployment models, such as heterogeneous systems, microservices, and lambda functions may also offer better tradeoffs for more effective record-and-replay technology. It remains to be seen whether we can overcome the hurdles of record-and-replay, but there is hope.

Second, we should look for opportunities to treat an execution as a data object without relying on record-and-replay. Are there deployment models and systems that are

"deterministic-enough" for data-centric execution inspection? Can we leverage RaceMob-like [39] parallelism to split `OmniTable` queries across equivalent independent program executions rather than querying a single execution? Exploring this space of applications outside of record-and-replay is a fantastic step towards better applicability for these ideas, but also presents a number of major systems challenges. For example, employing RaceMob-Like parallelism to partition `OmniTable` queries across executions will require low-overhead methods for identifying executions that are equivalent.

6.3 Towards Data-Centric Software Reliability

This dissertation shows how a data-centric view enables fundamentally more powerful execution inspection. However, software reliability requires more than just execution inspection, it also requires bug detection, bug response, and bug avoidance. Data-centric execution inspection has the potential to be a building block for these tools, helping us design better systems by making it easier to understand what they are doing.

As we look to the next decade, we are fascinated with how data-oriented approaches can more generally improve software reliability. Our software is pervasively deployed, and generates untold amounts of execution data. Most of this data is discarded and treated as useless; the original vision of the Eidetic System [23] was that we could feasibly store all of this information. The vision of data-centric software reliability is that we can use this data to make our software systems more correct, efficient, and secure.

BIBLIOGRAPHY

BIBLIOGRAPHY

[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] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, Hollywood, CA, October 2012.

[4] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, Vancouver, BC, October 2010.

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

[6] R. M. Balzer. Exdams: Extendable debugging and monitoring system. In *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference*, AFIPS '69 (Spring), pages 567–580, New York, NY, USA, 1969. ACM.

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

[8] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *NDSS*, 2018.

[9] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. On the dichotomy of debugging behavior among programmers. In *Proceedings of the 40th International Conference on Software Engineering*, pages 572–583, 2018.

[10] Bug 25520. https://bz.apache.org/bugzilla/show_bug.cgi?id=25520.

[11] Bug 45605. https://bz.apache.org/bugzilla/show_bug.cgi?id=45605.

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

[13] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 15–28, Boston, MA, June 2004.

[14] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 39–50, New York, NY, USA, 2008. ACM.

[15] P.M. Chen and B.D. Noble. When Virtual is Better Than Real. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems)*, Schloss Elmau, Germany, May 2001.

[16] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Mchiael Ryan, and Evangelos Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, Beijing, China, June 2008.

[17] Jim Chow, Tal Garfinkel, and Peter M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 1–14, June 2008.

[18] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[19] 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.

[20] Jeffrey Dean and Luiz Andre Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, February 2013.

[21] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCarmant, Jeff H. Perkins, and Martin Rinard. Inference and enforcement of data structure consistency specifications. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analaysis*, ISSTA 2006, July 2006.

[22] 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. ACM, 2018.

[23] 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.

[24] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 211–224, Boston, MA, December 2002.

[25] Marc Eisenstadt. My hairiest bug war stories. *Communications of the ACM*, 40(4):30–37, 1997.

[26] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, Vancouver, BC, October 2010.

[27] 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.

[28] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2), February 2001.

[29] 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.

[30] David J Gilmore. Models of debugging. *Acta psychologica*, 78(1-3):151–172, 1991.

[31] 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.

[32] John D Gould and Paul Drongowski. An exploratory study of computer program debugging. *Human Factors*, 16(3):258–277, 1974.

[33] Philip J. Guo and Dawson Engler. Using automatic persistent memoization to facilitate data analysis scripting. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA)*, pages 287–297, New York, NY, USA, 2011.

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

[35] N. Honarmand and J. Torrellas. Replay debugging: Leveraging record and replay for program debugging. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 455–456, June 2014.

[36] Kangkook Jee, Vasileios P. Kermerlis, Angelos D. Keromytis, and Georgios Portokalidis. ShadowReplica: Efficient parallelization of dynamic data flow tracking. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, November 2013.

[37] Kangkook Jee, Georgios Portokalidis, Vasileios P. Kermerlis, Soumyadeep Ghosh, David I. August, and Angelos D. Keromyrtis. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *Proceedings of the 19th Network and Distributed System Security Symposium*, San Diego, CA, February 2012.

[38] Ashlesha Joshi, Sam T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 91–104, Brighton, United Kingdom, October 2005.

[39] Baris Kasikci, Cristian Zamfir, and George Candea. Racemob: Crowdsourced data race detection. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 406–422, New York, NY, USA, 2013. Association for Computing Machinery.

[40] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, 2012.

[41] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich. Efficient patch-based auditing for Web application vulnerabilities. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, Hollywood, CA, October 2012.

[42] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 223–236, Bolton Landing, NY, October 2003.

[43] 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.

[44] Herb Krasner. The cost of software quality in the us: A 2020 report. January 2021.

[45] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the 2004 IEEE/ACM International Symposium on Code Generation and Optimization*, 2004.

[46] Michael A Laurenzano, Yunqi Zhang, Lingjia Tang, and Jason Mars. Protean code: Achieving near-free online code transformations for warehouse scale computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 558–570, 2014.

[47] 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.

[48] 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.

[49] 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.

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

[51] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes — a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–339, 2008.

[52] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.

[53] 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.

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

[55] 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.

[56] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. Towards practical default-on multi-core record/replay. In *Proceedings of the 22th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

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

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

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

[60] memtier_benchmark: A high-throughput benchmarking tool for redis & memcached, June 2013.

[61] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. TaintPipe: Pipelined symbolic taint analysis. In *Proceedings of the 24th Usenix Security Symposium*, Washington, D.C., August 2015.

[62] JR Minkel. The 2003 northeast blackout—five years later. *Scientific America*, August 2008.

[63] 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.

[64] James Newsome and Dawn Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, February 2005.

[65] 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.

[66] 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.

[67] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, and Brad Chen. Locating system problems using dynamic instrumentation. In *Proceedings of the Linux Symposium*, pages 49–64, Ottawa, ON, Canada, July 2005.

[68] Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting general security attacks. In *The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '06)*, Orlando, FL, 2006.

[69] Andrew Quinn, David Devecsery, Peter M Chen, and Jason Flinn. Jetstream: Cluster-scale parallelization of information flow queries. In *OSDI*, pages 451–466, 2016.

[70] 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.

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

[72] 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.

[73] 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.

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

[75] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '19, New York, NY, USA, 2019. Association for Computing Machinery.

[76] David Shepardson. Gm recalls 4.3 million vehicles over air bagrelated defect. https://www.reuters.com/article/us-gm-recall/gm-recalls-4-3-million-vehicles-over-air-bag-related-defect-idUSKCN11F2AH, September 2016. [Online; posted 9-September-2016].

[77] 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.

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

[79] 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.

[80] Michael Stonebraker and Lawrence A. Rowe. The design of postgres. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 1986. Association for Computing Machinery.

[81] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 269–280, New York, NY, USA, 2010. ACM.

[82] George Tassey. The economic impacts of inadequate infrastructure for software testing. 2002.

[83] 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.

[84] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. DoublePlay: Parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Long Beach, CA, March 2011.

[85] Nicolas Viennot, Siddharth Nair, and Jason Nieh. Transparent mutable replay for multicore debugging and patch validation. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2013.

[86] Vincent M. Weaver, Dan Terpstra, and Shirley Moore. Non-determinism and overcount on modern hardware performance counter implementations. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 215–224, 2013.

[87] 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.

[88] Zack Whittaker. Two years after wannacry, a million computers remain at risk. *TechCrunch*, May 2019.

[89] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 133–147, New York, NY, USA, 2020. Association for Computing Machinery.

[90] 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.

[91] Ding Yuan, Soyeun Park, and Yuanyuan Zhou. Characterising logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, Zurich, Switzerland, June 2012.

[92] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: Dynamic bandwidth sharing in a dbms. In *VLDB*, 2007.

[93] Gefei Zuo, Jiacheng Ma, Andrew Quinn, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. *Execution Reconstruction: Harnessing Failure Reoccurrences for Failure Reproduction*, page 1155–1170. Association for Computing Machinery, New York, NY, USA, 2021.