Sledgehammer: Cluster-fueled debugging

Andrew Quinn, Jason Flinn, and Michael Cafarella
University of Michigan

Abstract

Current debugging tools force developers to choose between power and interactivity. Interactive debuggers such as gdb let them quickly inspect application state and monitor execution, which is perfect for simple bugs. However, they are not powerful enough for complex bugs such as wild stores and synchronization errors where developers do not know which values to inspect or when to monitor the execution. So, developers add logging, insert timing measurements, and create functions that verify invariants. Then, they re-run applications with this instrumentation. These powerful tools are, unfortunately, not interactive; they can take minutes or hours to answer one question about a complex execution, and debugging involves asking and answering many such questions.

In this paper, we propose *cluster-fueled debugging*, which provides interactivity for powerful debugging tools by parallelizing their work across many cores in a cluster. At sufficient scale, developers can get answers to even detailed queries in a few seconds. Sledgehammer is a cluster-fueled debugger: it improves performance by timeslicing program execution, debug instrumentation, and analysis of results, and then executing each chunk of work on a separate core. Sledgehammer enables powerful, interactive debugging tools that are infeasible today. *Parallel retro-logging* allows developers to change their logging instrumentation and then quickly see what the new logging would have produced on a previous execution. *Continuous function evaluation* logically evaluates a function such as a data-structure integrity check at every point in a program's execution. *Retro-timing* allows fast performance analysis of a previous execution. With 1024 cores, Sledgehammer executes these tools hundreds of times faster than single-core execution while returning identical results.

1 Introduction

Debugging is onerous and time-consuming, comprising roughly half of all development time [24]. It involves detective work: using the tools at her 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 non-deterministic data corruption in a Web server. She cannot use gdb because she does not yet know which values to inspect or which part of the server execution to monitor. So, she employs more heavyweight tools. She adds 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, she could ask complex questions about her 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 [38] or invariant checks, recompiling, 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 [17, 32, 34] and commercial [27, 31] 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 [6, 15, 36] 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 [9], 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.

2 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.

3 Debugging tools

We have created three new parallel debugging tools.

3.1 Parallel retro-logging

Retro-logging [6, 15, 36] 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 [22] and Valgrind [26] that also isolate debugging code from the application have high overhead. Sledgehammer hides this overhead via parallelization.

3.2 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.

3.3 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 nondeterministic 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 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.

4.1 Atomicity Violation

Concurrency errors such as atomicity violations are notoriously difficult to find and debug [21]. 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, she decides 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 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

```
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 1: Tracer for the first memcached query.

```
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 2: Analyzer for the first memcached query.

show the function she would write in Figure 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.

```
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 3: Tracer for the second memcached query.

Figure 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. She adds 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 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.

4.2 Apache 45605

In this previously reported bug [3], 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

```
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: Analyzer for the second memcached query.

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 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 Apache 25520

In another previously-reported bug [2], 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 5.2). 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.

4.4 Data corruption

Memory corruption is a common source of software bugs [20] 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 developer 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.

4.5 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 [20]. 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.

4.6 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 [26] 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.

4.7 Lock Contention

Rare performance anomalies are hard to debug. One common source of performance anomalies is lock contention [33]. 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. She runs a profiler, but the tool reports only average behavior, which obscures the occasional outlier. So, she sprinkles timing measurements at key points in her 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, she retroactively adds even more timing code, and she can iterate quickly to drill down to the suspect lock acquisition. Her 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.

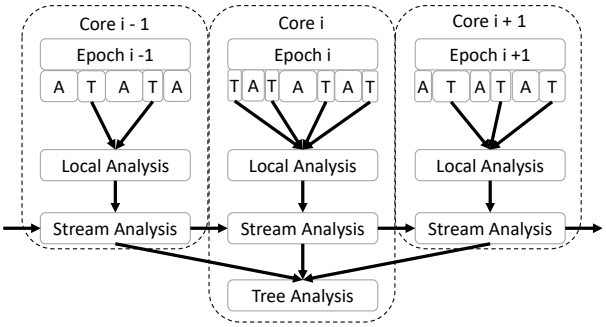

Figure 5: Sledgehammer architecture overview. A replay is divided into n epochs. Each core runs an epoch by executing the original application (A boxes) and injecting tracers (T boxes). Local analysis runs on each core with output from a single epoch, stream analysis takes input from previous epochs and sends output to subsequent ones. Tree analyzers combine output from multiple epochs.

5 Design and implementation

Figure 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 5.2 discusses how developers specify tracers by annotating their source code to add logging and instrumentation. Sledgehammer parses the source code to extract the tracers, the arguments passed to each tracer, and the locations where tracers should be invoked.

Section 5.3 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 5.4, 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 5.5 discusses how analyzers process the stream of output from tracers. As shown in Figure 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.

5.1 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 [11] 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 [35] 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 [10] for deterministic record and replay due to its low overhead (less than 10% for most workloads) and because Arnold supports epoch parallelism [29]. 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 [6, 15, 16]. 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.

5.2 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.

5.2.1 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 5.4.1 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 [18] 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 6.5 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.

5.2.2 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.

5.3 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 [10]. 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.

5.4 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.

5.4.1 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.

5.4.2 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 *monitor 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.

5.4.3 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 6.6).

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 1 us.

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.

5.5 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 analysis routine and run it as a tree analyzer at the root of the tree. However, Section 6.4 reports substantial performance benefits for many queries from using local, stream, and tree aggregation to parallelize analysis.

6 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?

6.1 Experimental Setup

We evaluated Sledgehammer using a CloudLab [30] cluster of 16 r320 machines (8-core Xeon E5-2450 2.1 GHz processors, 16 GB RAM, and 10 Gb NIC). Since several applications we evaluate use at least 2 GB 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 5.3 (e.g., parallel checkpoint genera-

Benchmark	Application	Replay time(s)	Tracer calls (millions)	1 Core query time(s)	64 Cores query time(s)	64 Cores speedup	1024 Cores query time(s)	1024 Cores speedup
Data corruption	nginx	2.0	7.7	324.8 (±2.2)	7.2 (±0.2)	45 (±1.0)	1.0 (±0.0)	330 (±13)
Wild store	MongoDB	30.1	3.4	7688.4 (±13.5)	181.3 (±6.0)	42 (±2.0)	17.2 (±1.0)	446 (±15)
Atomicity violation	memcached	98.5	42.8	7852.4 (±20.1)	173.1 (±1.2)	45 (±0.3)	13.7 (±0.7)	573 (±15)
Memory leak	nginx	76.0	3.6	1575.2 (±8.1)	30.3 (±0.2)	52 (±0.3)	2.8 (±0.2)	559 (±25)
Lock contention	memcached	93.4	75.5	3281.8 (±17.5)	68.3 (±0.6)	48 (±0.5)	10.9 (±1.2)	301 (±16)
Apache 45605	Apache	50.7	1.9	249.9 (±1.1)	5.2 (±0.5)	48 (±0.5)	1.0 (±0.6)	255 (±15)
Apache 25520	Apache	60.1	3.7	717.3 (±1.6)	12.9 (±0.0)	55 (±0.2)	1.2 (±0.0)	601 (±4.4)

Table 1: 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.

tion) 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 measured 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.

6.2 Benchmarks

We reproduce the 7 scenarios described in Section 4 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. 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 [7].
- **Atomicity violation** We use memtier [25] 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 [25] 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 [37] and run the final query.
- **Apache 25520** We recreate the bug by stress testing Apache and run the final query.

6.3 Scalability

Table 1 shows results for the 7 scenarios. The first column shows the time to replay the execution with no debugging. The next column shows the number of tracers executed during the query. The remaining columns compare sequential (1 core) query time with performance at 64 and 1024 cores, respectively.

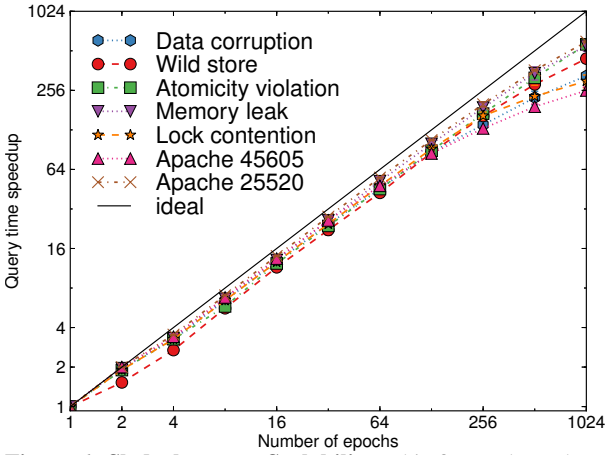

Figure 6: Sledgehammer Scalability. This figure shows how query time improves as the number of cores increases.

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.

Figure 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.

6.3.1 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.

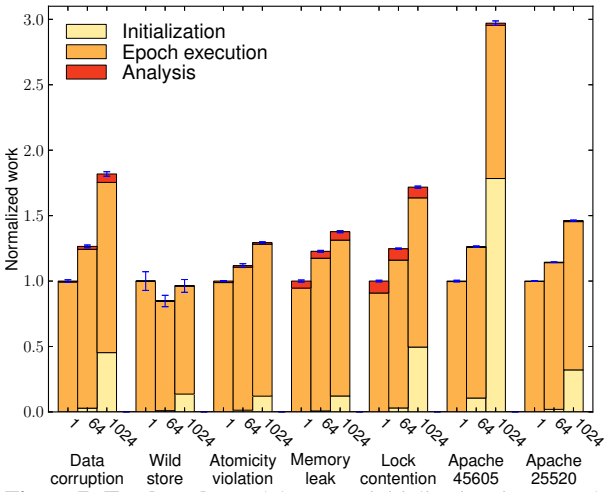

Figure 7: Total work. Each bar sums initialization time, epoch execution time, and analysis time overs all epochs. This shows how much extra work is created by parallelization.

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 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 done by different epochs. Outlier epochs lead to high tail latency [8]. We quantify skew in Table 2 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 6.

	Skew	
Benchmark	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 2: Skew. The reported values are the longest epoch execution time divided by the average execution time.

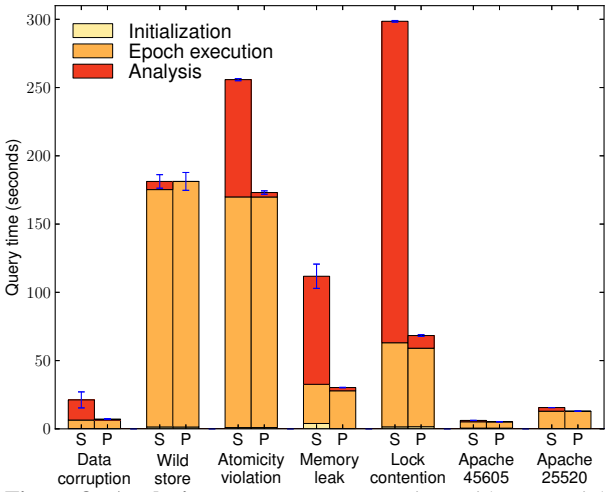

Figure 8: Analysis. We compare query time with sequential (S) and parallel (P) analysis using 64 cores. The regions in each bar show how much time is spent in each phase along the critical path of query processing.

6.4 Benefit of parallel analysis

We next quantify how much benefit is achieved by parallelizing analysis. Figure 8 compares query response time for sequential analysis and parallel analysis using the analyzers for each query described in Section 4. We show results with 64 cores, i.e., the largest number of cores we can support without emulation.

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.

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

6.5 Isolation

Table 3 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.

6.6 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.

7 Related Work

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 [11] is a great help in debugging [5, 17, 27, 32, 36]. Sledgehammer leverages Arnold [10] replay both to ensure that results of successive queries are consistent and also to parallelize work via epoch parallelism [35]. JetStream [29] uses epoch parallelism for a different task: dynamic information flow tracking (DIFT). Sledgehammer's 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 [4, 28]. Execution mining [19] 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 [12] lets users introspect at the start and end of functions but injected code must be side-effect free. Pivot tracing [23] 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 using fork-based approaches [6, 15]; this comes with high overhead. Others use binary rewriting approaches for isolation such as Pin and Valgrind; these tools do not scale to thousands of cores [29]. Sledgehammer reduces isolation overhead through compiler-based isolation and hides remaining overhead through parallelization. Like Sledgehammer, rdb [14] allows users to modify source code and executes the modifications during replay; however, rdb prohibits program state modifications instead of isolating them. Dora [36] 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 [9] or detect likely invariants [13]. Continuous function evaluation takes this to an extreme by logically running a function after every instruction. X-Ray [1] 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.

8 Conclusion

Sledgehammer is a cluster-fueled debugger: it makes powerful debugging tools interactive by parallelizing application and tool execution, as well as analysis, across many cores in a cluster. This makes tools such as 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.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Rebecca Isaacs, for their thoughtful comments. This work has been supported by the National Science Foundation under grants CNS-1513718 and CNS-1421441, and by NSF GRFP and MSR Ph.D Fellowships. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] ATTARIYAN, M., CHOW, M., AND FLINN, J. 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).

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

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

[4] CANTRILL, B. M., SHAPIRO, M. W., AND LEVENTHAL, A. H. Dynamic instrumentation of production systems. In *Proceedings of the 2004 USENIX Annual Technical Conference* (Boston, MA, June 2004), pp. 15–28.

[5] CHEN, P., AND NOBLE, B. When Virtual is Better Than Real. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems)* (Schloss Elmau, Germany, May 2001).

[6] CHOW, J., GARFINKEL, T., AND CHEN, P. M. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the 2008 USENIX Annual Technical Conference* (June 2008), pp. 1–14.

[7] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), pp. 143–154.

[8] DEAN, J., AND BARROSO, L. A. The tail at scale. *Communications of the ACM 56*, 2 (February 2013), 74–80.

[9] DEMSKY, B., ERNST, M. D., GUO, P. J., MCCARMANT, S., PERKINS, J. H., AND RINARD, M. Inference and enforcement of data structure consistency specifications. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analaysis* (July 2006).

[10] DEVECSERY, D., CHOW, M., DOU, X., FLINN, J., AND CHEN, P. M. Eidetic systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation* (Broomfield, CO, October 2014).

[11] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 211–224.

[12] ERLINGSSON, U., PEINADO, M., PETER, S., AND BUDIU, M. Fay: Extensible distributed tracing from kernels to clusters. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (October 2011), pp. 311–326.

[13] ERNST, M. D., COCKRELL, J., GRISWOLD, W. G., AND NOTKIN, D. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering 27*, 2 (February 2001).

[14] HONARMAND, N., AND TORRELLAS, J. Replay debugging: Leveraging record and replay for program debugging. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)* (June 2014), pp. 455–456.

[15] JOSHI, A., KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 91–104.

[16] KIM, T., CHANDRA, R., AND ZELDOVICH, N. Efficient patch-based auditing for Web application vulnerabilities. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation* (Hollywood, CA, October 2012).

[17] KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Annual Technical Conference* (April 2005), pp. 1–15.

[18] LATTNER, C., AND ADVE, V. 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).

[19] LEFEBVRE, G., CULLY, B., HEAD, C., SPEAR, M., HUTCHINSON, N., FEELEY, M., AND WARFIELD, A. Execution Mining. In *Proceedings of the 2012 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)* (March 2012).

[20] LI, Z., TAN, L., WANG, X., LU, S., ZHOU, Y., AND ZHAI, C. 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* (2006), ACM, pp. 25–33.

[21] LU, S., PARK, S., SEO, E., AND ZHOU, Y. 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* (2008), pp. 329–339.

[22] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation* (Chicago, IL, June 2005), pp. 190–200.

[23] MACE, J., ROELKE, R., AND FONSECA, R. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles* (2015).

[24] MCCONNELL, S. *Code complete*. Pearson Education, 2004.

[25] memtier_benchmark: A high-throughput benchmarking tool for redis & memcached, June 2013. `https://redislabs.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached/`.

[26] NETHERCOTE, N., AND SEWARD, J. 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).

[27] O'CALLAHAN, R., JONES, C., FROYD, N., HUEY, K., NOLL, A., AND PARTUSH, N. Engineering record and replay for deployability. In *Proceedings of the 2017 USENIX Annual Technical Conference* (Santa Clara, CA, July 2017).

[28] PRASAD, V., COHEN, W., EIGLER, F. C., HUNT, M., KENISTON, J., AND CHEN, B. Locating system problems using dynamic instrumentation. In *Proceedings of the Linux Symposium* (Ottawa, ON, Canada, July 2005), pp. 49–64.

[29] QUINN, A., DEVECSERY, D., CHEN, P. M., AND FLINN, J. JetStream: Cluster-scale parallelization of information flow queries. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation* (Savannah, GA, November 2016).

[30] RICCI, R., EIDE, E., AND THE CLOUDLAB TEAM. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login: 39*, 6 (Dec. 2014).

[31] rr: lightweight recording and deterministic debugging. `http://www.rr-project.org`.

[32] SRINIVASAN, S., ANDREWS, C., KANDULA, S., AND ZHOU, Y. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Annual Technical Conference* (Boston, MA, June 2004), pp. 29–44.

[33] TALLENT, N. R., MELLOR-CRUMMEY, J. M., AND PORTERFIELD, A. Analyzing lock contention in multithreaded applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2010), PPoPP '10, ACM, pp. 269–280.

[34] VEERARAGHAVAN, K., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. Detecting and surviving data races using complementary schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (Cascais, Portugal, October 2011).

[35] VEERARAGHAVAN, K., LEE, D., WESTER, B., OUYANG, J., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. 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).

[36] VIENNOT, N., NAIR, S., AND NIEH, J. 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).

[37] YU, J., AND NARAYANASAMY, S. A case for an interleaving constrained shared-memory multiprocessor. In *Proceedings of the 36th International Symposium on Computer Architecture* (June 2009), pp. 325–336.

[38] YUAN, D., PARK, S., AND ZHOU, Y. Characterising logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)* (Zurich, Switzerland, June 2012).