Peeking into the Past: Efficient Checkpoint-assisted Time-traveling Debugging

Armando Miraglia*, Dirk Vogt*, Herbert Bos*, Andy Tanenbaum*, Cristiano Giuffrida*
*Vrije Universiteit Amsterdam

***Abstract*—Debugging long-lived latent software bugs that manifest themselves only long after their introduction in the system is hard. Even state-of-the-art record/replay debugging techniques are of limited use to identify the root cause of long-lived latent bugs in general and event-driven bugs in particular.**

We propose *DeLorean*, a new end-to-end solution for time-travelling debugging based on fast memory checkpointing. Our design trades off replay guarantees with efficient support for history-aware debug queries (or time-travelling introspection) and provides novel analysis tools to diagnose event-driven latent software bugs. *DeLorean* imposes low run-time performance and memory overhead while preserving in memory as much history information as possible by deduplicating and/or compressing the collected data. We evaluate *DeLorean* by extensive experimentation, exploring the performance-memory tradeoffs in different configurations and comparing our results against state-of-the-art solutions. We show that *DeLorean* can efficiently support high-frequency checkpoints and store millions of them in memory.

I. Introduction

Debugging sessions can be time-consuming, frustrating and expensive, especially for long-lived latent software bugs that require a long time to manifest themselves. Such latent bugs often result from particular *events* in the input. For example, consider a server receiving hundreds of thousands of requests, for which a single bad request corrupts its state. Unless the corruption leads to an immediate crash, rather than a failure or a misbehavior at a later time, it becomes hard to track down the exact request that caused it. We present a new solution that lets programmers efficiently create high-frequency checkpoints, store *millions* of them in memory, and efficiently search through them to look for the root cause of such event-driven latent bugs.

Our work fits under the general umbrella of targeted debugging solutions for "hard" bugs that also includes recent work on reproducing rare concurrency bugs [1], [2]. In fact, from the wide range of approaches based on record/replay [3], [4], [5], [6], [7], [8] and execution synthesis [2], to the recently proposed failure sketches [1], most existing solutions focus solely on buggy data races. In contrast, we aim for the *other* major category of hard bugs: latent corruption that only manifests itself millions of events later.

Unfortunately, even the most advanced record/replay debuggers [9], [8], [10], [11] are a poor match for such bugs. In theory, it is simply a matter of replaying a recorded trace, using either forward or backward execution, with additional breakpoints and watchpoints, until we find the target condition, but in practice this is inefficient and cumbersome.

First, despite the progress in recent years in bringing down the overhead [10], [8], record/replay systems are still *impractical*. This is true not just for the recording side (a popular optimization target for these systems), but especially for the replay side which is equally important in reducing the time to track down elusive bugs. Second, pure record/replay systems are *linear* by nature. Finding a bug typically requires at least one linear execution of a trace (i.e., re-executing the entire recorded trace from the beginning to the end.). In contrast, for many classes of bugs, we can examine a collection of snapshots much more efficiently, e.g., using binary search. Third, record/replay systems are mostly useful for understanding the problem once the trigger event is known, but offer limited help in *identifying* the trigger (e.g., by querying the history for specific events). Finally, and very pragmatically, the fastest record/replay systems today require *extensive kernel modifications*, making them hard to deploy (compared to, say, a self-contained kernel module).

In contrast, we believe that high-frequency memory checkpointing [12], [13], [14], [15], [16] (i.e., frequently taking a snapshot of a process' memory state) offers an alternative and complementary solution that couples low run-time overhead with a searchable trail of snapshots of the program state. Of course, this is only possible if we can make both the recording and the querying of the history sufficiently lightweight.

Memory checkpointing has already proven useful in the domains of error recovery [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [14], [28], record/replay debugging [29], [9], [4], [8], [10], application-level backtracking [30], [31], and periodic memory rejuvenation [32], but these solutions are all too heavyweight for the high-frequency checkpoints we need for debugging latent software bugs. We require checkpointing at high speed and fine granularity, say for every incoming request of a loaded server program (to record all the potential inputs of interest). Moreover, to address very long-lived latent bugs, the checkpoint history should be as long as possible, which poses the challenge of storing memory checkpoints efficiently. Finally, as the number of checkpoints can be high (potentially millions), this vast amount of data has to be processed by the user in a manageable manner.

Based on these observations, we propose *DeLorean*, a new end-to-end solution for time-travelling debugging of event-driven latent bugs by means of high-frequency checkpointing. *DeLorean* targets medium-sized applications such as web and database servers. The idea consists in taking frequent checkpoints for all the events of interest and then quickly querying the collected data for a given condition (e.g., using binary search). To implement both phases efficiently, *DeLorean* does not support full record/replay functionalities but instead focuses on automating and speeding up the process of diagnosing and determining the root-cause of event-driven latent soft-

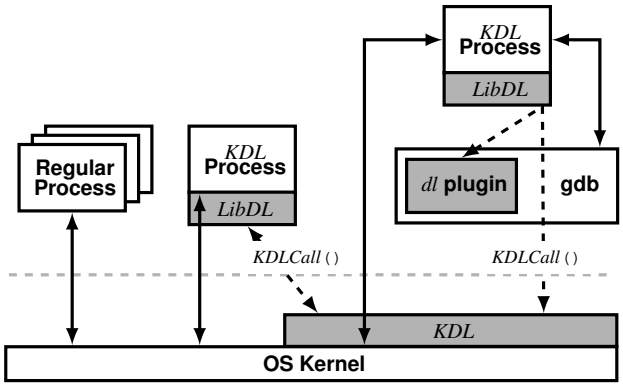


Fig. 1: Architecture overview of *DeLorean*, with both its user- and kernel-level components and their interactions.

ware bugs. To maintain a sufficiently long checkpoint history, *DeLorean* employs memory *deduplication* and/or *compression*, increasing the history size up to 10 times compared to plain checkpointing. Moreover, *DeLorean* is fully integrated into `gdb` offering effective ways to introspect the entire checkpoint history efficiently. Finally, *DeLorean* is easy to deploy, since we implemented its core functionalities in a self-contained Linux kernel module.

Although we present fast checkpointing mainly as a stand-alone debugging solution, we emphasize that our work is complementary to many existing debugging approaches. For instance, existing record/replay solutions often also use checkpointing to implement reverse debugging. However, as the checkpointing in these systems is expensive, they can take snapshots only very infrequently—up to once per 10-25 seconds in some cases [4]. Our high-frequency checkpointing strategy brings this number down to milliseconds.

Contribution. Our contribution is threefold. (a) We present *DeLorean*, a debugging solution prototype based on high-frequency memory checkpointing. *DeLorean* targets event-driven latent bugs and combines excellent run-time performance with an efficient memory footprint (allowing for millions of checkpoints). (b) We propose a new mechanism to perform fast queries on the collected checkpoints called *time-travelling introspection*. With such mechanism in place, we show that checkpoint-assisted debugging is a viable option to diagnose event-driven latent software bugs. (c) We evaluate *DeLorean* effectiveness, performance, and memory usage guarantees and compare our results to state-of-the-art debugging solutions.

II. Architecture Overview

Figure 1 depicts the high-level system architecture of *DeLorean*. *DeLorean* consists of *KDL*, the kernel module, *LibDL*, the shared library, and *dl*, the `gdb` plugin. This simple architecture makes *DeLorean* easy to deploy, without the need to recompile the kernel or the target application.

The *KDL* kernel module enables *DeLorean* checkpointing features and offers them to the userland through the *KDLCall* interface. *KDL* can handle multiple checkpointed processes and keeps all the checkpointed data in memory. Further, it offers several configuration parameters, e.g., the maximum number M of most recent checkpoints in its history, and incorporates several mechanisms to reduce the amount of checkpointed data. While *KDL* is part of the *DeLorean* debugging framework, the exported checkpointing features are generally applicable to other use cases that require support for time- and space-efficient memory checkpointing.

The *LibDL* user library unburdens debugged applications to directly interact with the *KDLCall* interface, our API to the kernel module. This library can be preloaded in the target application, as done by *dl*, but can also be directly linked to applications so they can use *KDL* checkpointing facilities for other use cases, such as backtracking [30], [33] and error recovery [17], [18], [19], [20], [21], [24].

We implemented the *dl* debugger as a generic `gdb` plugin that complements all the standard debugging functionalities already available in `gdb` with support for checkpoint-assisted debugging. This is possible by exporting custom commands in `gdb`, as detailed later in Section III.

To register an application with *DeLorean*, users simply start a `gdb` session with our *dl* plugin, which automatically sets up all the other *DeLorean* components. Note that running processes not being debugged by *DeLorean* are unaffected by the operation. By default, *DeLorean* monitors the entire address space of the targeted process. However, *DeLorean* allows users to limit the checkpoint surface to specific portions of the address space during the initialization (and configuration) phase. Each process in the application will be registered with *DeLorean* until it exits or is explicitly unregistered by the user.

Once an application is registered with *DeLorean*, the system provides users with a number of features: (a) take a new checkpoint, (b) rollback to a previous checkpoint, (c) restore the current checkpoint to continue execution, (d) and *query the checkpointed history*.

While the first three operations also feature in traditional checkpointing systems, querying the checkpointed history is a new feature offered by *DeLorean*. It allows the user to specify complex search queries and efficiently evaluate the given conditions through the checkpoint history. This is similar, in spirit, to temporal queries supported in modern databases with versioning support. Moreover, *DeLorean* may perform condition evaluation through the checkpointed history either *linearly* (i.e., through a linear search) or using a more efficient *bisect* approach (i.e., using a binary search, not unlike `git bisect` [34])—provided that the continuity of the condition throughout the collected history allows for it. This is typically the case for the long-lived effects caused by the event-driven latent bugs targeted by *DeLorean* (e.g., memory corruption).

Moreover, while memory checkpointing is fast, it generally incurs memory overhead which can grow significantly with the number of checkpoints in the history [16], [15], [14]. For this reason, we have implemented various modes of operation in *KDL* that allow users to tune the checkpoiting feature. Specifically, users can enable memory *deduplication* and/or *compression* features in *KDL* to allow for the storage of a large checkpoint history but trading off run-time performance. In the following sections, we detail the design and the features of *DeLorean* debugger.

ref.	checkpointing
[a]	*dl* `cp take [LOCATION [if <CONDITION>]]` Starts a new checkpointing interval. When a location is specified, a breakpoint is set that triggers the new interval. If a condition is given, the interval starts only when the condition is satisfied.
[b]	*dl* `rb <CP ID>` Rolls back the memory state of the process to the checkpoint with the specified ID.
[c]	*dl* `restore` Restores the memory state to the "present" state.
	time-travelling introspection
[d]	*dl* `for <CP SPEC> if <CONDITION> [do <CMD>]` For all the specified checkpoints, *dl* rolls back and evaluates the provided condition. If a `do-statement` is specified, the list of commands are executed when the condition is satisfied.
[e]	*dl* `for <CP SPEC> do <CMD>` For all the specified checkpoints, *dl* rolls back and executes the provided commands.
[f]	*dl* `search <CONDITION>` Searches for the furthest checkpoint in time which satisfies the condition. The command uses either a linear or a binary search. If such a checkpoint exists, the state is ultimately rolled back to the selected checkpoint.

TABLE I: Subset of *dl* commands. `CP ID` represents the numeric identifier of a checkpoint interval. `CP SPEC` indicates a set of checkpoints specified as arrays of IDs and/or intervals.

III. User-space Debugger

While *DeLorean* checkpointing functionalities reside in *KDL*, users do not have to interact with the kernel module directly. They only have to make sure that the kernel module is loaded, as our system exposes all the debugging functionalities through *dl*, our checkpoint-assisted time-traveling debugging tool. *dl* is implemented as a `gdb` plugin and allows to frequently checkpoint the target process while preserving checkpointed data for millions of checkpoints.

An extract of some of the commands exported by *dl* is shown in Table I. The Table also contains the references for each command used throughout this section.

A. Initialization

A debugging session is initiated in the same way a user would normally start a standard `gdb` session but additionally specifying *dl* as a `gdb` plugin from the command line. The user needs to load the kernel module before starting the session. After that, *dl* preloads the *LibDL* shared library. As soon as the application starts running, *dl* puts the corresponding process into *DeLorean* mode transparently to the user. This ensures the entire process address space is by default registered with *KDL*, except for the memory regions reserved for *LibDL*. Furthermore, to rollback the stack area of the process safely, *dl* starts a dedicated worker thread owned by the target process. The worker thread owns a private memory area used for its stack and metadata, which is also excluded from the checkpointing surface. The initialization starts an implicit initial *checkpoint interval*[1] by write-protecting all registered memory pages[2]. This is necessary to checkpoint the target pages efficiently and incrementally, as detailed later in Section IV.

[1] A *checkpoint interval* is the portion of execution between two consecutive checkpoints.

[2] A *memory page* (or simply *page*) is the smallest fixed-length contiguous block of virtual memory—typically 4KB.

B. Checkpoint

The main features of *dl* are the capability to take checkpoints, rollback the memory to an older state and restore for further execution (Table I show an extract of the commands).

While the user can take new checkpoints explicitly (starting new checkpoint intervals) by issuing [a] without arguments, this is of limited use by itself, as it requires the user to interrupt the process. A more interesting option for our target domain is to checkpoint the memory image of the target process automatically when particular events occur during execution (e.g., a new request received by a server program). The user can associate *checkpoint requests* to the program events of interest by additionally supplying a location and, possibly, a condition to be evaluated. This is very similar to setting a new breakpoint in `gdb`.

We initially implemented our checkpoint request functionality on top of the `gdb` breakpoint mechanism. However, we observed that piggybacking on `gdb` breakpoints was relatively expensive. Whenever the program hits a breakpoint associated to the checkpoint request, it first transfers CPU to `gdb` which, in turn, hands over control to *LibDL* to perform the checkpoint operation, and later resumes the execution. Unfortunately, the transfer is costly. Static instrumentation, i.e., instrumenting the points of interest with direct calls into *LibDL*, was one option we considered to reduce the overhead. Efficient static instrumentation, however, would typically require recompiling the target application. For this reason, we opted instead for a design based on hardware breakpoints. We extended *KDL* adding an interface to set hardware breakpoints and automatically take checkpoints whenever a breakpoint is hit. We ran a microbenchmark to analyze the overhead of using breakpoints to issue checkpoint requests. We estimated CPU cycles by averaging 1,000 executions of each test. While a checkpoint request directly issued by a process (*static instrumentation*) costs approximately $14K$ cycles, a software breakpoint is more than two orders of magnitude slower (ca. $2.7M$ cycles). Checkpoints issued by *DeLorean* hardware breakpoint handler, while being somewhat more expensive (ca. $19k$ cycles), are still in the same ballpark as instrumentation-based checkpointing.

Our debugger allows the use of hardware breakpoints via a configuration option that is enabled by default. When this option is set, *DeLorean* will always try to convert a *soft-checkpoint request* into a *hard-checkpoint request*. This will only fail if the system runs out of available hardware breakpoints. Further, hardware breakpoints do not support conditional checkpoint requests other than equality checking.

C. Time-traveling Introspection

DeLorean provides `rollback` [b] and `restore` [c] commands to navigate the memory history. Travelling to a specific checkpoint is certainly useful when the number of checkpoints is relatively low. However, it is much harder to find the right target for millions of checkpoints.

To deal with a large checkpoint history, *DeLorean* provides two commands to *query* the set of previously taken checkpoints: `for` [d][e] and `search` [f]. Commands [d] and [e] linearly iterate over all the specified checkpoints and for each of them evaluate conditionals, and execute commands (either

conditionally or unconditionally). The user specifies the set of checkpoints as an interval (e.g., `(0,1000)` for the last 1000 checkpoints) a list of IDs, a mix of lists and intervals, or the keyword `all`, to include all the taken checkpoints.

The `search` [f] command, in turn, locates the first checkpoint satisfying a provided condition and rolls back to that state. The condition can be anything ranging from a simple comparison to the result of a function call. The command can be configured to either search strategies `forward`, `backward` and `bisect`. The `forward` and `backward` strategies perform a linear scan over the given checkpoint in opposite direction. Differently, the `bisect` strategy locates a checkpoint using binary search, similar to `git bisect` [34]. This strategy can dramatically reduce the duration of the search operation, although it is only applicable when the condition is met continuously from a particular point onward (e.g., from the point when `sanity_test()` returns `false`).

IV. Kernel Support

KDL, the *DeLorean* kernel module, is the core of our debugging system, providing all the necessary features to checkpoint a user process, roll it back to inspect past memory states, and restore the "present" memory state to resume execution. An overview of the module components and their interactions is depicted in Figure 2.

A. Taking Checkpoints

There are three entry points in *KDL* that cause a new checkpoint interval to begin. First, whenever a process registers with *KDL*, the initialization implicitly starts a checkpoint interval, as mentioned earlier. Second, it is possible to explicitly request a checkpoint via the *KDLCall* interface. This method is used by *dl* to implement targeted checkpoints on top of the `gdb` breakpoint mechanism. Finally, the kernel module triggers checkpoints directly using hardware breakpoints.

To efficiently satisfy checkpoint requests, *KDL* follows an incremental memory checkpointing strategy, shown as the best approach for high frequency memory checkpointing [35]. At the beginning of a new checkpoint interval, *KDL* write-protects all the memory pages to checkpoint. This strategy allows *KDL* to get notified whenever any of these pages is modified during a checkpoint interval to add a copy of the checkpointed page to the *checkpoint list* (i.e., using a *copy-on-write* strategy on a write *page fault*). This list is stored in the *process context* [3], a per-process data structure initialized whenever a process registers with *KDL*. As depicted in Figure 2, each *process context* also contains a *journal* of past checkpoints. Whenever the current checkpoint interval terminates (and a new interval starts), *KDL* re-protects all the dirty memory pages, i.e., pages with a copy in the current *checkpoint list*. Next, the current *checkpoint list* becomes an entry in the *journal* and is replaced by a new empty list. When the configurable journal size K is exhausted, *DeLorean* evicts the last checkpoint list from the journal to make room for new ones. Currently, our eviction strategy effectively deletes the oldest checkpointed pages, limiting the observable history.

[3] A *process context* refers to the memory associated to a given process.

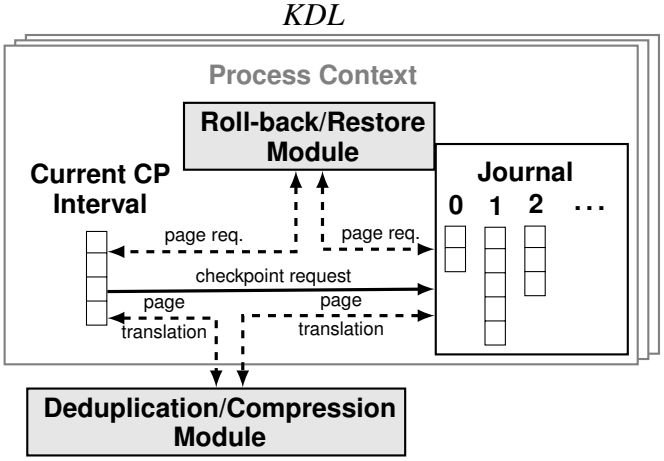


Fig. 2: Overview of the core *KDL* kernel module components.

B. Reducing Memory Overhead

As long-lived latent software bugs may trigger even hours after their root cause, they require a large checkpoint history to allow the user to locate the root cause of the bug. For this reason, our design seeks to retain as many checkpoints in the history as possible, trading off some run-time performance for a more space-efficient checkpoint list representation. Specifically, *KDL* supports optional *page deduplication* and *page compression* for all the checkpointed pages to reduce the memory footprint and scale to a larger number of checkpoints in the history.

a) Page Deduplication: Given that programs, and in particular server processes, tend to frequently re-initialize long lived data structures with similar data, we can expect many checkpointed pages to be filled with the same content. Building on this intuition, *KDL* supports a page deduplication strategy that keeps the checkpoints in the history as compact as possible. Figure 2 shows that the deduplication module (and its tracking data structures) has global (rather than per-process) scope. This enables deduplication of pages shared across processes concurrently registered with *KDL*.

To support page deduplication, *KDL* relies on a red-black binary tree to keep the set of globally unique checkpointed pages ordered. To determine the order between any two given pages, *KDL* supports two possible strategies: (i) directly comparing the content of the pages using `memcmp` [4] or (ii) compute and compare a *checksum* based on the content of the pages. The first approach requires a `memcmp` operation between a given page and $O(log(n))$ other pages (where n is the number of pages stored in the deduplication tree). The second approach requires only $O(log(n))$ checksum comparisons and, in the best case, only a single additional `memcmp` operation, but this number increases with the number of colliding pages, for which the checksum is the same while the content is not.

Furthermore, we explored two possibilities to select the moment to deduplicate a page (i.e., merge two checkpointed pages with the same content), based on either a *greedy* or a *lazy* approach. The greedy approach deduplicates each page at copy-on-write time, which eliminates unnecessary copying

[4] https://www.kernel.org/doc/htmldocs/kernel-api/API-memcmp.html

of duplicated pages. The lazy approach, in turn, deduplicates all the checkpointed pages at the end of every checkpoint interval. This approach does not eliminate unnecessary copying of checkpointed pages, but has the advantage of batching deduplication operations and removing expensive deduplication tree lookups in the page fault handler at copy-on-write time.

The effectiveness of page deduplication is heavily subject to the memory usage patterns of the debugged application. A very unstable working set[5] with random memory write patterns is unlikely to benefit from page deduplication. Luckily, real-world applications normally exhibit fairly stable working sets which benefit from checkpointed page deduplication, as the encouraging results in our evaluation demonstrate. In fact, Section VI shows that greedy checksum-based approach deduplication is the best candidate for our applications.

b) Page Compression: To improve space-efficiency guarantees with programs exhibiting poor temporal working set similarity, *KDL* also supports a page compression strategy. While page compression is more robust than deduplication against random memory write patterns and consequently more effective in reducing the memory footprint of the checkpoint history, the compression algorithm generally imposes a higher run-time overhead during checkpoint and roll-back operations. Furthermore, page compression requires *KDL* to store the compressed data in memory efficiently, or fragmentation would eliminate most of the savings gained. To address this problem, *KDL* stores the compressed pages using the `zsmalloc` allocator [36], designed for optimal storage of compressed memory.

Another key challenge concerns the selection of the compression algorithm to guarantee a good tradeoff between data compression and run-time performance. Inspired by `zram` [37], *KDL* relies on the LZO algorithm and deals with special-cases (such as zero-filled pages and pages whose compressed content is larger than the original content) so to implement an effective and efficient compression strategy.

Similar to deduplication, our page compression strategy can, in principle, operate using either a *greedy* or a *lazy* approach. Our current *KDL* implementation, however, only supports a *lazy* approach, given that, due to a caveat in the implementation of `zsmalloc`, it is not possible to integrate compression support in our page-fault handling code path.

Finally, while page compression and deduplication are conceptually independent (and competing) alternatives, *KDL* can support both deduplication and compression at the same time. We evaluate all the possible deduplication/compression configurations of *KDL* in Section VI.

C. Rolling Back to a Checkpoint

The *KDL* rollback component is responsible for rolling back the memory state of the target process to a given checkpoint. As depicted in Figure 2, this component operates directly in the target process context. To implement the required functionalities, the rollback component identifies all the pages that have to be restored in order to rollback to a given checkpoint and stores them in a binary search tree—the *roll-back tree*. Furthermore, it replaces all the page mappings of that target process managed memory regions with the ones that are part of the specified checkpoint (by looking them up in the rollback tree created earlier). To protect the checkpointed pages, the new mappings are initially all read-only. Should the target process want to write into an initially writeable page after rollback, *DeLorean* lazily creates a scratch copy of the page in question using a copy-on-write strategy. When the "current" process state is restored, *KDL* discards all the scratch pages created by any user operations while rolled-back.

For efficiency reasons, the rollback tree usually only contains entries for one specific checkpoint. *KDL* creates the tree by looking up the checkpoint list for the target checkpoint and adding all its pages to the rollback tree. Furthermore, *KDL* iterates through the younger journal entries to add all the pages referring to memory locations not already included in the tree but mapped at the time the checkpoint was taken.

Given that generating the rollback tree is expensive, *KDL* also supports the generation of a permanent and global version of the rollback tree, which includes data for all the checkpoints. The initial cost to create a permanent tree is higher, but the cost is amortized when a large number of rollback operations are requested, e.g., when the user performs a search over a large number of checkpoints.

The rollback approach described thus far is the most transparent and general possible: *KDL* reverts the *entire* memory state to a given checkpoint and the user may freely inspect the old state and implement arbitrarily complex operations over it. However, this approach may not be the most efficient if user operations over the checkpointed state are localized (e.g., a simple search over a given global variable), given that an excessive number of pages may need to be mapped (and remapped back). To address this problem, *KDL* supports an alternative. Whenever the userland is aware of the variables that need to be accessed beforehand, *KDL* supports an *on-demand* rollback mechanism. This approach only rolls back the pages that are actually required to access the target data structures. To implement this mechanism, *KDL* exports a user-level API which requires a list of memory addresses and the corresponding sizes (the targeted memory pages are identified automatically by *KDL*).

V. Implementation

We implemented our *DeLorean* prototype on Linux. *KDL* is implemented as a standard Linux *loadable kernel module* supporting recent kernel versions (3.2 until 3.19), allowing for easy deployability. The module is compatible with the `x86` and `x86_64` architectures. The total module size is $5,493$ LOC[6], including all the modes of operation described in the paper.

To keep *KDL* as portable as possible, we relied on *Kernel Probes* [38] (the standard Linux instrumentation framework to non-disruptively hook into kernel routines) to implement two key features: (i) page fault handling (and copy-on-write) interposition by hooking into `handle_mm_fault` before a write page fault is handled; (ii) process exit interposition to clean up each process context by hooking into `do_exit` and `do_execve`. The *KDLCall* system call interface, in turn, is implemented using the Linux `sysctl` [39] mechanism,

[5] A *working set* refers to the memory pages actively being used by a process during the execution.

[6] Source lines of code as reported by the CLOC tool.

the standard Linux interface to configure kernel parameters at runtime. This interface is used by the userland to issue checkpoint/roll-back/restore requests, retrieve module statistics, and access configuration options.

The *dl* debugging component is implemented as a combination of a Python-based `gdb` plugin [40] and a `gdb` command file [41], accounting for a total of $1,075$ LOC. Other than implementing *dl*-specific options and commands, the scripts add hooks to allow for proper cleanup when the target process exits and to force a restore if the user resumes execution while the process is in rolled-back state. As *dl* is a `gdb` extension, our debugger targets programs developed in languages supported by `gdb`[7]. The *LibDL* shared library, finally, implemented in 417 lines of C code, is injected into the target process via the standard `LD_PRELOAD` interface.

VI. Evaluation

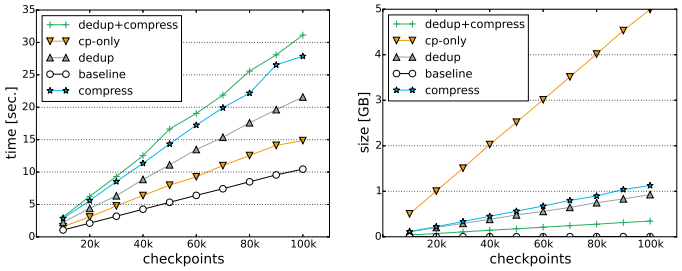
We evaluated *DeLorean* on an Ubuntu 14.04 workstation running the stock kernel Linux v3.16 (x86_64), with a dual-core Intel Pentium G6950 2.80GHz processor and 16GB of RAM. To evaluate the impact of our system on real-world applications, we selected five popular server programs, similarly to prior work in memory checkpointing and record/replay debugging [25], [10], [8], [9], [17], [27], [22], [23], [15]. In particular, we focused our evaluation on three popular open-source web servers, Apache `httpd` [42] (v2.2.23), `nginx` [43] (v0.8.54) and `lighttpd` [44] (v1.4.28), a widely used database server, PostgreSQL [45] (v9.0.10), and the vastly deployed `bind` DNS server [46] (v9.9.3).

We evaluated the impact of *DeLorean* on our server programs using well-known benchmark suites: Apache benchmark (`ab`) [47] for our web servers, SysBench [48] for PostgreSQL, and queryperf [49] for `bind`. To evaluate the worst-case performance with high CPU pressure, we run our performance tests issuing requests through the loopback device (also a common latent bug debugging scenario), which fully saturated our test programs. In addition, we evaluated additional aspects of our system using dedicated microbenchmarks.

We configured our programs and benchmarks with their default options. We instrumented our programs with our *LibDL* shared library and allowed them to take a checkpoint for each request (the finest granularity to debug latent event-driven bugs). Due to the lack of RAM in our setup, we interpolated (without loss of accuracy) the checkpointing results obtained with no footprint reduction method (i.e., compression or deduplication). We repeated our experiments 11 times (with negligible variations) and report the median values. We summarize normalized benchmark results using the geometric mean [50].

Our evaluation answers the following questions: (a) *Deduplication and compression performance*: How efficient are the various deduplication and compression strategies supported by *DeLorean*? (b) *Deduplication and compression effectiveness*: How effective are the various deduplication and compression strategies supported by *DeLorean* in reducing the memory footprint and scaling to a large checkpoint history? (c) *Time–travelling introspection performance*: How efficiently can we query the checkpointed memory pages through the history?

[7]See https://goo.gl/MTfdlY for a list of supported languages.



(a) runtime (b) memory usage

Fig. 6: Impact of deduplication and compression vs. number of checkpoints (`lighttpd`).

(d) *Comparison with existing solutions*: How do our results compare to existing debugging solutions? (e) *Case studies*: How can we use *DeLorean* to debug real-world software bugs?

A. Deduplication Performance

Deduplicating a page at record time can be done by directly *comparing* any two given pages, or by first computing (and comparing) a *checksum* of the two pages. Both solutions incur two costs: (i) maintaining a data structure to store the unique instance of each page and (ii) actually checking for duplicate pages. The distribution of such costs depends on whether we opt for a *greedy* (at page fault handling-time) or a *lazy* (at checkpoint finalization-time) deduplication strategy.

Figure 3 presents the throughput of each deduplication method compared to the baseline. The baseline, namely programs running without checkpointing, resulted in an average of $12k$ requests per second (geometric mean). Applying plain checkpointing (*cp-only*) resulted in 39 % throughput degradation, compared to 78.2 % and 77.7 % when applying comparison-based deduplication (*page-greedy* and *page-lazy*, respectively), and 65.8 % and 64 % when applying checksum-based deduplication (*crc-greedy* and *crc-lazy*). This demonstrates that page comparison costs are higher than checksumming costs. Furthermore, the greedy approach performs slightly better than the lazy approach independently of the comparison strategy adopted. This result is in line with page deduplication significantly reducing the number of checkpointed pages and the cost of greedily copying pages being thus amortized by the benefits of maintaining less unique page instances on tracking data structures.

This trend is reversed for the comparison-based strategy on `nginx`, which causes *page-greedy* to produce a higher throughput degradation compared to *page-lazy*. A closer inspection revealed that, while both comparison-based strategies are less effective in deduplicating pages compared to the checksum-based strategies, *page-greedy* deduplicates even less than *page-lazy* on `nginx`. We attribute this behaviour to the run-time overhead imposed by page deduplication, which slows down the server and causes it to behave less deterministically with additional data accounting. This, in turn, results in reduced deduplication effectiveness, further amplified by the *page-greedy* strategy, which introduces extra page fault-time latency throughout the execution.

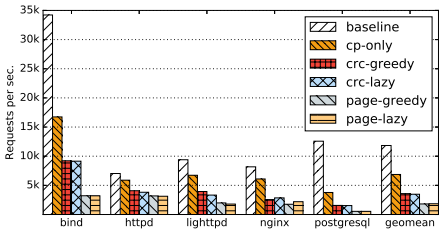


Fig. 3: Server throughput for our dedup. strategies ($1M$ checkpoints). `cp-only` results are interpolated.

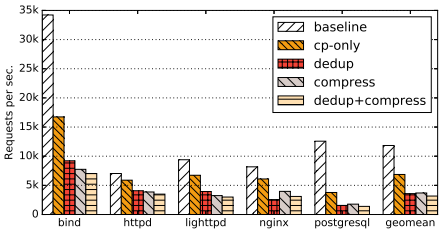


Fig. 4: Server throughput with compression and/or dedup. ($1M$ checkpoints). `cp-only` results are interpolated.

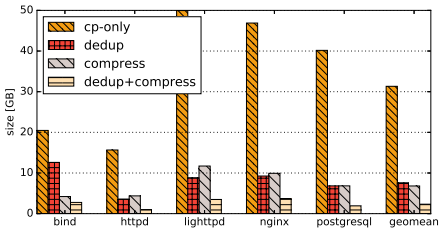
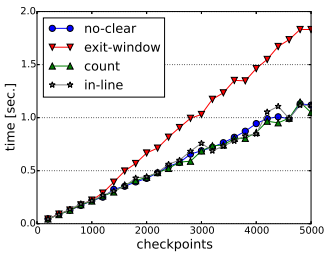
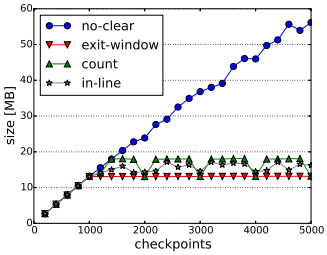


Fig. 5: Memory footprint with compression and/or dedup. ($1M$ checkpoints). `cp-only` results are interpolated.



(a) runtime overhead (crc-based).



(b) size overhead (crc-based).

Fig. 7: Impact of orphan cleanup mechanisms vs. number of checkpoints (`lighttpd`).

While *crc-greedy* performs sliglhtly better than *crc-lazy*, we will focus on *crc-lazy* deduplication (or simply *dedup*, hereafter). Due to implementation limitations, *crc-greedy* can only be combined with compression by uncompressing all the checkpointed pages to determine duplicates, while *crc-lazy* can deduplicate based on compressed data.

B. Orphans Cleanup Impact

Deduplication requires maintaining a dedicated tracking data structure (the deduplication tree) and relinquishing pages that are no longer needed. In particular, when all the references to a given deduplicated page are dropped (e.g., when a particular checkpoint list is deleted), the corresponding now *orphaned* page remains in the deduplication tree. Without further intervention, the number of orphan pages would grow over time, resulting in increasing performance overhead (i.e., the cost to walk the deduplication tree increases) and memory overhead (i.e., the number of tracked pages increases). To address this problem, we implemented *orphan cleanup* mechanisms that triggered at various points during the checkpointing process.

Figure 7 shows the effect on the benchmark run time and memory usage incurred by such mechanisms on `lighttpd` (for clarity, we omit the very similar results on the other servers). For this experiment, we set the journal size to $1,000$ checkpoints and linearly increased the number of checkpoints taken. As shown in the figure, the naive no-op (*cp-only*) mechanism rapidly increases the increases the memory footprint of the system. Cleaning up orphans every time a checkpoint is deleted (*exit-window* mechanism), in turn, does not impose additional memory overhead, but imposes a higher performance overhead. Performing orphan cleanup operations after taking a certain number of checkpoints (*count*, i.e., every $1,000$ checkpoints in our case) or at deduplication time (*inline*) reduces the impact of orphans cleanup, while retaining reasonable memory usage guarantees. To isolate the effect of orphan deletion, the rest of our analysis assumes that the journal size coincides with the observation window.

C. Deduplication vs. Compression

While deduplication is an effective solution to reduce memory usage at record time in the common case, compression represents a more general (but slower) alternative. *DeLorean* can also support both deduplication and compression to minimize the overall memory footprint. Figure 6 illustrates the impact of different deduplication/compression configurations for an increasing number of checkpoints on `lighttpd`. As expected, both the benchmark run time (Figure 6a) and memory usage (Figure 6b) increase linearly with the number of checkpoints taken.

c) Memory Overhead: To measure the effectiveness of deduplication (*dedup*) and compression (*compress*), we measured the amount of memory (the page data and the metadata required for accounting) used by each approach and compared the results with the baseline, namely checkpointing without any memory reduction strategy (*cp-only*).

Figure 5 shows that most of our server programs' write patterns allow for effective deduplication of the checkpoint history. Plain checkpointing yields substantial memory impact (up to $50GB$), while deduplication reduces the memory footprint up to 84 % (`nginx`). For `bind`'s less stable working set, deduplication is less effective, resulting in only 38 % footprint reduction. Separately, compression and deduplication reported on average similar reduction across our server programs. Finally, The combination of both strategies (*dedup+compress* is the most effective method, allowing for memory footprint reductions of up to 94 % (`postgresql`).

d) Run-time Overhead: While reducing the memory footprint, enabling deduplication and/or compression also increases the run-time overhead. Figure 4 shows the throughput (requests per second) reported by our server programs when enabling deduplication, compression, or both.

Starting from an average of $12k$ requests per second for the baseline (no checkpointing), throughput degradation induced by plain checkpointing ranges between 16 % (`httpd`) and 70 % (`postgresql`) with an average of 33 % (geometric mean). Server programs with shorter request-processing loops yield a higher checkpointing frequency, naturally increasing

Test	Linear [sec]			Binary [sec]		
	$10k$	$100k$	$1M$	$10k$	$100k$	$1M$
Full (compression)	0.960	9.851	99.065	0.061	0.724	8.544
Full	0.155	1.728	18.655	0.061	0.724	8.605
On-demand	0.088	0.999	11.414	0.061	0.739	8.592

TABLE II: Search tests with 13 pages working sets.

the performance impact. Applying compression or deduplication separately imposes a similar throughput degradation on the server programs (63.19 % vs. 63.96 %, geometric mean). When combining both approaches, the cost of deduplication and compression are amortized, resulting in an average overall degradation of 68 %. Since the combination of both approaches incurs only slightly higher overhead (+4.5 % compared to deduplication or compression, geometric mean) while enabling the most space-efficient (and scalable) checkpointing strategy, *DeLorean* enables both memory saving techniques by default. However, considering the significant difference with plain checkpointing (+35 %), *DeLorean* allows the user to disable the space-efficent methods for better runtime performance.

D. Time-travelling Introspection Performance

In this section, we evaluate the performance of *DeLorean*'s *time travelling introspection* strategy, which relies on efficient queries over the checkpoint history accumulated during execution. Table II reports the time to complete a query through the full checkpoint history with an average of 13 pages per checkpoint—corresponding to the highest average checkpoint size observed across our server programs during the execution of our benchmarks. For this experiment, we selected a simple query condition comparing the content of a known integer global variable against an expected value. Moreover, we filled the checkpoint history with 13 pages per checkpoint. We employed deduplication to make sure that the pages would fit in memory for $1M$ checkpoints. However, this does not impact the rollback performance, hence the results of the experiment proposed. We ran this microbenchmark 11 times (with negligible variations) and report the median.

To thoroughly evaluate the query run time, we have analysed the impact of different factors: (a) the rollback mechanisms, namely the *full* mechanism (rolling back all the pages), and the *on-demand* mechanism (rolling back only the target page); (b) the query strategy, either *linear* or *bisect*, (c) and whether compression is used.

As expected, Table II shows that the query run time linearly increases with the number of checkpoints independently of the particular factors considered. We now examine the impact of each factor in more detail.

e) Rollback mechanism: The *full* rollback mechanism seeks to support arbitrary search queries, but it also requires building the permanent rollback tree first and issuing a full rollback operation for each condition evaluation. Table II depicts the impact of these costs on the query run time. For example, for $1M$ checkpoints, the query takes approximately 99 seconds to inspect all the 13 million pages in the checkpoint history (*linear* query strategy). While this is relatively efficient, the other rollback mechanisms provide better performance. Finally, the *on-demand* rollback mechanism, applicable to our example test case since the location and the size of the inspected variables are known (e.g., no pointers are used), reduces the time to process all the checkpoints by around 88%.

f) Query strategy: Compared to the *linear*-based strategy, our *bisect* strategy significantly improves query performance. Bisecting only requires to search through $O(log(n))$ checkpoints (where n is the size of the checkpoint history). Table II shows that, while much faster, the *bisect* performance is still bottlenecked by the constant cost of building the permanent rollback tree (around 8 seconds for $1M$ checkpoints).

g) Compression: Unlike deduplication, the use of compression has a negative impact on query performance, given that every checkpoint examined during the search operations needs to be decompressed at rollback time. Table II depicts this cost when using the `full` rollback mechanism (providing worst-case results). As shown in the table, a query becomes approximately 5.5 times slower when using compression. When bisecting is possible, however, the same cost is negligible.

E. Comparison with Existing Solutions

We compared our checkpointing and introspection overhead to two other systems that offer comparable (and state-of-the-art) functionality: the `gdb` general-purpose debugger and the `rr` record-and replay debugger [51].

We implemented checkpointing-based introspection, namely linear and bisect search, on top of `gdb`'s own fork-based checkpointing as a `gdb` extension. We configured `rr`'s replay strategy to re-execute the recorded trace and perform search using a conditional breakpoint set at the point where *DeLorean* would take a checkpoint. *DeLorean* itself is configured to use compression and deduplication. We report the results of a synthetic experiment on `lighttpd` (but we observed similar results on the other server programs), which we exercised with the Apache Benchmark (`ab`) issuing 10,000 requests. We used this number of requests since `gdb` could not scale to a much larger checkpoint history, but we remark that our results can linearly extend to millions of requests. Table III shows our results.

Compared to 1.1 seconds execution time without recording, *DeLorean* has the smallest recording overhead (3 seconds). The runner-up is `rr`, whose recording time is roughly 3.5 times slower than *DeLorean* (11 seconds) and approximately 11 times slower than the baseline. `gdb` recording time reported the worst recording performance (464 seconds), which translates to a 422x slowdown. `gdb` also consumed the largest amount of memory (over 2 GB), followed by *DeLorean* (42.4 MB). `rr` reported the smallest memory footprint (3.6 MB). As for search performance, when linearly searching through the checkpoints in the worst case scenario—search for an expression that is never satisfied—`gdb` takes the longest time (590 seconds), while `rr` and *DeLorean* are in the same ballpark (30 and 28 seconds, respectively). `rr`'s design cannot, however, support the more efficient bisect search, which allows both gdb and *DeLorean* to complete the example query in less than a second, with a ∼30x speedup in the case of *DeLorean*.

In summary, `psystem` performs much better than `gdb` with smaller space requirements. Compared to `rr`, *DeLorean* has better recording performance (as a result of trading off on space usage), comparable search performance when not bisecting, and much better search performance when bisecting.

	Recording		Introspection	
	time [sec]	space [MB]	linear [sec]	binary [sec]
DeLorean	3.2	42.4	30.2	0.2
GDB	464.1	2,298.0	590.1	0.9
RR	11.1	3.6	28.1	−−

TABLE III: Comparison of DeLorean, RR, and GDB on `lighttpd` for $10k$ server requests.

We also compared our results with Scribe [10], a transparent low-overhead record-and-replay system. For practicality reasons, we used the virtual machine image offered for download by its authors (thus reporting even optimistic overheads for Scribe, given the generally slower baseline). The virtual machine was equipped with Ubuntu 12.04 and the custom Scribe Linux kernel v2.6.35. Due to kernel failures, we were only able to test two of our servers, namely `lighttpd` and Apache `httpd`. While the memory used to log the events of the two servers was relatively small ($83MB$ and $90MB$, respectively), we reported significant run-time overhead. We measured 682.74% run-time overhead for `lighttpd` and 255.31% for Apache httpd. *DeLorean* in a similar configuration—same virtualization techniques, but newer kernel—only reported roughly half the of the overhead when run in a virtual machine on the same physical machine.

F. Case Studies

To demonstrate the effectiveness of *DeLorean*, we show two case studies where *DeLorean* can prove useful in debugging arbitrary memory corruption and memory leaks.

1) Arbitrary Memory Corruption: an infamous bug in the FFmpeg library [52], which drives popular servers such as FFserver, causes erroneous handling of 4X videos under particular inputs and results in wild writes to random memory locations [53]. Servers that use the buggy FFmpeg library are susceptible to arbitrary state corruption when receiving particular requests. For example, a request-triggered wild memory write in FFserver can corrupt one of the `FFServerStreams` contained in `FFserverConfig` used to handle client feeds information. Since such streams are stored as a linked list, the `next` pointer referencing the next entry in the list may be overwritten with, say, `0xDEADBEEF`. The program may only crash several requests later, when the corrupted pointer is used while processing a completely unrelated event. Reproducing the crash alone does not necessarily help the user to identify the root cause, since the crash and trigger event are temporally distant, making difficult to relate the pointer corruption to the crash. This course of action generalizes to other memory corruption bugs, e.g., bind's CVE-2015-8000[8], nginx's CVE-2016-4450[9], and others.

With *DeLorean*, the user can configure the debugger to take a checkpoint for each server request. This is possible by issuing the command `dl checkpoint take` and specifying the end of the server request loop as a target location. The user then allows the program to continue the execution and activate the test workload. Once the server crashes, *DeLorean* allows the user to efficiently query all the checkpoints taken during the execution in search of the trigger event.

[8]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8000

[9]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4450

In this example, we search for the first checkpoint containing the pointer with the corrupted value. This query can be performed by means of bisection. For instance, if the corrupted pointer contains the value `0xDEADBEEF`), the developer can issue `dl search stream->next == 0xDEADBEEF` search command.

If successful, the `search` command automatically rolls the program state back to the first checkpoint that satisfies the given condition, or the present state otherwise. After restoration, the user can check for irregularities in the trigger request and the program state to identify the root cause quickly.

2) Memory Leak: Some applications (e.g., PostgreSQL, the JVM, the Python interpreter, and others) implement garbage collection using techniques like reference counting or context management. In the case of PostgreSQL, most of the backend operations rely on a context which keeps track of allocated memory and, each time a context is freed, so is all the referenced memory in that context. As a result, mismanagement of contexts can cause memory leaks. A typical bug is for memory allocated for each request to be incorrectly assigned to a long-lived context which is not freed until the server is reloaded. In such a situation, the server will slowly go out of memory (after a certain number of requests), making it difficult for users to determine the root cause of the bug.

With *DeLorean*, a user can inspect the context and, using either `dl search` or `dl for`, can determine the number of referenced objects as well as their location in memory. Using this strategy, the cause of a memory leak becomes quickly apparent. Similarly, in the case of other systems employing reference counting, *DeLorean* can allow users to determine where and when a reference cycle was introduced by analyzing counter management data structures for a target object.

VII. Discussion

Firstly, *DeLorean* is mainly aimed at aiding debugging for applications that have a medium-sized impact in terms of memory footprint (i.e., server applications). However, applications with bigger memory footprint can also be debugged while inevitably increasing the memory requirements for storing a high number of checkpoints.

Additionally, since *DeLorean* makes a significant use of the features and the environment provided by `gdb`, we share the same limitations that already affect `gdb`. This is also the case for the applicability of the tool.

While our efforts have focused on improving the memory footprint of our checkpoint-based system, we also aim at trading as little run-time performance as possible. For this reason, we explored the possibility to improve performance by extending *DeLorean* with *working set estimation* (WSE) strategies. WSE [33], [54] can help predict pages that are part of the writable working set in the next checkpoint interval, hence applying a WSE-driven precopying strategy rather than a fully copy-on-write strategy may reduce page fault handling costs [35]. However, when applying WSE to our *DeLorean* prototype we have observed little improvement. We attribute this outcome to the significant overhead imposed by the deduplication, which, when enabled, overshadows the benefits introduced by WSE. To improve the effectiveness of WSE and

the overall system performance, an option is to consider an asynchronous deduplication and compression strategy. However, asynchronously processing the checkpointed data also introduces other concerns. In particular, if the speed with which new pages are copied is higher than that of asynchronous page processing operations, the memory reduction may become insufficient to support a large checkpoint history.

Another aspect that we did not explore in *DeLorean* is the effect of swapping excessive checkpoint data to the disk, rather than simply evicting them from memory. This strategy can allow users to maintain even larger checkpoint histories, but the impact on query performance may be significant.

Finally, while we consider *DeLorean* on its own a powerful debugging tool, we do not necessarily perceive checkpoint-assisted time-travelling introspection as an alternative to record-replay-based systems. We instead see potential in combining other debugging systems with *DeLorean* introspection capabilities, for example to assist the user in identifying the root cause of a bug after a record-and-replay system has helped her reproduce the bug on a lengthy test workload.

VIII. Related Work

a) Record/Replay systems: Research in record/replay techniques has led to a broad range of software-based [55], [4], [56], [8], [10], [57] and hardware-based [58], [59], [3], [7] solutions, which improve debuggability for both operating systems [4], [56] and user applications [8], [10], [57], [3]. Nonetheless, the capabilities and focus of these systems vary considerably, as some focus on replay speed [4], [3], [60], others on live execution after replay [10], and yet others on supporting replay sessions on modified versions of the debugged applications [8]. While hardware-based solutions are appealing, dedicated hardware is not usually available on commodity systems. Software-based solutions are more convenient, but incur much higher overheads on recording and replaying. As a trade-off, OS-level solutions [8], [10], [57] and VMM-level solutions [56], [4] yield better performance, although they still incur significant record-time overheads. *DeLorean* does not offer replay capabilities but focuses on providing *better* debugging instead—with support for queries over the execution history for root-cause analysis—and explores high-speed checkpointing as a convenient and efficient way to collect and store such history. Moreover, our approach is orthogonal to record/replay systems, since checkpointing can be employed during the replay phase to add search functionalities in off-line debugging sessions.

b) Checkpointing: Over the past 15 years, the research community has proposed many checkpointing solutions. User-level implementations [61], [62], [17], [63], [23], [64], [24], [9] are generally easier to deploy, but they also typically incur significant run-time overhead, making them ill-suited for high-frequency memory checkpointing. Compiler-based solutions [15] are more efficient, but require recompilation, reducing deployability. Other approaches improve performance by adding checkpointing support at the VMM level [4], [30], [65], [14]. Unfortunately, this method requires checkpointing the whole virtual machine rather than specific applications of interest. Approaches using dedicated hardware [66], [67] are more efficient but impractical for commodity systems. Finally, kernel-level checkpointing solutions have been explored before [68], [69], [70], [71], [72], [25], [73]. Unlike *DeLorean*, however, they either rely on kernel patches [68], [69], [71] compromising their general applicability, or operate as kernel modules that implement full-blown memory containers [72], [25]. The recent SMC [35] implements a more deployable solution in a self-contained kernel module, but tends to generate duplicates—trading memory usage for better performance—resulting in a nonscalable solution when maintaining a large checkpoint history. *DeLorean*, in contrast, relies on page deduplication and compression to store a large history, with better scalability compared to regular checkpointing solutions.

c) Deduplication: Deduplication is a common strategy to reduce memory usage in virtualized environments [74]. Prior work on disk-based checkpointing also used deduplication to reduce the size of on-disk checkpoints [75], [76], [65]. *DeLorean* follows a similar intuition to reduce the footprint of in-memory checkpoints for debugging. Beside deduplicating across applications, *DeLorean* also deduplicates pages across checkpoints for a given process, exploiting the intuition that programs often exhibit recurring steady states and thus yield a growing number of duplicated pages in the checkpoint history.

d) Compression: Page compression is another common technique to reduce memory usage [77], [37], [78]. Prior work in the area, however, is mostly concerned with reducing the impact of I/O operations and improving system, rather than debugging, performance. Similar to deduplication, compression has also been used in prior disk-based checkpointing solutions to improve disk space utilization [79], [80], [81]. *DeLorean*, in contrast, draws inspiration from memory compression solutions such as `zram` [37], to aggressively compress checkpointed data in memory and support a very large checkpoint history despite deduplication low effectiveness. Finally, in contrast to previous work in the general area of disk-based checkpointing, *DeLorean* combines compression and deduplication to further reduce its memory footprint.

IX. Conclusion

DeLorean is a checkpoint-assisted time-travelling debugging system which trades replay capabilities for more efficient debugging and root-cause analysis. Its high-frequency checkpointing strategy reduces the run-time overhead of the recording phase, while minimizing the footprint of checkpointed memory by means of page deduplication and compression. We further proposed a new time-travelling introspection mechanism that iterates over the checkpointed memory to pinpoint the root cause of long-lived event-driven software bugs. Our evaluation shows that checkpoint-assisted time-travelling introspection is efficient in quickly searching over the checkpoint history for a large window of observation.

X. Acknowledgments

We thank the anonymous reviewers for their feedback. This work was supported by the European Commission through project H2020 ICT-32-2014 “SHARCS” under Grant Agreement No. 64457 and the Netherlands Organisation for Scientific Research through grant 639.023.309 VICI “Dowsing”.

REFERENCES

[1] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea, "Failure sketching: a technique for automated root cause diagnosis of in-production failures," in *Proceedings of the Symposium on Operating Systems Principles*, 2015.

[2] C. Zamfir and G. Candea, "Execution synthesis: A technique for automated software debugging," in *Proceedings of the European Conference on Computer Systems*, 2010.

[3] N. Honarmand, N. Dautenhahn, J. Torrellas, S. T. King, G. Pokam, and C. Pereira, "Cyrus: Unintrusive application-level record-replay for replay parallelism," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[4] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging operating systems with time-traveling virtual machines," in *Proceedings of the USENIX Annual Technical Conference*, 2005.

[5] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "Pres: Probabilistic replay with execution sketching on multiprocessors," in *Proceedings of the Symposium on Operating Systems Principles*, 2009.

[6] G. Altekar and I. Stoica, "Odr: Output-deterministic replay for multicore debugging," in *Proceedings of the Symposium on Operating Systems Principles*, 2009.

[7] A. Basu, J. Bobba, and M. D. Hill, "Karma: Scalable deterministic record-replay," in *Proceedings of the International Conference on Supercomputing*, 2011.

[8] N. Viennot, S. Nair, and J. Nieh, "Transparent mutable replay for multicore debugging and patch validation," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[9] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, "Flashback: A lightweight extension for rollback and deterministic replay for software debugging," in *Proceedings of the USENIX Annual Technical Conference*, 2004.

[10] O. Laadan, N. Viennot, and J. Nieh, "Transparent, lightweight application execution replay on commodity multiprocessor operating systems," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, 2010.

[11] "GDB," http://www.gnu.org/software/gdb/, 2015.

[12] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 2008.

[13] P. Lu, B. Ravindran, and C. Kim, "Vpc: Scalable, low downtime checkpointing for virtual clusters," in *Proceedings of the International Symposium on Computer Architecture and High Performance Computing*, 2012.

[14] L. Wang, Z. Kalbarczyk, R. K. Iyer, and A. Iyengar, "Checkpointing virtual machines against transient errors," in *Proceedings of the International On-Line Testing Symposium*, 2010.

[15] D. Vogt, C. Giuffrida, H. Bos, and A. Tanenbaum, "Lightweight memory checkpointing," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2015.

[16] D. Vogt, C. Giuffrida, H. Bos, and A. S. Tanenbaum, "Techniques for efficient in-memory checkpointing," in *Proceedings of the Workshop on Hot Topics in Dependable Systems*, 2013.

[17] Q. Gao, W. Zhang, Y. Tang, and F. Qin, "First-aid: surviving and preventing memory management bugs during production runs," in *Proceedings of the European Conference on Computer Systems*, 2009.

[18] C. Giuffrida, L. Cavallaro, and A. S. Tanenbaum, "We crashed, now what?" in *Proceedings of the International Conference on Hot Topics in System Dependability*, 2010.

[19] C. Giuffrida, C. Iorgulescu, and A. S. Tanenbaum, "Mutable checkpoint-restart: automating live update for generic server programs," in *Proceedings of the International Middleware Conference*, 2014.

[20] A. Kadav, M. J. Renzelmann, and M. M. Swift, "Fine-grained fault tolerance using device checkpoints," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[21] A. Lenharth, V. S. Adve, and S. T. King, "Recovery domains: an organizing principle for recoverable operating systems," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[22] G. Portokalidis and A. D. Keromytis, "REASSURE: a self-contained mechanism for healing software using rescue points," in *Proceedings of the International Conference on Advances in Information and Computer Security*, 2011.

[23] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: treating bugs as allergies—a safe method to survive software failures," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, 2005.

[24] J. F. Ruscio, M. Heffner, S. Varadarajan *et al.*, "Dejavu: Transparent user-level checkpointing, migration, and recovery for distributed systems," in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2007.

[25] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis, "ASSURE: automatic software self-healing using rescue points," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[26] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou, "Triage: diagnosing production run failures at the user's site," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, 2007.

[27] A. Zavou, G. Portokalidis, and A. D. Keromytis, "Self-healing multitier architectures using cascading rescue points," in *Proceedings of the Annual Computer Security Applications Conference*, 2012.

[28] K. Bhat, D. Vogt, E. van der Kouwe, B. Gras, L. Sambuc, A. S. Tanenbaum, H. Bos, and C. Giuffrida, "OSIRIS: Efficient and consistent recovery of compartmentalized operating systems," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2016.

[29] J. Hursey, C. January, M. O'Connor, P. H. Hargrove, D. Lecomber, J. M. Squyres, and A. Lumsdaine, "Checkpoint/restart-enabled parallel debugging," in *Proceedings of the European Conference on Recent Advances in the Message Passing Interface*, 2010.

[30] E. Bugnion, V. Chipounov, and G. Candea, "Lightweight snapshots and system-level backtracking," in *Proceedings of the Workshop on Hot Topics on Operating Systems*, 2013.

[31] C. C. Zhao, J. G. Steffan, C. Amza, and A. Kielstra, "Compiler support for fine-grain software-only checkpointing," in *Proceedings of the International Conference on Compiler Construction*, 2012.

[32] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala, "Checkpointing and its applications," in *Proceedings of the International Symposium on Fault-Tolerant Computing*, 1995.

[33] I. Zhang, T. Denniston, Y. Baskakov, and A. Garthwaite, "Optimizing vm checkpointing for restore performance in vmware esxi." in *Proceedings of the USENIX Annual Technical Conference*, 2013.

[34] "git-bisect manual page," https://www.kernel.org/pub/software/scm/git/docs/v1.7.10/git-bisect.html, 2015.

[35] D. Vogt, A. Miraglia, G. Portokalidis, H. Bos, A. S. Tanenbaum, and C. Giuffrida, "Speculative memory checkpointing," in *Proceedings of the ACM/IFIP/USENIX Middleware Conference*, 2015.

[36] "The zsmalloc allocator," https://lwn.net/Articles/477067/, 2012.

[37] D. Magenheimer, "In-kernel memory compression," https://lwn.net/Articles/545244/, 2013.

[38] "Kernel probes," https://www.kernel.org/doc/Documentation/kprobes.txt, 2015.

[39] "/proc/sys documentation," https://www.kernel.org/doc/Documentation/sysctl/README, 1998.

[40] "GDB python API," https://sourceware.org/gdb/onlinedocs/gdb/Python-API.html, 2015.

[41] "GDB command files," https://sourceware.org/gdb/onlinedocs/gdb/Command-Files.html#Command-Files, 2015.

[42] "Apache HTTP server," http://httpd.apache.org/, 2015.

[43] "nginx," http://nginx.org/, 2010.

[44] "lighttpd," http://www.lighttpd.net/, 2009.

[45] "PostgreSQL," http://www.postgresql.org/, 2012.

[46] "BIND," https://www.isc.org/downloads/bind/, 2013.

[47] "Apache benchmark," http://httpd.apache.org/docs/2.2/programs/ab.html, 2015.

[48] "SysBench," https://github.com/akopytov/sysbench, 2015.

[49] "queryperf," https://github.com/davidmmiller/bind9/tree/master/contrib/queryperf, 2001.

[50] P. J. Fleming and J. J. Wallace, "How not to lie with statistics: The correct way to summarize benchmark results," *Commun. ACM*, vol. 29, no. 3, pp. 218–221, Mar. 1986.

[51] "rr: lightweight recording and deterministic debugging," http://rr-project.org/, 2015.

[52] "FFmpeg," https://www.ffmpeg.org/, 2015.

[53] "CVE 2009-0385," http://www.trapkit.de/advisories/TKADV2009-004.txt, 2009.

[54] I. Zhang, A. Garthwaite, Y. Baskakov, and K. C. Barr, "Fast restore of checkpointed memory using working set estimation," in *Proceedings of the International Conference on Virtual Execution Environments*, 2011.

[55] Y. Saito, "Jockey: a user-space library for record-replay debugging," in *Proceedings of the International Symposium on Automated Analysis-driven Debugging*, 2005.

[56] J. Chow, D. Lucchetti, T. Garfinkel, G. Lefebvre, R. Gardner, J. Mason, S. Small, and P. M. Chen, "Multi-stage replay with crosscut," in *Proceedings of the International Conference on Virtual Execution Environments*, 2010.

[57] H. Thane and H. Hansson, "Using deterministic replay for debugging of distributed real-time systems," in *Proceedings of the Euromicro Conference on Real-Time Systems*, 2000.

[58] P. Montesinos, L. Ceze, and J. Torrellas, "Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently," in *Proceedings of the International Symposium on Computer Architecture*, 2008.

[59] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas, "Capo: a software-hardware interface for practical deterministic multiprocessor replay," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[60] S. Bhansali, W.-K. Chen, S. De Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau, "Framework for instruction-level tracing and analysis of program executions," in *Proceedings of the International Conference on Virtual Execution Environments*, 2006.

[61] "CRIU," http://criu.org/, 2015.

[62] W. R. Dieter and L. J. E., "User-level checkpointing for LinuxThreads programs," in *Proceedings of the USENIX Annual Technical Conference*, 2001.

[63] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under UNIX," in *Proceedings of the USENIX Annual Technical Conference*, 1995.

[64] M. Rieker, J. Ansel, and G. Cooperman, "Transparent user-level checkpointing for the native POSIX thread library for Linux," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, vol. 6, 2006.

[65] E. Park, B. Egger, and J. Lee, "Fast and space-efficient virtual machine checkpointing," in *Proceedings of the International Conference on Virtual Execution Environments*, 2011.

[66] I. Doudalis and M. Prvulovic, "KIMA: hybrid checkpointing for recovery from a wide range of errors and detection latencies," Georgia Institute of Technology, Tech. Rep., 2010.

[67] D. Sorin, M. Martin, M. Hill, and D. Wood, "SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *Proceedings of the International Symposium on Computer Architecture*, 2002.

[68] "OpenVZ - checkpointing and live migration," http://wiki.openvz.org/Checkpointing_and_live_migration, 2014.

[69] R. Gioiosa, J. C. Sancho, S. Jiang, F. Petrini, and K. Davis, "Transparent, incremental checkpointing at kernel level: A foundation for fault tolerance for parallel computers," in *Proceedings of the Conference on Supercomputing*, 2005.

[70] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart BLCR for linux clusters," *Journal of Physics: Conference Series*, vol. 46, no. 1, 2006.

[71] O. Laadan and S. E. Hallyn, "Linux-CR: Transparent application checkpoint-restart in Linux," in *Proceedings of the Linux Symposium*, 2010.

[72] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of zap: A system for migrating computing environments," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, 2002.

[73] M. Vasavada, F. Mueller, P. H. Hargrove, and E. Roman, "Comparing different approaches for incremental checkpointing: The showdown," in *Proceedings of the Linux Symposium*, 2011.

[74] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using ksm," in *Proceedings of the Linux Symposium*, 2009.

[75] J. Heo, S. Yi, Y. Cho, J. Hong, and S. Y. Shin, "Space-efficient page-level incremental checkpointing," in *Proceedings of the ACM Symposium on Applied Computing*, 2005.

[76] B. Nicolae, "Towards scalable checkpoint restart: A collective inline memory contents deduplication proposal," in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2013.

[77] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *Proceedings of the International Symposium on Computer Architecture*, 2005.

[78] F. Douglis, "The compression cache: Using on-line compression to extend physical memory." in *Proceedings of the USENIX Annual Technical Conference*, 1993.

[79] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. R. De Supinski, and R. Eigenmann, "McrEngine: a scalable checkpointing system using data-aware aggregation and compression," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2012.

[80] D. Ibtesham, D. Arnold, P. G. Bridges, K. B. Ferreira, and R. Brightwell, "On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance," in *Proceedings of the International Conference on Parallel Processing*, 2012.

[81] J. S. Plank, J. Xu, and R. H. B. Netzer, "Compressed differences: An algorithm for fast incremental checkpointing," University of Tennessee, Tech. Rep., 1995.