Lightweight Memory Checkpointing

Dirk Vogt, Cristiano Giuffrida, Herbert Bos and Andrew S. Tanenbaum
Dept. of Computer Science
VU University Amsterdam
Amsterdam, The Netherlands
Email: d.vogt@vu.nl, {giuffrida, herbertb, ast}@cs.vu.nl

***Abstract*—Memory checkpointing is a pivotal technique in systems reliability, with applications ranging from crash recovery to replay debugging. Unfortunately, many traditional memory checkpointing use-cases require high-frequency checkpoints, something for which existing application-level solutions are not well-suited. The problem is that they incur either substantial runtime performance overhead, or poor memory usage guarantees. As a result, their application in practice is hampered. This paper presents *Lightweight Memory Checkpointing* (*LMC*), a new user-level memory checkpointing technique that combines low performance overhead with strong memory usage guarantees for high checkpointing frequencies. To this end, LMC relies on compiler-based instrumentation to shadow the entire memory address space of the running program and incrementally checkpoint modified memory bytes in a LMC-maintained *shadow state*. Our evaluation on popular server applications demonstrates the viability of our approach in practice, confirming that LMC imposes low performance overhead with strictly bounded memory usage at runtime.**

***Keywords*—Memory checkpointing; Shadow memory; Compiler-based instrumentation.**

I. Introduction

Memory checkpointing has great potential to improve the reliability of today's software stack as it plays a vital role in many important application domains. Unfortunately, prior solutions generally impose awkward tradeoffs of deployability, performance, and memory usage—in the common scenario of high-frequency memory checkpointing. This paper presents a novel memory checkpointing strategy that combines deployability, performance, and memory usage guarantees for such scenarios, facilitating practical deployment of memory checkpointing in real-world systems.

A. Memory Checkpointing

The last decade has witnessed a growing interest in memory checkpointing, an important technique that allows users to snapshot the memory image of a running program in main memory (as opposed to the disk, which is much less efficient [1]), and restore (or simply inspect) the checkpointed image later on. This is, for example, a fundamental building block in automatic error recovery techniques, which need to periodically revert the active memory image to a safe and stable state [2]–[10]. Memory checkpointing has also been applied to several other application scenarios, including debugging [11]–[14], software transactional memory [5], program backtracking [15], [16], fast initialization [17], and memory rejuvenation [17].

To be of practical use, most application scenarios require high *checkpointing frequencies*. For example, automatic error recovery techniques typically checkpoint the active memory image at every client request [4], [5] or at carefully selected rescue points [7], [10], commonly resulting in thousands of checkpoints per second. In debugging applications, frequent checkpoints allow users to efficiently inspect arbitrary memory states throughout the observed execution [14].

Memory checkpointing can be implemented at the user or at the kernel level. Kernel-level solutions are generally more efficient, but also increase the *reliable* computing base [18] of the entire system. For this reason, prior kernel-level solutions [19]–[22] have failed to reach adoption in commodity kernels—even when explicitly seeking mainline inclusion [22]. The lack of mainline support forces users to manually patch the kernel, which ultimately results in *poor deployability* guarantees of the solution in practice. Further, kernel-based implementations, in general, are very specific to the targeted kernel and as a consequence offer *bad portability* to other operating systems.

B. User-level Memory Checkpointing

Most existing solutions rely on page-granular copy-on-write (COW) or dirty page tracking mechanisms. Both can be implemented either by means of kernel mechanisms (e.g., `fork`-based COW [6] and *soft dirty bit*-based dirty page tracking [23]), or in userland using application-level page fault handling [24], [25]. All these techniques, however, ultimately rely on hardware-supported page protection mechanisms to trigger a minor page fault every time the application tries to first modify a particular memory page. At high checkpointing frequencies, the cost for implementing checkpointing operations and handling those minor page faults inevitably translates to *poor performance*, confirming that traditional user-level interfaces are ill-suited for efficient low-level memory management tasks such as memory checkpointing [26]. To address these limitations, several recent memory checkpointing solutions resort to more fine-grained instrumentation-based strategies, which—as demonstrated by preliminary results in our prior work [27]—are promising to improve memory checkpointing performance especially in high-frequency checkpointing scenarios. Some solutions rely on static analysis [4], [9] to identify all the memory objects to checkpoint, but are forced to make strong assumptions on the system model to avoid unnecessary copying (and thus *poor performance*) induced by the conservativeness of the analysis.

Those solutions record all the memory writes in an undo log generated by static [5], [15] or dynamic [10] instrumentation. Simply recording all writes in a log provides an efficient checkpointing strategy, but also yields very *poor memory usage* guarantees—as the log may grow uncontrollably when programs repeatedly write data into the same memory location.

Current remedies to this problem are largely unsatisfactory. These include swapping the log to disk [3]—translating to *poor performance*—hashing to identify duplicate log entries [15]—translating to *poor performance*—or relying on specialized hardware support—translating to *poor deployability*.

In this paper, we present *Lightweight Memory Checkpointing* (*LMC*), a new memory checkpointing technique, which combines the performance guarantees of undolog-based checkpointing with the memory guarantees of traditional page-granular checkpointing. LMC relies on a compiler-assisted *shadow state* organization—similar, in spirit, to shadow memory approaches used in state-of-the-art memory tracing techniques [28]–[35]—to incrementally checkpoint the active memory image at the byte granularity. Our prototype implementation of LMC is targeted to Linux, but as it is a pure user-land solution it is easily portable to other operating systems.

The contribution of this paper is threefold: (i) we present the design of lightweight memory checkpointing; (ii) we present a prototype implementation of LMC; (iii) we thoroughly evaluate and compare our prototype implementation to existing user-level memory checkpointing solutions and show that a carefully optimized instrumentation design can provide strong memory usage guarantees without sacrificing performance and, in some cases, even significantly improve the performance of prior undolog-based strategies [15].

II. Overview

Figure 1 presents a high-level overview of LMC. To deploy LMC, users need to link their program against the LMC checkpointing library (`liblmc.a`) and instrument it using the LMC transformation/optimization passes (`lmc.so`/`lmc-opt.so`) implemented using LLVM [36]. Both are accomplished by instructing the linker (`ld`) to use our instrumentation strategy via build flags. LMC is currently tailored to Linux programs, but our prototype is portable to other UNIX systems—Linux-specific extensions will be explicitly mentioned, hereafter.

The LMC checkpointing library exports a simple API to create and restore memory checkpoints from user programs. Internally, it also maintains LMC's shadow state organization and implements all the necessary hooks used by our instrumentation. The LMC transformation pass relies on such hooks to instrument all the memory writes in the program and incrementally maintain memory checkpoints using byte-granular copy-on-write semantics, i.e., copying every byte of memory to the shadow state at the first modification in the current *checkpoint interval*—interval between consecutive memory checkpoint/restore operations. Finally, the LMC optimization pass carefully reoptimizes the transformed code before generating the final binary.

When a user program issues a memory *checkpoint* request to our checkpointing library, LMC prepares a new shadow state for the current checkpoint and instructs our instrumentation to track all the changes to the current program state into it. By the end of the checkpoint interval, the shadow state contains a copy of all the data in the original program state that has been modified by the program since the last checkpoint operation, as well as all the necessary tracking information to locate such modifications. When a user program issues a memory *restore* request to our checkpointing library, this information allows

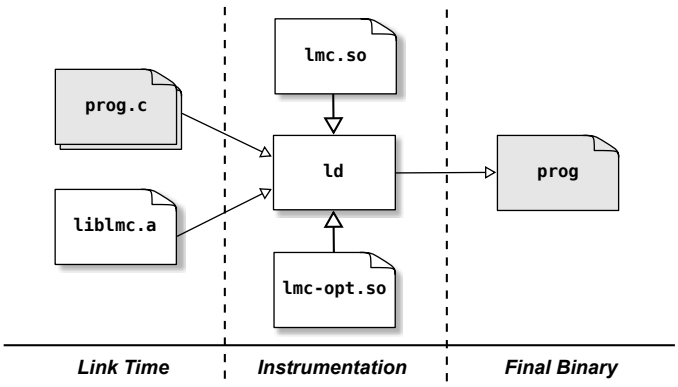

Fig. 1. High-level overview of LMC.

LMC to automatically revert the current memory image to a given memory checkpoint.

This strategy requires maintaining one shadow state for each checkpoint stored in memory. In the following, we assume a single checkpoint maintained in memory at any given time for simplicity—a common assumption in traditional memory checkpointing applications [3]–[10], [17]—but LMC can, in principle, retain an arbitrary number of checkpoints during the execution of the program. The latter is only constrained by the amount of the virtual memory address space available—limited on 32-bit programs, but a plentiful resource on modern 64-bit architectures.

A. Memory Write Instrumentation

Our memory write instrumentation tracks all the possible memory altering instructions in the original program, i.e., write instructions and calls to standard memory intrinsics—`memcpy` and `memset`—both referred to as store instructions from now on. Our transformation pass replaces each store instruction with a call to a `store_hook` function provided by `liblmc`, which (i) checks if the soon-to-be-altered memory location has not already been saved in the current checkpoint interval, (ii) copies the original data to the shadow state if necessary, and finally (iii) performs the store instruction as originally intended. Our instrumentation operates entirely at the LLVM IR level, allowing LMC to employ effective optimization strategies on the transformed code. In particular, for optimization purposes, LMC relies both on standard compiler optimizations—e.g., *inlining*, to reduce the costs associated to frequent calls to the `store_hook` function in our library—and on checkpointing-specific optimizations implemented in our optimization pass (Section V).

B. Shadow State

Our memory checkpointing strategy splits the original program state into a *primary state*—the portion of the memory address space in use by the running program—and a *shadow state*—the portion of the memory address space in use by our instrumentation to incrementally store the data associated to the current checkpoint. The tracking information for the checkpointed data, in turn, is maintained in a separate per-shadow state *tagmap*. Each tag in the tagmap refers to a predetermined memory region, providing information on whether the corresponding data in the primary state has already been saved in the

shadow state. The tagmap is entirely maintained in software and fundamental to the internal operations of our `store_hook`.

III. Shadow State Organization

Our shadow state organization fulfills three key design goals: (i) *deployability*, that is no changes to commodity operating systems, user programs, or widely deployed security mechanisms such as address space layout randomization; (ii) *portability*, that is support for multiple architectures; (iii) *efficiency*, that is fast shadow state and tagmap management with minimal run-time performance overhead.

A. Memory Address Space Layout

LMC's shadow state strategy dictates splitting the virtual memory address space of a program into three independent memory areas accommodating the primary state, the shadow state, and the tagmap. To support efficient memory lookups across the different areas, LMC maintains predetermined linear mappings for any given memory address from one area to another. This approach ensures that, given an address in the primary state, LMC can locate the corresponding address in the shadow state and the corresponding tag in the tagmap in constant time—using preassigned offsets. In the simplest shadow state organization possible, this strategy can be enforced by splitting the memory address space into three equally-sized areas, with 1-byte tags in the tagmap each referring to 1 byte in the primary (and shadow) state.

To increase the size of the primary state available to the program and improve the memory locality of the checkpointing activity, however, LMC relies on a more general tagmap implementation, with each tag referring to a generic primary/shadow memory region of ρ bytes—with ρ selected by empirical measurements by default (see Section IV-A). This design choice results in a more general shadow state organization, with the final sizing and positioning of the different memory areas subject to the particular architecture adopted. Figure 2 shows the final memory layout adopted by LMC for both 32-bit and 64-bit Linux programs.

1) 32-bit Address Space Layout: On 32-bit architectures, LMC resorts to a compact memory layout—using the `ADDR_COMPAT_LAYOUT` personality on Linux—to locate the entire primary state in the lower half of the memory address space. In particular, this strategy locates the text, data, and heap segments at the very bottom of the memory address space and memory mapped segments—i.e., anonymous mappings, file-backed mappings, and shared library mappings—above, in the first 1 GB of the address space. On Linux—and on typical UNIX systems in general—this strategy leaves the stack as the only memory area resident in the upper half of the program's usable address space. To implement its shadow state organization, LMC relocates the stack (Section III-B) during early program initialization to the lower half of the address space, leaving the upper half—starting from 1.5 GB—entirely allocated to the shadow state. The tagmap, finally, is placed at the top of the lower half of the memory address space, with a total size of 12 MB in the default $\rho = 128$ configuration (Section IV-A). This strategy yields a program-usable primary state size of less than 1.5 GB, a necessary compromise for any shadow memory organization on the limited 32-bit architecture, which, however, did not prevent LMC from successfully running all the test programs considered in our experimental evaluation.

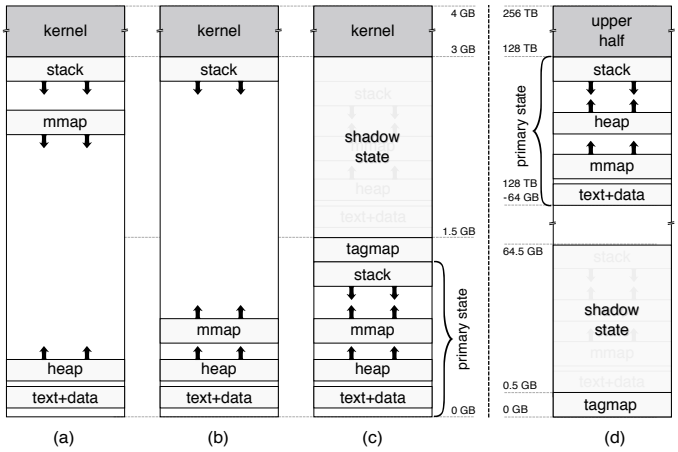

Fig. 2. The address space layout used by LMC on Linux: (a) Unmodified 32-bit layout. (b) Compact 32-bit layout. (c) Compact 32-bit layout with shadow state organization. (e) 64-bit layout with shadow state organization.

2) 64-bit Address Space Layout: On 64-bit architectures, LMC's instrumentation generates position-independent executable (PIE) binaries to freely relocate the primary state. Note that, although position-independent code is known to introduce nontrivial performance overhead on particular architectures [37], 64-bit architectures have been designed to efficiently support PIE binaries. As a matter of fact, increasingly many operating system distributions—e.g., Ubuntu—have started to ship 64-bit software packages in PIE format, which also improves the coverage of address space layout randomization and thus software security. LMC follows the same strategy, which automatically ensures relocation of the entire primary state in the upper 64 GB of the memory address space. The tagmap and the shadow state, in turn, are consecutively allocated at the bottom of the memory address space. Under the default $\rho = 128$ configuration (Section IV-A), this strategy yields a 512 MB tagmap. The upper half—starting from 128 TB—finally, is reserved at initialization time and made inaccessible to the program, for simplicity. This strategy still leaves nearly 128 TB of memory address space available to the running program.

B. Stack Relocation

To relocate the stack on 32-bit architectures, the LMC checkpointing library instruments the program to intercept the application entry point and transparently perform the relocation. On Linux, this is equivalent to overriding `glibc`'s `__libc_start_main` with a library function that allocates a new stack, copies the arguments and environment variables to the new stack location, and finally returns control to libc. This strategy, however, is alone insufficient to relocate the stack, given that the operating system is not aware of the change and, for example, can no longer export the program command-line arguments through the *proc* filesystem—i.e., `/proc/pid/cmdline`. To address this problem, LMC relies on kernel support introduced by recent user-level checkpoint-restart solutions [23], which allows a user program owning the necessary capability (i.e., `CAP_SYS_RESOURCE`) to inform the kernel of a new relocated stack via a dedicated interface (i.e., `prctl`).

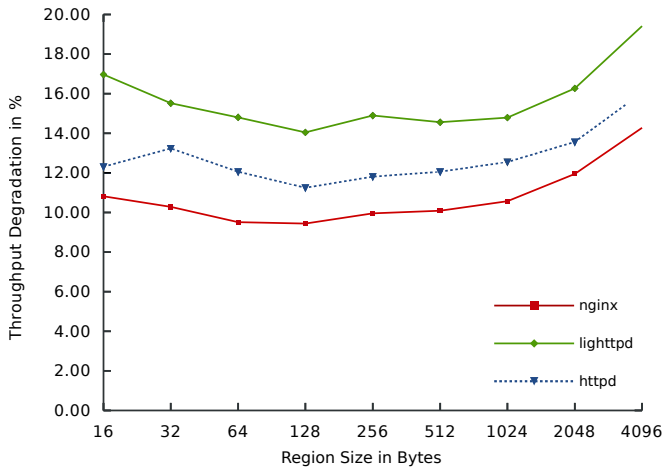

Fig. 3. Throughput degradation on lighttpd, nginx and Apache httpd for different memory region sizes.

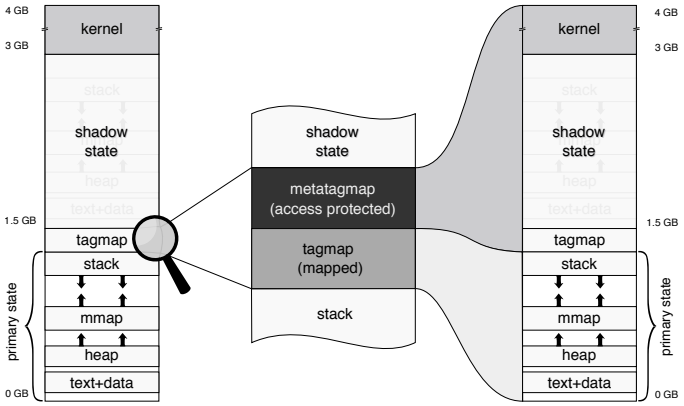

Fig. 4. Tagmap and metatagmap mappings to memory regions.

C. Tagmap

LMC maintains a tagmap using 1 byte tags and addressing memory regions of a configurable size S_t. To reconfigure the tagmap layout, users can specify a custom S_t (power of two) and recompile the checkpointing library. When called by the instrumented code, our `store_hook` locates the tag associated to the given store instruction by calculating $(a >> log_2(S_t)) + t_{off}$, where t_{off} is the offset between the base address of the primary state and the base address of the tagmap and a is the destination address for the store instruction. If the tag is not set, our `store_hook` first copies the memory region from the primary state into the shadow state and sets the tag in the tagmap before issuing the given store instruction into the primary state. The tagmap is reset to a pristine state at the next checkpoint/restore operation associated to a given shadow state. Store instructions based on memory intrinsics are segmented into multiple memory writes according to the region size selected.

IV. Tagmap Management

This section details the key issues LMC addresses to manage the tagmap as part of its checkpointing strategy.

A. Memory Region Size

The memory region size determines the size of the tagmap and also the granularity of the checkpointing strategy, i.e., a smaller region size results in a larger tagmap and a finer level of granularity. Selecting a large region size has two advantages. First, larger regions—and thus smaller tagmaps—yield more program-usable address space. Second, larger regions increase the probability of two given store instructions pointing into the same region. This leads to fewer—albeit larger—copy operations and better program-perceived locality. Large regions, however, can also lead to unnecessary copying to the shadow state, since the entire region is always copied to the shadow state even when a single byte within the region is modified in the entire checkpoint interval. These observations highlight the different tradeoffs involved in selecting the optimal memory region size.

To investigate the tradeoffs, we measured the LMC-induced throughput degradation for the three most popular web servers according to the experimental setup adopted in our evaluation (Section VII). We repeated the experiment across several different memory region sizes and reported the results in Figure 3. The figure shows that, in all cases, the throughput increases with the region size for $\rho \leq 128$ $bytes$ and decreases for $\rho > 128$ $bytes$, due to the excessive amount of unnecessary copying outweighing the benefits derived from increased locality and degrading the overall run-time performance. Based on this experiments, LMC currently assumes $\rho = 128$ $bytes$ in its default configuration.

B. Metatagmap

Memory checkpointing is often used in error recovery scenarios, where one cannot safely assume that program-issued store instructions are free from errors that could corrupt arbitrary memory address space areas, including the shadow state and the tagmap itself. To protect the program against accidental metadata corruption, LMC can be configured to use a separate *metatagmap*—allocated within the tagmap and with a similar rationale—to map all the store instructions that erroneously specify a destination address inside the shadow state, tagmap, or metatagmap itself. Inhibiting access to the metatagmap using page protection mechanisms—a strategy inspired by prior work on taint tracking techniques [33]—is sufficient to prevent arbitrary metadata corruption on invalid store instructions and induce fail-stop behavior—i.e., segmentation fault—that can be effectively handled by error recovery techniques instead. LMC's tagmap and metatagmap mapping strategy is depicted in Figure 4.

C. Resetting the Tagmap

The most straightforward tagmap implementation is a bitmap with boolean tag values. This strategy, however, requires zeroing out the entire bitmap every time LMC needs to reset the tagmap at the end of a given checkpoint interval. For this purpose, an option is to simply `bzero` the entire tagmap. Another option is to use the `mmap` system call to request the kernel to overmap the tagmap with zero pages. The latter strategy is generally more efficient—only the pages of the tagmap that are actually mapped in are zeroed—but still introduces nontrivial performance costs associated to increased

zeroing, page faulting—demand paging dictates zero pages to be only mapped in at first access—and setting up a new memory area in `mmap`. To eliminate the latter cost, our current LMC implementation relies on the `madvise` system call and the `MADV_DONTNEED` flag to efficiently repopulate a given memory area with zero pages without recreating the area, a strategy commonly employed in modern memory allocators [38]. The other costs, however, are much harder to eliminate without dedicated kernel support.

To avoid incurring such costs at every checkpoint interval, LMC opts for a more sophisticated tagmap implementation based on *epoch* numbers. For this purpose, the LMC checkpointing library keeps track of a 1-byte global epoch number incremented at every checkpoint. Every tag in the tagmap is set with the current epoch number when the corresponding memory region is first modified in a given checkpoint interval. To check whether a particular region has already been saved, LMC simply compares the corresponding tag in the tagmap with the current epoch number. This scheme eliminates the need to zero out the tagmap at every checkpoint interval, only forcing LMC to repopulate the entire tagmap with zero pages when epoch numbers run over, i.e., every 255 intervals assuming 8-bit epoch numbers.

D. Thread Safety

LMC can natively support thread safe behavior, but our current implementation disables thread safety by default. This is to eliminate extra tagmap management complexity, which is often unnecessary in practice given that memory checkpointing applications typically enforce thread safety on their own to guarantee a sound checkpointing model in a multithreaded context. For example, error recovery techniques rely on a well-defined thread model to implement their recovery activities, assuming nonthreaded execution by construction [4], [15], allowing only one thread to enter a new checkpoint interval and explicitly blocking all the other threads [3], [9], [10], or only allow checkpoint intervals that have been proven thread safe by static program analysis [39].

Enforcing thread safety at the memory checkpointing level requires LMC to synchronize accesses to LMC-maintained metadata, so that no race conditions can result from multiple threads writing into the same memory region at the same time. To address this problem, the obvious solution is to serialize accesses to the tagmap, epoch numbers, and shadow state using dedicated locks—e.g., mutexes. This strategy, however, may introduce nontrivial complexity and lock contention overhead at runtime. For this reason, LMC opts for a simpler solution which piggybacks on the synchronization mechanisms already present in the original program. To this end, LMC lowers the memory region size to 1 byte—similar to prior memory shadowing techniques for multithreaded programs [32]—a strategy which naturally yields thread safety by construction as long as the original program did not contain race conditions with two threads attempting to modify the same memory byte at the same time. This assumption may be overly conservative in error recovery applications, but such applications generally deal with thread safety explicitly, as mentioned earlier. Further, LMC needs to translate all the atomic instructions—e.g., atomic increment—into fully synchronized store operations.

V. Optimizations

This section details the optimizations adopted in our prototype to minimize the LMC-induced performance overhead.

A. Reducing Instrumentation Costs

Instrumented store instructions introduce nontrivial performance costs even if the target memory region has already been checkpointed. Such costs have two main sources: (i) new call instructions to the `store_hook` function and (ii) new load and branch instructions to check the tagmap as part of the checkpointing activity. As a result, the LMC optimization pass can minimize the performance overhead by preventing instrumentation of store instructions that static analysis can prove (i) *redundant*—i.e., operating on already checkpointed memory regions—or (ii) *transient*—i.e., operating on short-lived memory regions whose effects are never exposed outside the associated checkpoint interval and are thus not relevant for the checkpoint. Further, the LMC optimization pass *aggregates* store instructions for which the pass can statically assess good spatial locality.

1) ***Redundant Stores****:* To avoid instrumenting redundant stores, the LMC optimization pass examines each instrumented store instruction I and its pointer operand p, and creates a set S including all the other store instructions that store into the memory location pointed to by p. The analysis establish this fact by checking for equivalence of the pointer operands, i.e., stays conservative in the case of pointer aliasing. In a second step, LMC tests for each instruction $I_{canditate}$ included in S, if $I_{canditate}$ is dominated by I—i.e., I is proven to be always executed before $I_{canditate}$—and if so, un-instruments $I_{canditate}$.

2) ***Transient Stores****:* To avoid instrumenting transient stores, the LMC optimization pass seeks to identify both *heap transient stores*—i.e., store instructions referring to heap objects allocated and freed within a single checkpoint interval—and *stack transient stores*—i.e., store instructions referring to stack objects in short-lived functions, whose lifetime never spans across multiple checkpoint intervals by construction.

Heap transient stores are identified using *checkpoint escape analysis*, which follows the same static analysis strategy used in standard thread escape analysis techniques [40]. To assess whether a memory object escapes a function in the checkpoint interval, LMC relies on data structure analysis [41], an efficient context-sensitive and field-sensitive points-to analysis implemented in LLVM [36]. In particular, LMC follows the approach adopted by poolalloc [42] to identify function-local memory pools. This strategy allows LMC to identify store instructions that never *escape* a given checkpoint interval and can thus be safely left uninstrumented in the final binary.

To prevent instrumenting stack transient stores, LMC eagerly checkpoints the active call stack at checkpointing time and relies on the points-to information provided by data structure analysis [41] to avoid instrumenting *all* the store instructions that are statically proven to always refer to stack objects within the checkpoint interval. This strategy reflects the intuition that checkpoint requests are usually issued when the amount of state active on the call stack—and thus the amount of data to checkpoint eagerly—is relatively small. This is especially evident in common request-oriented checkpointing models [4], [5] in long-running applications, which typically yield minimal long-lived call stack state at memory checkpointing time.

TABLE I. LONG-LIVED CALL STACK SIZE FOR DIFFERENT SERVER APPLICATIONS.

Server	Long-lived stack in kB
nginx	1224
lighttpd	712
httpd	784
prostgresql	3048
bind	352
proftpd	5784
pureftpd	4608
vsftpd	1088

Table I confirms our intuition, showing that the size of the long-lived call stack for all the server applications considered in our evaluation is typically smaller than 1 memory page, which introduces minimal copying costs—and thus minimal performance degradation—at checkpointing time.

3) ***Aggregating Store Instrumenation****:* While not instrumenting redundant store instructions is effective for single memory locations, it cannot account for stores to different, but spatially collocated memory locations. Aggregating the instrumentation for these collocated store instructions, however, has the advantage that stores to the same underlying memory region are potentially only instrumented once.

LMC performs this optimization for store instructions into the same underlying object by constructing dominator chains of store instructions for each memory object (e.g., `struct`). The pass identifies the underlying memory object by stripping all constant offsets of LLVM's `getelementptr` instruction, and stays conservative in the case of pointer aliasing. After establishing the modified range of the memory object for each chain, all the store instruction in the chain are left uninstrumented and a call to LMC's `store_hook` covering the entire region is placed before the chain's leading instruction.

B. Reducing Checkpointing Costs

In its current form, our LMC prototype naturally imposes an always-on checkpointing strategy, given that all the relevant store instructions always run instrumented throughout the execution. In other words, either an implicit or explicit checkpoint interval is always active during the execution. While the cost of copying is gradually amortized throughout the execution when no explicit checkpoint request is issued—i.e., the same memory region is never checkpointed more than once—or can even be eliminated altogether by explicitly setting all the tags in the tagmap, the running program is still exposed to nonmarginal tagmap management costs.

To eliminate such costs, an option is to rely on program instrumentation to implement a simple basic block cloning strategy, a well-known technique incurring a relatively small memory [43] and performance impact [44], [45]. Basic block cloning results in a final binary containing two versions of each basic block and additional code to efficiently switch from one version to another on demand. For our purposes, one version would reflect code from the original uninstrumented program and the other version would reflect the corresponding code with store instructions instrumented to perform memory checkpointing. We are planning to thoroughly investigate the impact of such a basic block cloning strategy in our future work.

VI. ALTERNATIVE TECHNIQUES

This section provides a general overview of existing memory checkpointing techniques and draws a high-level comparison with LMC. An experimental comparison, in turn, is presented in Section VII. We focus here on user-level memory checkpointing, and refer the reader to existing surveys [46], [47] for more intrusive kernel-level [19]–[22], [48]–[51] and VMM-level [52] checkpointing techniques.

A. Fork-based Checkpointing

The most common way to implement page-granular checkpointing at the user level is to rely on the copy-on-write semantics supported by the `fork` system call. For our purposes, memory checkpointing can be simply implemented by spawning a new child process at the beginning of a checkpoint interval and programmatically terminate the process at the end of the interval. For its simplicity and isolation properties, fork-based checkpointing has been widely used in prior solutions [3], [6], [8], [12], [25]. This technique, however, can introduce substantial checkpoint-time overhead—`fork` requires creating a new process context—and is not entirely transparent—a new process instance is made "*visible*" to the running program.

B. Mprotect-based Checkpointing

Another popular page-granular checkpointing strategy is to rely on the `mprotect` system call to write-protect all the user memory pages and intercept the resulting write faults from a user-level `SIGSEGV` signal handler, a mechanism frequently used to implement generic user-level page fault handling. This mechanism can be used to implement COW semantics similar to fork-based checkpointing or, as an alternative, to implement dirty page tracking entirely in user space—and incrementally copy dirty pages at the end of each checkpoint interval, starting with an initial full memory checkpoint. After copying (or tracking) the faulting page, the write protection can be removed and reestablished only at the next checkpoint request. When compared to fork-based checkpointing, this technique typically introduces a more modest checkpoint-time overhead, but, at the same time, substantially increases the cost of page fault handling and lowers the isolation guarantees—the checkpointed data resides in the same address space as the running program and extra protection mechanisms are necessary to prevent data corruption.Further, this technique cannot be made application-transparent without additional recovery mechanisms in place, given that kernel execution page faulting on a write-protected page may result in a system call returning an error code to the application—i.e., `EFAULT`, according to POSIX [25]. For all these reasons, mprotect-based checkpointing has found relatively limited use in prior checkpointing solutions [16], [24], [25].

C. Soft Dirty Bit-based Checkpointing

A popular dirty page tracking strategy suitable for page-granular memory checkpointing is to rely on the dirty bit information maintained by the hardware in individual page tables entries. Traditional UNIX systems do not directly expose dirty bit information to user programs and typically require nontrivial kernel extensions to implement reliable dirty page tracking for checkpointing purposes [53], a strategy

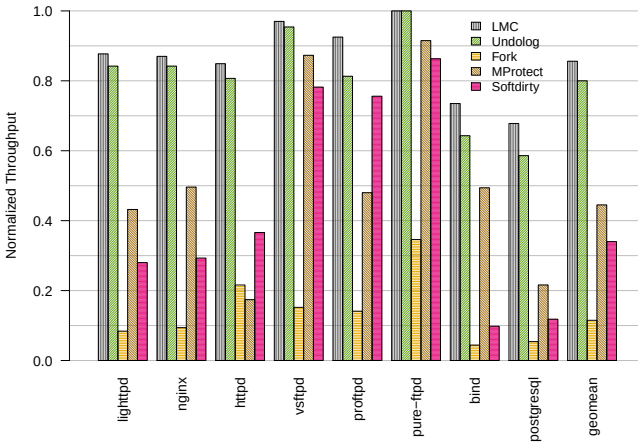

Fig. 5. Normalized throughput for our server programs across different memory checkpointing techniques (64-bit results).

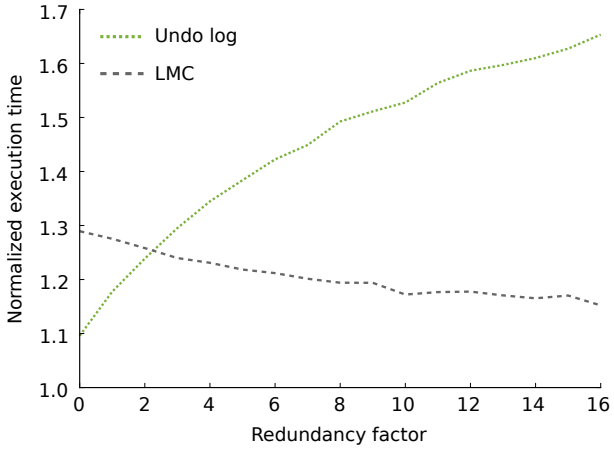

Fig. 6. Checkpointing-induced microbenchmark execution time for different redundancy factors normalized against the baseline (64-bit results).

often explored in prior kernel-level solutions [49], [51], [53]. On recent Linux releases, however, kernel support adopted for emerging checkpoint-restart frameworks [23] allow user programs to implement dirty page tracking by periodically reading and clearing software-maintained dirty bits—or *soft dirty bits*. This scheme, however, still requires extra protection mechanisms to prevent checkpoint data corruption and does not efficiently scale to large address spaces—reading soft dirty bits requires scanning all the mapped memory regions in the address space. Other user-level solutions have also suggested emulating soft dirty bit tracking using block-level checksumming [54], [55], an approach which still shares the same limitations of soft dirty bit-based tracking.

D. Undolog-based Checkpointing

Undolog-based checkpointing relies on program instrumentation techniques to log all the store instructions—i.e., their data, size, and target addresses—issued by the program within a checkpoint interval and revert the logged changes at restore time. Both dynamic [10] and static instrumentation [5], [15] techniques can be used to instrument the store instructions—we implemented the latter approach in our prototype implementation (similar to the instrumentation strategy used in LMC) to ensure a fair experimental comparison. Previous work on instrumentation-based undolog approaches [15] apply optimizations similar to LMC's "uninstrumentation" optimizations, but do not thoroughly evaluate their impact on runtime performance. To prevent checkpoint data corruption, the instrumented code needs to perform bounds checking at every store instruction to verify that the target address is not in the range in use by the undolog itself.

VII. Evaluation

We implemented LMC on Linux for Intel x86 and x64 architectures. We evaluated the resulting solution on a Intel Core2 E6550 clocked at 2.4 GHz and equipped with 4 GB of RAM (32-bit experiments) and a Intel Core i5-3340M clocked at 2.4 GHz with 8 GB of RAM (64-bit experiments).

For our experimental evaluation, we selected the three most popular open-source web servers—nginx (v0.8.54), lighttpd (v1.4.28), and Apache httpd (v2.2.23)—the three most popular open-source ftp servers—proftpd (v1.3.3), pureftpd (v1.0.36), and vsftpd (v1.2.1)—the most popular open-source name server—bind (v9.9.3)—and a popular open-source database server—postgresql (v9.0.10). We also considered all the C benchmarks in the SPEC CPU2006 benchmark suite. We instrumented all our test programs across different memory checkpointing techniques and enabled all the optimizations described in Section V for all the compiler-based checkpointing techniques (undolog and LMC itself) unless otherwise stated.

To stress the web servers, we relied on the Apache benchmark (*AB*) [56] part of the Apache httpd suite. To emulate a realistic workload, we configured *AB* to issue a total number of 25,000 requests with 10 concurrent connections and 10 requests per connection through the loopback device. To benchmark the FTP servers, we relied on the pyftpbench benchmark [57], configured to open 100 control connections and request 100 1 KB-sized files per connection. Finally, we relied on the sysbench [58] and queryperf [59] benchmarks to evaluate postgresql and the bind name server, respectively. We ran all our experiments 11 times—while checking that the CPUs were fully loaded throughout our tests—and reported the median.

A. Checkpointing Performance

To evaluate the checkpointing-induced performance overhead, we measured the throughput degradation on our server programs while checkpointing at every client request, following the common request-oriented checkpointing model adopted in prior work [4], [5]. Figure 5 presents our results for 64-bit Linux—we omit 32-bit results exhibiting similar behavior.

As shown in the figure, fork-based checkpointing induces the highest checkpointing performance overhead compared to all the other techniques (88.5% degradation, geometric mean). Further, mprotect-based checkpointing is the top-performing page-granular checkpointing technique in this scenario (55.5% degradation, geometric mean). In particular, mprotect-based

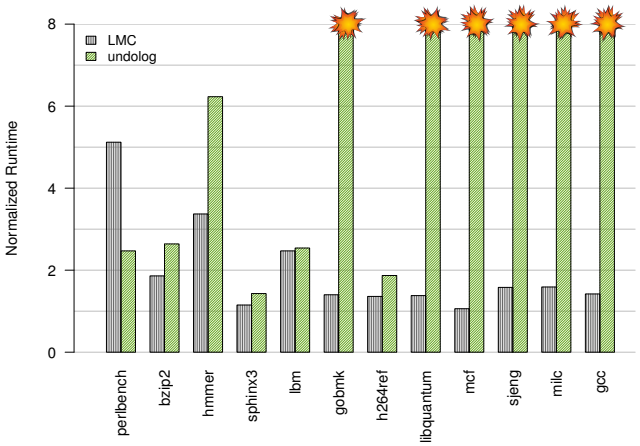

Fig. 7. Checkpointing performance on SPEC for LMC and undolog (64-bit results).

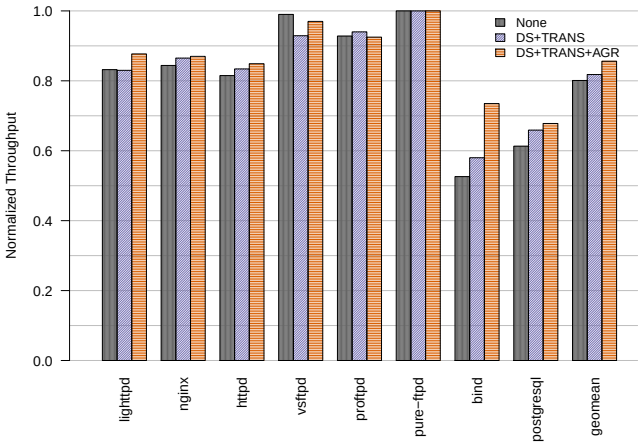

Fig. 9. Throughput of server programs with LMC using different combinations of optimizations.

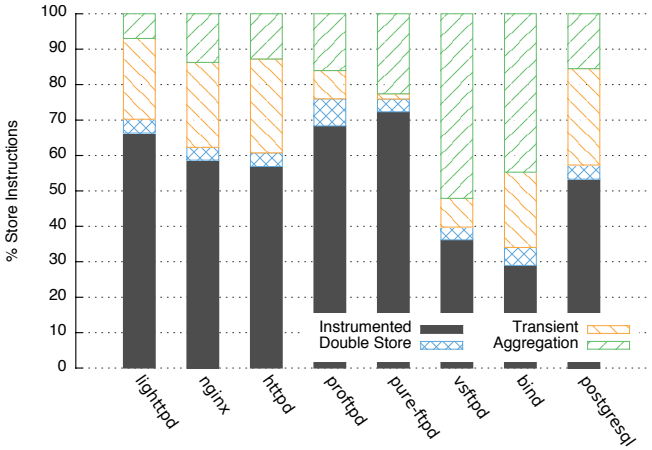

Fig. 8. Relative number of uninstrumented store instructions for different server programs.

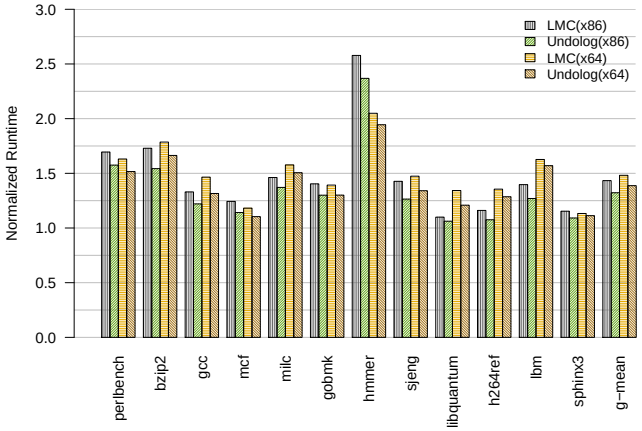

Fig. 10. Instrumentation-induced execution time for the C programs in the SPEC CPU2006 benchmark suite normalized against the baseline. Actual state saving is disabled in the `store_hook`.

checkpointing reported remarkable performance for programs that exhibit good locality, e.g., Apache httpd, which memory profiling revealed modifying the smallest number of memory pages in the selected checkpoint interval. Finally, instrumentation-based techniques significantly outperform page-granular techniques in most cases, as anticipated. In particular, our results show that LMC performs comparably, and even better on average (15.0% vs 20.0% degradation, geometric mean), than undo log-based checkpointing. In detail, LMC consistently outperforms undolog-based checkpointing for memory-intensive programs, e.g., bind, and programs that exhibit significant write locality, e.g., Apache httpd. In both scenarios, a large number of duplicate writes can cause the undolog to grow quickly, disrupting spatial locality and increasing cache trashing.

B. *Effectiveness of the Optimizations*

Figure 7 shows the checkpointing performance for the SPEC CPU2006 benchmark suite using our LMC and the undolog checkpointing technique. To simulate an event-based recovery scenario, we identified an inner loop inside the programs. During each iteration of the loop, a checkpoint is taken, resulting in checkpoint intervals ranging from 13 microseconds (perlbench) to 391 seconds (milc). A special case is gcc, in which case only one checkpoint is taken, since no inner loop could be identified. The results in the figure are ordered by checkpoint frequency.

We limited the undolog to 4 GB leading to programs crashing, due to undolog overflows in cases where the checkpoint interval is fairly large. Programs start to crash when checkpointed with a frequency of 0.39 seconds (gobmk). This underlines the necessity of memory boundedness. Further, the figure shows that in nearly all cases LMC outperforms the undolog-based technique. Only for perlbench, which is checkpointed with an extremely high frequency, the undolog performs better than LMC. In this case, the very low cost incurred by the undolog to start a new checkpoint—i.e., simply resetting the index into the log—is the key to better performance.

To evaluate the effectiveness of the optimizations operated by LMC to reduce the instrumentation overhead, we measured (1) the number of store instructions that are uninstrumented by

TABLE II. CHECKPOINTING-INDUCED PSS INCREASE FOR OUR SERVER PROGRAMS ACROSS DIFFERENT MEMORY CHECKPOINTING TECHNIQUES AND INTERVALS.

	Baseline PSS	LMC			Undolog			mprotect		
Requests:		1	10	100	1	10	100	1	10	100
nginx	1872 KB	20 %	19 %	20 %	9 %	21 %	136 %	12 %	12.6 %	12 %
lighttpd	851 KB	33 %	43 %	33 %	8 %	32%	184 %	16 %	19 %	15 %
httpd	3257 KB	52 %	54 %	52 %	26 %	128 %	1193 %	16 %	16%	15 %
proftpd	71982 KB	94 %	94 %	93 %	420%	900 %	5900 %	8 %	5 %	5 %
pureftpd	268 KB	41 %	39 %	39 %	14 %	33 %	208 %	111 %	121 %	195 %
vsftpd	89 KB	29 %	35 %	35 %	5 %	9 %	11 %	5 %	13%	11 %
bind	8897 KB	27 %	26 %	27 %	79 %	94%	218 %	2 %	3 %	1 %
postgresql	20919 KB	11 %	29 %	13 %	412 %	470 %	1104 %	16 %	5 %	1 %

the different optimization stages and (2) the resulting impact on the run-time performance of the server programs.

1) Uninstrumented Store Instructions: Figure 8 shows the percentage of store instructions that are uninstrumented by the different optimization stages. The results show that our optimizations are generally effective, leading to a reduction of instrumented store instructions of between 29 % (bind) and 72 % (pure-ftpd). In addition, the effectiveness of the double store optimization (3.6–7.6 % of all store instructions uninstrumented) is outweighted by the effectiveness of the transient store and aggregation optimizations.

Further, our results show that, in some cases, (proftpd, nginx, httpd, and postgresql) the aggregation optimization is able to uninstrument the largest fraction of store instructions. In the other cases (lighttpd, pureftpd, vsftpd, and bind), the transient store optimization is able to uninstrument the largest fraction of store instructions. We attribute this behavior to the latter programs' heavier use of stack-allocated variables.

2) Run-time Impact of the Optimizations: Figure 9 shows the normalized throughput of our unoptimized (None), doubles store and transient store optimized(DS+Trans) and fully optimized(DS+Trans+AGR) server programs. The general trend shows that uninstrumenting store instructions is indeed reflected in better run-time performance, leading, on average, to a performance gain of 1.8 % for the DS+TRANS case and and additional 4.8 % when also enabling the aggregation optimization.

At the same time, our results also show that an increase in the number of uninstrumented instructions induced by a certain optimization is not necessarily reflected in an equally-sized performance gain, since the uninstrumented store instructions may happen to lie in cold (i.e., nonperformance critical) code paths. This was, for example, the case for postgresql, where the aggregation optimization uninstruments the largest fraction of store instructions but has a smaller impact on the overall performance. Finally, in some cases (lighttpd, vsftpd, and proftpd), our results seem to suggest that optimizations may occasionally have a slightly negative performance impact. In practice, the reported slowdowns are well within the noise caused by optimization-induced memory layout changes [60].

C. Impact of Duplicate Writes

To compare the previously noted impact of duplicate writes on instrumentation-based memory checkpointing techniques, we relied on a homegrown microbenchmark, which runs through 5000 loop iterations, each of which checkpoints the entire memory image and subsequently writes 1 KB of data into a 128 KB memory range—sampled uniformly at each iteration. To simulate duplicate writes with a redundancy factor of R, we repeated each write operation inside the loop R times. Figure 6 shows the time to complete the checkpointing-enabled version of our (64-bit) microbenchmark normalized against the baseline—for growing values of R. As we can see, undolog-based checkpointing yields better performance only for $R = \{0, 1\}$ but it is increasingly outperformed by LMC for greater values of R.

D. Instrumentation Performance

To evaluate the performance overhead induced by instrumentation-based memory checkpointing techniques with no checkpoint operation issued during regular execution, we measured the time to complete an instrumented version of the C programs in the SPEC CPU2006 benchmark suite compared to the baseline. As checkpointing-induced costs are not considered, we disabled logging for undolog-based checkpointing and epoch number management for LMC.

Figure 10 shows that the instrumentation-induced performance overhead lies between 17 % and 206 %. Further, the overhead for LMC is slightly higher than that for the undolog. This is especially the case for libquantum and lbm, which introduce significant cache pressure [61]. The latter is further increased by tagmap management operations operated by LMC's `store_hook` function. Finally, Figure 10 shows that the overall overhead is slightly higher for 64-bit systems, which we attribute to the position-independent code used by our prototype implementation on such systems. The apparent speedup reported for hmmer, in turn, is likely caused by a higher instruction per cycle ratio on 64-bit architectures [61].

E. Memory Usage

To evaluate the checkpointing-induced memory usage overhead, we measured the *Proportional Set Size* (*PSS*)—physical memory usage normalized to account for shared memory pages in a multiprocess context—increase on our servers programs while checkpointing at every $R = \{1, 10, 100\}$ requests—highlighting the memory usage growth for the different techniques. Table II presents our findings—omitting fork-based checkpointing results, comparable to mprotect-based results but

harder to stabilize with nonatomic PSS measurements across multiple processes. As expected, LMC generally consumes more memory than page-granular checkpointing techniques (32% vs 12% increase with $R=1$, geometric mean), but induces a similarly limited and steady memory usage increase across different checkpoint intervals. Undo log-based checkpointing, in contrast, introduces a substantial memory usage increase, which grows very quickly as we relax the duration of the checkpoint interval. For example, with $R=10$, undolog-based checkpointing induces a PSS increase of 900 % in the worst case—i.e., proftpd—which quickly grows up to 6000 % with $R=100$.

VIII. Conclusion

Existing high-frequency memory checkpointing techniques operating at the user level force users to tradeoff performance and memory usage guarantees, a painful compromise when systems reliability is at stake. To address these concerns, this paper presented LMC, a new memory checkpointing technique based on a compiler-assisted shadow state organization which efficiently implements byte-granular copy-on-write semantics. To evaluate the viability of our approach, we implemented LMC for generic 32- and 64-bit Linux programs and evaluated it on eight popular open-source server applications using the common request-oriented checkpointing model. Our experimental results show that LMC matches the performance guarantees of state-of-the-art instrumentation-based strategies—i.e., undolog—while also providing much stronger memory usage guarantees.

Acknowledgment

We would like to thank the anonymous reviewers for their comments. This work was partially supported by "the Rosetta" (ERC Starting Grant 259108) and "Re-Cover" (NWO 628.001.006) projects.

References

[1] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *TPDS*, vol. 9, no. 10, p. 972–986, Oct. 1998.

[2] J. L. Lawall and G. Muller, "Efficient incremental checkpointing of Java programs," in *DSN*, 2000, pp. 61–70.

[3] A. Zavou, G. Portokalidis, and A. D. Keromytis, "Self-healing multitier architectures using cascading rescue points," in *ACSAC*, 2012, pp. 379–388.

[4] C. Giuffrida, L. Cavallaro, and A. S. Tanenbaum, "We crashed, now what?" in *HotDep*, 2010, pp. 1–8.

[5] A. Lenharth, V. S. Adve, and S. T. King, "Recovery domains: An organizing principle for recoverable operating systems," in *ASPLOS*. ACM, 2009, pp. 49–60.

[6] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: Treating bugs as allergies—a safe method to survive software failures," in *SOSP*, 2005, pp. 235–248.

[7] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis, "ASSURE: Automatic software self-healing using rescue points," in *ASPLOS*, 2009, pp. 37–48.

[8] Q. Gao, W. Zhang, Y. Tang, and F. Qin, "First-aid: Surviving and preventing memory management bugs during production runs," in *EUROSYS*, 2009, pp. 159–172.

[9] A. Kadav, M. J. Renzelmann, and M. M. Swift, "Fine-grained fault tolerance using device checkpoints," in *ASPLOS*, 2013, pp. 473–484.

[10] G. Portokalidis and A. D. Keromytis, "REASSURE: A self-contained mechanism for healing software using rescue points," in *ACSAC*, 2011, pp. 16–32.

[11] D. Subhraveti and J. Nieh, "Record and transplay: Partial checkpointing for replay debugging across heterogeneous systems," in *SIGMETRICS*, 2011, pp. 109–120.

[12] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, "Flashback: A lightweight extension for rollback and deterministic replay for software debugging," in *USENIX ATC*, 2004, p. 3.

[13] J. Hursey, C. January, M. O'Connor, P. H. Hargrove, D. Lecomber, J. M. Squyres, and A. Lumsdaine, "Checkpoint/restart-enabled parallel debugging," in *EuroMPI*, 2010, pp. 219–228.

[14] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging operating systems with time-traveling virtual machines," in *USENIX ATC*, 2005, p. 1.

[15] C. C. Zhao, J. G. Steffan, C. Amza, and A. Kielstra, "Compiler support for fine-grain software-only checkpointing," in *CC*, 2012, pp. 200–219.

[16] E. Bugnion, V. Chipounov, and G. Candea, "Lightweight snapshots and system-level backtracking," in *HotOS*, 2013, p. 23.

[17] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala, "Checkpointing and its applications," in *FTCS*, 1995, p. 22.

[18] B. Döbel and H. Härtig, "Who watches the watchmen? – protecting operating system reliability mechanisms," in *HotDep*, 2012.

[19] "OpenVZ," http://openvz.org.

[20] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of zap: A system for migrating computing environments," in *OSDI*, Dec. 2002, p. 361–376.

[21] O. Laadan and J. Nieh, "Transparent checkpoint-restart of multiple processes on commodity operating systems," in *USENIX ATC*, 2007, pp. 1–14.

[22] O. Laadan and S. E. Hallyn, "Linux-CR: Transparent application checkpoint-restart in linux," in *Linux Symposium*, 2010, pp. 159–172.

[23] "CRIU," http://criu.org.

[24] J. F. Ruscio, M. A. Heffner, and S. Varadarajan, "DejaVu: Transparent user-level checkpointing, migration and recovery for distributed systems," in *SC*, 2006.

[25] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under unix," in *USENIX ATC*, 1995, p. 18.

[26] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged CPU features," in *OSDI*, 2012, pp. 335–348.

[27] D. Vogt, C. Giuffrida, H. Bos, and A. S. Tanenbaum, "Techniques for efficient in-memory checkpointing," in *HotDep*, 2013.

[28] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: A fast address sanity checker," 2012, p. 28.

[29] Q. Zhao, D. Bruening, and S. Amarasinghe, "Efficient memory shadowing for 64-bit architectures," in *ISMM*, 2010, pp. 93–102.

[30] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong, "How to do a million watchpoints: Efficient debugging using dynamic instrumentation," in *CC*, 2008, pp. 147–162.

[31] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in *VEE*, 2007, pp. 65–74.

[32] M. Payer, E. Kravina, and T. R. Gross, "Lightweight memory tracing," in *USENIX ATC*, 2013.

[33] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks," in *USENIX Security*, 2006, pp. 121–136.

[34] Q. Zhao, D. Bruening, and S. Amarasinghe, "Umbra: Efficient and scalable memory shadowing," in *CGO*, 2010, pp. 22–31.

[35] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *PLDI*, 2007, pp. 89–100.

[36] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004, p. 75.

[37] M. Payer, "Too much PIE is bad for performance," Tech. Rep., 2012.

[38] P. Akritidis, "Cling: A memory allocator to mitigate dangling pointers," in *USENIX Security*, 2010, p. 12.

[39] W. Zhang, M. de Kruijf, A. Li, S. Lu, and K. Sankaralingam, "ConAir: Featherweight concurrency bug recovery via single-threaded idempotent execution," in *ASPLOS*, 2013, pp. 113–126.

[40] Y. Sade, M. Sagiv, and R. Shaham, "Optimizing C multithreaded memory management using thread-local storage," in *Compiler Construction*, pp. 137–155.

[41] C. Lattner, A. Lenharth, and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," in *PLDI*, 2007, pp. 278–289.

[42] C. Lattner and V. Adve, "Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap," in *PLDI*, Chigago, Illinois, June 2005.

[43] A. Jimborean, V. Loechner, and P. Clauss, "Handling multi-versioning in LLVM: Code tracking and cloning," in *Workshop on Intermediate Representations*, Apr. 2011.

[44] J. Wu, H. Cui, and J. Yang, "Bypassing races in live applications with execution filters," in *OSDI*, 2010.

[45] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "EDFI: A dependable fault injection tool for dependability benchmarking experiments," in *PRDC*, 2013.

[46] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and Checkpoint/Restart implementations for high performance computing systems," *J. Supercomput.*, vol. 65, no. 3, p. 1302–1326, Sep. 2013.

[47] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa, and S. Jiang, "Current practice and a direction forward in Checkpoint/Restart implementations for fault tolerance," in *IPDPS*, 2005, p. 300.

[48] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (BLCR) for linux clusters," *Journal of Physics: Conference Series*, vol. 46, no. 1, p. 494, Sep. 2006.

[49] Y. Li and Z. Lan, "FREM: A fast restart mechanism for general Checkpoint/Restart," *IEEE Trans. Comput.*, vol. 60, no. 5, p. 639–652, May 2011.

[50] A. Zarrabi, K. Samsudin, and W. A. Wan Adnan, "Linux support for fast transparent general purpose Checkpoint/Restart of multithreaded processes in loadable kernel module," *J. Grid Comput.*, vol. 11, no. 2, pp. 187–210, Jun. 2013.

[51] 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 *Super Computing*, 2005, p. 9.

[52] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *SOSP*, 2003, pp. 164–177.

[53] M. Vasavada, F. Mueller, P. H. Hargrove, and E. Roman, "Comparing different approaches for incremental checkpointing: The showdown," in *Linux Symposium*, 2011, pp. 69–79.

[54] K. B. Ferreira, R. Riesen, R. Brighwell, P. Bridges, and D. Arnold, "libhashckpt: Hash-based incremental checkpointing using GPU's," in *EuroMPI*, 2011, pp. 272–281.

[55] H.-c. Nam, J. Kim, S. Hong, and S. Lee, "Probabilistic checkpointing," in *FTCS*, 1997, p. 48.

[56] "Apache benchmark (AB)," http://httpd.apache.org/docs/2.0/programs/ab.html.

[57] "pyftpdlib," https://code.google.com/p/pyftpdlib.

[58] "SysBench," http://sysbench.sourceforge.net.

[59] "BIND," http://www.isc.org/downloads/bind.

[60] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *ASPLOS*, 2009, pp. 265–276.

[61] D. Ye, J. Ray, C. Harle, and D. R. Kaeli, "Performance characterization of SPEC CPU2006 integer benchmarks on x86-64 architecture." in *IISWC*, 2006, pp. 120–127.