Techniques for Efficient In-Memory Checkpointing

Dirk Vogt Cristiano Giuffrida Herbert Bos Andrew S. Tanenbaum

d.vogt@vu.nl, {giuffrida,herbertb,ast}@cs.vu.nl

The Network Institute, VU University Amsterdam

ABSTRACT

Checkpointing is a pivotal technique in system research, with applications ranging from crash recovery to replay debugging. In this paper, we evaluate a number of in-memory checkpointing techniques and compare their properties. We also present a new compiler-based checkpointing scheme which improves state-of-the-art performance and memory guarantees in the general case. Our solution relies on a *shadow* state to efficiently store incremental in-memory checkpoints, at the cost of a smaller user-addressable virtual address space. Contrary to common belief, our results show that in-memory checkpointing can be implemented efficiently with moderate impact on production systems.

1. INTRODUCTION

In-memory checkpointing is an important technique that allows one to record (and later restore) a memory snapshot of the entire program state. This general strategy has been applied in different contexts, including crash recovery [5,7,10, 12,13], replay debugging [6,8,14,16], and automated program backtracking [16].

Traditional in-memory checkpointing implementations operate at the page level using the standard copy-on-write (COW) mechanisms offered by modern operating systems [8, 13]. Albeit simple and effective, this strategy can translate to much unnecessary copying for checkpointing purposes, ultimately resulting in poor end-to-end performance especially for programs that exhibit sparse memory write patterns.

More recent techniques [5,7] have relied on static program analysis to restrict memory copying only to the set of objects that can possibly be modified by the program in the current checkpoint/restart transaction. The conservative nature of the analysis, however, can greatly overestimate the amount of data to be copied in the general case, de facto limiting the technique to checkpoint/restart intervals with well-defined and scoped entry points (e.g., driver calls [7]).

HotDep'13, November 03 - 06 2013, Farmington, PA, USA

Other techniques have suggested recording individual memory writes in an *undo* log generated by compiler-based [10,16] or dynamic [12] instrumentation. While an important step forward over prior techniques, this strategy is still prone to unnecessary copying and unbounded memory overhead for programs that exhibit highly-duplicated memory write patterns.

In this paper, we compare traditional page-granular checkpointing with more fine-grained compiler-based techniques implemented entirely at the application level. In particular, we evaluate the tradeoffs between undo log-based checkpointing and our newly proposed scheme based on a *shadow* state entirely managed by instrumentation. We show how our techniques scale efficiently in presence of duplicate writes, thus outperforming existing solutions in the general case. Our evaluation demonstrates that in-memory checkpointing can be realistically deployed in production systems for general use.

2. CHECKPOINTING TECHNIQUES

In this section we describe the in-memory checkpointing techniques considered and compare their key properties in terms of performance, memory consumption, and run-time behavior.

Page-granular Checkpointing

Traditional page-granular checkpointing leverages the features offered by modern MMUs to implement COW semantics and limit the amount of copying that a full memory snapshot would otherwise require. This is typically done by write protecting the entire memory address space and copying the content of a page to a safe location when the first write occurs.

A straightforward way to implement this strategy on UNIX systems is to create checkpoints using the `fork` system call, which delegates the COW semantics entirely to the kernel. As we are only interested in maintaining one in-memory checkpoint at the time (sufficient for most checkpointing applications), we can simply kill the child process and `fork` again when the next checkpoint is requested.

An alternative is to use the `mprotect` system call to write protect all the user pages and implement COW semantics in an application-level `SIGSEGV` signal handler. In practice, the signal handling overhead can significantly hurt performance and most implementations—including ours—resort to the simpler and faster `fork`-based strategy described earlier.

Undo Log

Undo log-based checkpointing techniques [12, 16] log all the memory writes performed by the program after the last checkpoint request. Every single write is typically recorded in an undo log entry that contains the address, size, and previous content of the target memory location. To reduce the overhead, the log is never searched for duplicate entries, which are simply appended to the end of log. This strategy still allows undoing the entire log to reconstruct the last checkpoint at any point during program execution. The log is maintained in memory and the write offset reset in constant time when the next checkpoint is requested.

To intercept individual memory writes, both static [16] and dynamic [12] instrumentation strategies are possible. In our prototype, we opted for the former strategy since it is faster and allowed us to implement several optimizations to reduce the number of instrumented memory writes, as discussed in [16]. We implemented our compiler-based instrumentation as a link-time pass with the LLVM compiler framework [9]. Our pass prepends each memory-altering instruction (i.e., store, `memcpy`, etc.) with a call instruction to our homegrown logging function. The latter simply creates a new memory write entry and appends it to the log.

2.1 Shadow State

While relatively efficient, undo log-based techniques incur nontrivial extra costs to store log entry metadata and record duplicated entries. To address these issues, we propose a new *shadow* state-based approach to keep track of all the memory changes occurred from the last checkpoint request, similar, in spirit, to Valgrind's Memcheck [11]. The shadow-state checkpointing strategy splits the address space into a primary state (i.e., the original program state), a shadow state (i.e., changed data from the last checkpoint), and a tagmap keeping track of the state changes. Each tag in the tagmap describes a specific region of memory in the primary state (currently 128 bytes) and indicates whether its counterpart in the shadow state contains a copy of the original checkpointed data. The tagmap and the shadow state are located at the end of the virtual address space at fixed offsets. As a result, every byte of the primary state has a corresponding tag and a candidate shadow copy located only one *constant* offset away in memory.

In the simplest case, the tagmap can be implemented as a simple bitmap, with every bit marking the target region as being "*shadowed*" from the last checkpoint request. This scheme, however, would require resetting the tagmap every time a new checkpoint is requested. While our current implementation mitigates the cost of resetting the tagmap by overmapping it with kernel-prefetched zero pages, the cost of doing so has still proven nontrivial. For this reason, our tagmap opts instead for 8-bit epoch numbers to express a single tag. The epoch number is incremented at every new checkpoint, allowing the tagmap to be reset only every 255 cycles.

Our current prototype implements this strategy with another LLVM link-time pass, which instruments memory writes to create a shadow copy of a given region in the primary state *only* on the first write from the last checkpoint, essentially implementing byte-level COW semantics. This is done by checking if the target tag stored in the tagmap matches the current epoch number and by shadowing the target region (and also updating the tagmap) in case of mismatch.

Method	Performance Overhead	Virtual Memory Overhead	Physical Memory Overhead	Data Integrity
Page-granular	–	++	+	++
Undo Log	+	–	–	+
Shadow State	++	–	+	+

Table 1: Comparison of the checkpointing techniques considered. A plus sign denotes a positive characteristic according to a given property.

3. COMPARISON

This section provides a comparison of all the presented techniques, with key properties and results summarized in Table 1.

3.1 Performance Overhead

Page-granular checkpointing is only efficient when the program exhibits high spatial locality (with an arbitrary number of duplicate writes) and checkpoints are infrequent events. When the former assumption is violated, sparse memory writes can trigger several page-level COW operations which can slow down the execution with unnecessary memory copying. When the latter assumption is violated, the cost of frequent `fork` system calls can impose a significant performance penalty.

Undo log-based checkpointing has somewhat complementary properties compared to page-granular checkpointing. First, the checkpointing frequency has essentially no impact on performance, given that a log reset is a relatively inexpensive operation. Second, the checkpointing strategy is agnostic to the locality of the program, given the low cache pressure expected by sequential writes into the log. The inability to handle duplicate writes, however, results in extra costs associated to redundant memory copying, with performance overhead increasing linearly with the number of memory writes.

Shadow state-based checkpointing is somewhat more sensitive to the locality of the program (memory write patterns in the shadow state and the tagmap follow closely those of the primary state) and the checkpointing frequency (the cost of resetting the tagmap is amortized by our epoch-based tagging strategy), but we expect the performance impact of these factors to be negligible in practice. The shadow state, however, eliminates any redundant memory copying associated to duplicate writes, which results in much better scalability with the number of memory writes in the general (and *average*) case.

3.2 Memory Overhead

As shown in Table 1, page-granular checkpointing is the one approach that exhibits nearly optimal virtual/physical memory behavior. In particular, this strategy has no virtual memory overhead (the `fork`-based strategy checkpoints data in a different address space) and typically low physical memory overhead (only touched physical pages need to be

reallocated), although the latter may increase rapidly for sparse memory write patterns.

Undo log-based checkpointing can potentially have the lowest virtual and physical memory overhead (even better than page-granular checkpointing in presence of sparse memory write patterns), but its inability to discard duplicate writes results in very poor guarantees on the memory consumption and makes the memory overhead unbounded in the general case.

Shadow state-based checkpointing, in turn, provides the worst-case virtual memory overhead, with more than half of the user-addressable address space reserved for checkpointing purposes. This is a necessary compromise to perform efficient tagmap lookups and shadow copying. Although this might lead to the problem of address space exhaustion on 32-bit systems, we did not run into problems for our test applications. Further, on 64-bit systems this problem is virtually non-existent. In addition, this property has no impact on physical memory usage, given that demand paging guarantees our shadow state-based strategy to be asymptotically equivalent to page-granular checkpointing (extra memory is only required for used tagmap pages).

3.3 Data Integrity

Given that one of the main goals of in-memory checkpointing is to recover or examine the last checkpointed state after a crash (or some form of state corruption) occurs, a good checkpointing strategy should strive to preserve data (and metadata) integrity and be able to restore the last checkpoint correctly even in face of program-generated memory errors.

Page-granular checkpointing provides the highest level of data integrity, given that checkpointed data is always isolated in a separate address space not directly accessible from the original program. Both the undo log-based and the shadow state-based scheme, in turn, maintain data and metadata in the original program address space, thus providing inherently lower data integrity guarantees. Shielding data and metadata from memory errors caused by instrumented memory writes in the program is, however, still a viable option. For this purpose, we extended our log-based implementation to perform pre-write bounds checking (to prevent the program from writing into the undo log) and our shadow state-based implementation to use an extra *"shadow"* tagmap, which tagmaps all the memory writes into both the original tagmap and the shadow state, similar, in spirit, to [15]. This strategy requires no extension to our original implementation and will cause any instrumented write operation to read from an unmapped tagmap (and thus segfault).

4. EVALUATION

To evaluate the performance overhead of all the considered techniques, we measured the checkpointing-induced throughput degradation on three real-world web servers and the runtime overhead of one scientific computing application. In addition, we developed a memory-intensive microbenchmark to further stress the proposed techniques and fully explore the design space. In particular, we wish to compare our shadow state-based checkpointing technique with prior fine-grained strategies (undo log) with respect to locality and duplicate writes. We implemented all the proposed techniques on a 32-bit Linux installation and performed our experiments on a Intel Core2 Dou E6550 clocked at 2.33 GHz.

	Fork	Undo Log	Shadow State
`nginx`	79.5 %	11.7 %	9.5 %
`lighttpd`	87.2 %	16.6 %	14.0 %
`httpd`	75.2 %	14.2 %	11.6 %

Table 2: Throughput degradation for `nginx`, `lighttpd` and `httpd` using different checkpointing techniques (4 kilobyte sized file).

	Fork	Undo Log	Shadow State
`nginx`	76.0 %	2.2 %	1.0 %
`lighttpd`	81.4 %	0.9 %	0.2 %
`httpd`	66.7 %	10.4 %	7.4 %

Table 3: Throughput degradation for `nginx`, `lighttpd` and `httpd` using different checkpointing techniques (64 kilobyte sized file).

4.1 Macrobenchmarks

Web Server Benchmarks. We evaluated the throughput degradation induced by the proposed techniques on `nginx` [4], `lighttpd` [3] and Apache `httpd` [1], three popular open source web servers. The event-driven design of `nginx` and `lighttpd` naturally yields predetermined execution points—the top of a long-running event-handling loop—in which the program state is stable and suitable for checkpointing and recovery [5]. For our purposes, we instrumented `nginx` and `lighttpd` to create a checkpoint every iteration of the event-handling loop. Further, we instrumented Apache `httpd` to perform a checkpoint whenever the http-request handler-function is called.

We measured the throughput in terms of processed requests per second using the Apache benchmark (`AB`) part of Apache `httpd`. To mimic a realistic scenario, we configured `AB` to perform 25.000 requests on 10 parallel connections with 10 requests per connection requesting a 4 kilobyte file and in a second run a 64 kilobyte file. We compared the throughput of the checkpoint-enabled version of our servers against the baseline and reported the mean of 11 runs.

Tables 2 and 3 show the throughput degradation introduced by the proposed techniques for 4 kilobyte and 64 kilobyte sized files, respectively. As expected, fork-based checkpointing yields with 67 % to 81 % throughput degradation the highest performance impact. The results, in turn, confirm that our shadow-state based technique provides better performance than the undo log-based strategy, with a 2.2 (`nginx`) to 3.0 (`httpd`) percentage points lower throughput degradation.

To examine the effect of different region sizes on the shadow-state based approach's performance, we measured the throughput degradation for region sizes from 16 bytes to 4096 bytes. The results in Figure 1 show that the optimal region size for all three web-servers is 128 bytes. For region sizes bigger than 1024 bytes the performance drops significantly. This advocates that fine-grained checkpoints can help to improve checkpointing performance in general.

Scientific Computation Benchmark. To stress the presented checkpointing techniques we also ran experiments with `hmmer` [2], a CPU-intensive program that uses hidden Markov models to search in gene sequence databases. In this case we instrumented the application to take a checkpoint for each

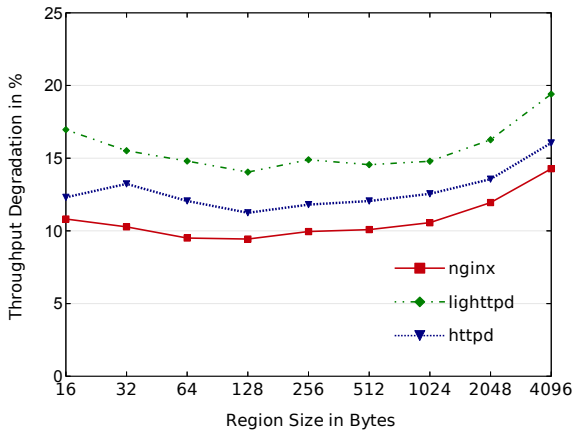

Figure 1: Throughput degradation of the shadow-state based approach with different region sizes for `nginx`, `lighttpd` and `httpd`.

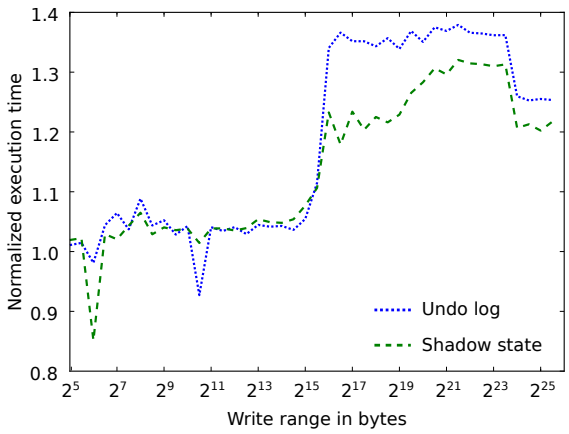

Figure 2: Normalized execution time for different memory ranges.

iteration of its compute loop. The runtime of the uninstrumented program is 226 seconds. The runtime overhead for the instrumented versions can be found in Table 4.

Fork	Undo Log	Shadow State
202.8 %	121.5 %	64.4 %

Table 4: Runtime overhead for `hmmer` using different checkpointing techniques.

Again, the fork-based approach performs worst, whereas our shadow-state scheme has 56.1 percentage points less runtime overhead than the undo log. Another interesting fact is the memory consumption: As expected, the shadow-state roughly doubles the memory consumption from 33 megabytes to 80 megabytes. The undo log however grows during the computation up to more than 1.5 gigabytes, which is a clear indication for a significant amount of duplicate writes.

4.2 Microbenchmarks

To thoroughly analyze the behavior of fine-grained checkpointing techniques in presence of poor locality and duplicate writes, we evaluated the performance overhead imposed on the completion of our own homegrown microbenchmark. We designed the latter to stress our checkpointing implementations as much as possible, with the base parameters chosen to resemble `nginx`'s write behavior. Our microbenchmark performs byte-wise memory writes for a total of 1 kilobytes distributed over a range of R bytes for 5000 iterations of a long-running (checkpointed) loop. We also introduced a redundancy factor r, which instructs the microbenchmark to repeat every single write $r + 1$ times at every loop iteration.

First, we explored the effect of bad locality. For this purpose, we varied the write range R between 32 bytes and 32 megabytes, while setting the redundancy factor r to 4. Figure 2 depicts the checkpointing-induced execution time normalized against the baseline as a function of r. As the figure shows, small write ranges yield comparable results. For larger ranges, in particular r greater than 32 kilobytes (i.e., 2^{15} bytes), however,the overhead increases for both

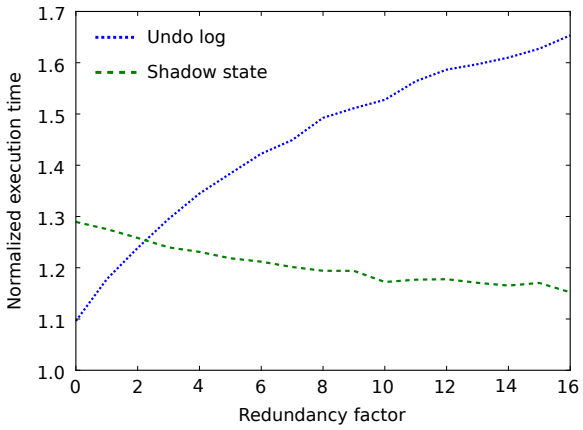

Figure 3: Normalized execution time for different redundancy factors.

schemes. We interpret this as a result of cache trashing in the L1 cache. Interestingly, this also makes the effect of duplicate writes more prominent, with our shadow state-based technique yielding around 15 % lower performance overhead for r set to 32 kilobytes. Further, for ranges larger than 8 megabytes (i.e., 2^{15} bytes), the overhead of both techniques drops again. We interpret this as a result of second-level cache trashing degrading the performance of the baseline, so that the relative overhead becomes smaller.

Second, we evaluated the two techniques against different degrees of write duplication. Figure 3 depicts the normalized execution time while varying the redundancy factor r and fixing the write range R to 128 kilobytes. As the figure shows, the undo log-based technique performs better only for $r = \{0, 1\}$ and it is increasingly outperformed by the shadow state-based approach for greater values of r. This confirms that our technique provides better performance for real-world programs that typically exhibit nontrivial duplicate write patterns.

5. CONCLUSION

While being a widely used technique in several system applications, in-memory checkpointing is still often deemed as impractical and inefficient for general use. In this paper, we compared a number of in-memory checkpointing techniques and argued that checkpointing can instead be implemented safely and efficiently, and even potentially deployed in production. To support our claim, we evaluated both traditional page-granular checkpointing techniques and more fine-grained compiler-based techniques. We also proposed a new compiler-based checkpointing technique based on a shadow state. Our solution relies on a tagmap to implement efficient fine-grained copy-on-write semantics between the shadow state and the primary state originally used by the program. Experimental results confirm that our approach provides better performance and memory guarantees than state-of-the art techniques, scaling up efficiently (and with bounded memory overhead) even in face of heavily-duplicated memory write patterns.

Ackowledgements

We would like to thank the anonymous reviewers for their comments. This work has been supported by European Research Council under grant ERC Advanced Grant 2008 – R3S3.

6. REFERENCES

[1] Apache benchmark (AB). `http://httpd.apache.org/docs/2.0/programs/ab.html`.

[2] hmmer. `http://hmmer.janelia.org`.

[3] lighttpd. `http://www.lighttpd.net/`.

[4] nginx. `http://nginx.net`.

[5] Giuffrida, C., Cavallaro, L., and Tanenbaum, A. S. We crashed, now what? In *Proc. of the Sixth Workshop on Hot Topics in System Dependability* (2010), pp. 1–8.

[6] Hursey, J., January, C., O'Connor, M., Hargrove, P. H., Lecomber, D., Squyres, J. M., and Lumsdaine, A. Checkpoint/restart-enabled parallel debugging. In *Proc. of the 19th European Message Passing Interface Conference* (2010), pp. 219–228.

[7] Kadav, A., Renzelmann, M. J., and Swift, M. M. Fine-grained fault tolerance using device checkpoints. In *Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* (2013), pp. 473–484.

[8] King, S. T., Dunlap, G. W., and Chen, P. M. Debugging operating systems with time-traveling virtual machines. In *Proc. of the USENIX Annual Tech. Conf.* (2005), p. 1.

[9] Lattner, C., and Adve, V. LLVM: a compilation framework for lifelong program analysis & transformation. In *Proc. of the Int'l Symp. on Code Gen. and Opt.* (2004), p. 75.

[10] Lenharth, A., Adve, V. S., and King, S. T. Recovery domains: an organizing principle for recoverable operating systems. In *Proc. of the 14th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* (2009), pp. 49–60.

[11] Nethercote, N., and Seward, J. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments* (New York, NY, USA, 2007), VEE '07, ACM, pp. 65–74.

[12] Portokalidis, G., and Keromytis, A. D. REASSURE: a self-contained mechanism for healing software using rescue points. In *Proc. of the 6th Int'l Conf. on Advances in Information and Computer Security* (2011), pp. 16–32.

[13] Sidiroglou, S., Laadan, O., Perez, C., Viennot, N., Nieh, J., and Keromytis, A. D. ASSURE: automatic software self-healing using rescue points. In *Proc. of the 14th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* (2009), pp. 37–48.

[14] Subhraveti, D., and Nieh, J. Record and transplay: partial checkpointing for replay debugging across heterogeneous systems. In *Proc. of the Int'l Conf. on Measurement and Modeling of Computer Systems* (2011), pp. 109–120.

[15] Xu, W., Bhatkar, S., and Sekar, R. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *Proc. of the 15th USENIX Security Symp.* (2006), pp. 121–136.

[16] Zhao, C. C., Steffan, J. G., Amza, C., and Kielstra, A. Compiler support for fine-grain software-only checkpointing. In *Proc. of the 21st Int'l Conf. on Compiler Construction* (2012), pp. 200–219.