Speculative Memory Checkpointing

Dirk Vogt  
Vrije Universiteit Amsterdam  
d.vogt@vu.nl

Armando Miraglia  
Vrije Universiteit Amsterdam  
a.miraglia@student.vu.nl

Georgios Portokalidis  
Stevens Institute of Technology  
gportoka@stevens.edu

Herbert Bos  
Vrije Universiteit Amsterdam  
herbertb@few.vu.nl

Andy Tanenbaum  
Vrije Universiteit Amsterdam  
ast@few.vu.nl

Cristiano Giuffrida  
Vrije Universiteit Amsterdam  
giuffrida@cs.vu.nl

ABSTRACT

High-frequency memory checkpointing is an important technique in several application domains, such as automatic error recovery (where frequent checkpoints allow the system to transparently mask failures) and application debugging (where frequent checkpoints enable fast and accurate time-traveling support). Unfortunately, existing (typically incremental) checkpointing frameworks incur substantial performance overhead in high-frequency memory checkpointing applications, thus discouraging their adoption in practice.

This paper presents Speculative Memory Checkpointing (SMC), a new low-overhead technique for high-frequency memory checkpointing. Our motivating analysis identifies key bottlenecks in existing frameworks and demonstrates that the performance of traditional incremental checkpointing strategies in high-frequency checkpointing scenarios is not optimal. To fill the gap, SMC relies on working set estimation algorithms to eagerly checkpoint the memory pages that belong to the writable working set of the running program and only lazily checkpoint the memory pages that do not. Our experimental results demonstrate that SMC is effective in reducing the performance overhead of prior solutions, is robust to variations in the workload, and incurs modest memory overhead compared to traditional incremental checkpointing.

Categories and Subject Descriptors

D.4.5 [Reliability]: Checkpoint/Restart

General Terms

Reliability, Algorithms

Keywords

Memory Checkpointing; Speculation; Error Recovery; Debugging; Backtracking; Reliability; Memory Management

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Middleware '15 December 07-11, 2015, Vancouver, BC, Canada
© 2015 ACM. ISBN 978-1-4503-3618-5/15/12...$15.00
http://dx.doi.org/10.1145/2814576.2814802.

1. INTRODUCTION

Memory checkpointing—the ability to snapshot/restore the memory image of a running process or set of processes—has recently gained momentum in several application domains. In automatic error recovery applications, memory checkpointing enables fast and safe recovery to known and stable program states [20, 22, 23, 32, 39, 53, 54, 57, 58, 62, 70]. In debugging applications, it enables users to efficiently navigate through several program states observed during the execution, while empowering advanced debugging techniques such as reverse/replay debugging [27, 34, 60, 61]. Memory checkpointing also serves as a key enabling technology for important first-class programming abstractions like software transactional memory [39], application-level backtracking [11, 76], and periodic memory rejuvenation [68].

Such application domains require very frequent checkpoints in real-world scenarios. For instance, automatic error recovery techniques rely on frequent checkpoints to mask failures to the clients [67]. This is typically accomplished by checkpointing the program state at every client request [22, 39]—or at carefully selected rescue points [33, 53, 58, 62]. In advanced debugging techniques, frequent checkpoints allow users to quickly navigate through arbitrary points in the execution history [33, 34]. Finally, first-class programming abstractions implemented on top of memory checkpointing, such as application-level backtracking, typically yield a very high checkpointing frequency by construction [11].

Traditional memory checkpointing techniques rely on commodity hardware—a strategy that provides superior deployability compared to instrumentation-based strategies [10, 14, 39, 40, 53, 64, 65, 70, 76]—to incrementally copy memory pages that were modified by the running program [11, 20, 21, 34, 37, 51, 54, 56, 58, 60, 63]. While incremental memory checkpointing is regarded as an efficient alternative to disk-based or full memory checkpointing [52], it still incurs nontrivial memory tracing costs for every taken checkpoint, resulting in relatively infrequent checkpoints used in practice.

In this paper, we present Speculative Memory Checkpointing (SMC), a new technique for high-frequency page-granular memory checkpointing. SMC seeks to improve upon current techniques to allow for very high-frequency checkpointing at a period that is below the one millisecond boundary, even making it possible to checkpoint every request in a highly loaded server. To fulfill this goal, SMC sets out to minimize the memory tracing costs of incremental checkpointing by eagerly copying the hot (frequently changing) pages, while lazily tracing and copying at first modification time only cold...
(infrequently changing) memory pages. Thus, SMC combines
the advantages of full memory checkpointing (efficient bulk
copies) with that of incremental memory checkpointing (copy
only when needed). The key challenge is to find the optimal
trade-off between eagerly copying too many memory pages—
that is, unnecessary memory copying costs—and copying an
insufficient number of pages which may result in unnecessary
memory tracing costs for every checkpoint.

To address this challenge, SMC relies on a general writable
working set (WWS) model [15] to detect the memory pages
that change most often—the ideal candidates for our specula-
copying strategy. To obtain fresh and accurate estimates,
our implemented SMC framework supports well-established
working set estimation (WSE) algorithms. In addition, we
complement our framework with GSpec, a novel writable
WSE algorithm specifically tailored to high-frequency mem-
ory checkpointing. GSpec follows a blackbox optimization
strategy inspired by genetic computing [45]. The latter ap-
proach provides SMC with a self-tuning and self-adapting
working set estimation strategy by design, which relies on no
program-specific parameters and ensures fresh and accurate
estimates across several different real-world workloads. This
is in stark contrast to traditional WSE algorithms, which,
while well established in several application domains such as
dynamic memory balancing [13,30,42,43,66,78], garbage
collection [26,69,72], virtual machine restore [73,74] and live
migration [71], are generally ill-suited to high-frequency mem-
ory checkpointing. In particular, these algorithms impose a
stringent performance-accuracy trade-off that typically re-
sults in a nontrivial overestimation of the real writable work-
set [7]. This is perhaps acceptable in many traditional
applications (e.g., dynamic memory balancing with sporadic
memory pressure), but leads to substantial overcopying, and
thus overhead, for SMC.

Contributions.
The contributions of this paper are fourfold. First, we
present an in-depth analysis of prior page-granular memory
checkpointing techniques, evidencing their direct and indirect
memory tracing costs. Our investigation uncovers important
bottlenecks for prior solutions in high-frequency checkpoint-
ing contexts and serves as a basis for our design. Second, we
present Speculative Memory Checkpointing (SMC), a new
 technique for high-frequency memory checkpointing based
on (several possible) WSE algorithms. Third, we introduce
GSpec, a novel WSE algorithm which draws inspiration from
genetic algorithms to speculatively copy memory pages that
are most likely to change in the next checkpointing interval.
Finally, we implemented and evaluated a kernel-module-
based SMC framework with support for GSpec and other
WSE algorithms, demonstrating its performance benefits in
high-frequency checkpointing scenarios. Our results demon-
strate that our WSE-based strategy is accurate, efficient,
robust to workload variations, and effectively reduces the
run-time overhead of high-frequency memory checkpointing
at the cost of modest memory overhead.

2. BACKGROUND

A straightforward way to implement process checkpoint-
ing involves freezing the execution and taking a snapshot
of memory by copying it [5,16,24,36,49,55]. Even though
this approach suffices in certain domains, like process mi-
gration, it is wasteful and slow in domains where frequent
checkpoints need to be made, as it requires the process to
stop for a significant amount of time and copies potentially
large amounts of data indiscriminately.

A more efficient strategy is to rely on incremental check-
pointing. Incremental checkpointing builds a checkpoint
gradually—minimizing the time that a process is suspended,
and reducing the amount of data to copy. We can gener-
ate incremental checkpoints in two ways. We can make a
full snapshot in the beginning, and then track and save all
modifications, so we can add them to the snapshot at the
next checkpoint [3,21,51,56,63]. To roll back, all mem-
ory is restored using the maintained snapshot. Alterna-
tively, we can do the inverse and copy only the data that are
modified after a checkpoint, right before they are overwrit-
ten [11,20,34,37,51,54,58,60]. To roll back we restore only
the overwritten data using their copies. We will refer to the
former solution as “copy new data” (it copies the new data at
checkpoint time), and the latter as “copy old data” (it copies
the old data prior to overwriting them).

Traditional incremental checkpointing mechanisms are usu-
ally page-granular, that is, a memory page is the smallest
data block copied (although more fine-grained techniques ex-
ist [17,39,53,70,76]). Below we discuss the core mechanisms
and techniques employed by these approaches.

Hardware dirty bit.
Incremental checkpointing techniques rely on dirty page
tracking. Modern memory management units (MMUs) in-
clude a dirty bit for each entry in the page tables maintained
by the operating system (OS), which is set by the hard-
ware when a page is written. The bit is used by the OS to,
for example, determine which pages need to be flushed to
disk. Directly using this dirty bit to detect modified pages
is potentially fast, but requires extensive changes to the OS
kernel [5,21,36,63] which is neither attractive, nor likely to
help deployability.

Soft dirty bit.
Linux also offers a soft dirty-bit mechanism, made available
to user space through the proc file system, which provides
the same functionality with HW dirty bits, albeit not as fast
(see Section 3).

Write Bit.
The write bit [63], also provided by the MMU, controls
whether a virtual memory page can be written. It is often
leveraged for checkpointing. For example, when the dirty bit
is missing, it is used to emulate the functionality. Briefly,
write protecting a page will generate faults on writes. By
capturing the faults, we identify the dirty pages and maintain
our own soft dirty bit.

Copy-on-write (COW) semantics.
“Copy old data” approaches that save memory pages on-
the-fly are in their majority utilizing the write bit and COW
semantics. The most well known use of COW is in the fork
system call in Linux. fork creates a new process, identical
to the parent process invoking it, but instead of duplicating
all memory pages, the two processes share the same pages
which are now marked as read-only and COW. When one
of them writes to a page, a fault is generated, causing the
kernel to create a copy of the page. User-space checkpointing
mechanisms are using fork to copy pages on-demand, but COW semantics can be also used directly from within the kernel, by setting the appropriate bits in the page table.

Page Checksums.
An alternative for determining dirty pages without relying on dirty bits involves periodically calculating the checksum of pages and comparing them over time. The precision of this approach is subject to the accuracy of the algorithm used for computing the checksums [46]. One could also compare the contents of individual memory pages directly [42], but this strategy is generally less space-efficient and more expensive due to poor cache behavior.

3. SMC
Checkpointing based on the write bit, which primarily includes approaches using COW, does not require changes within the kernel and can efficiently roll back, but suffers increasing overhead as the number of pages in a checkpoint grows. Besides the unavoidable cost of copying pages, handling page faults also induces overhead. Given a way to establish which pages are going to be modified after a checkpoint, we could avoid the page-faulting overhead and copy only the pages that need to be saved. This is the key idea behind Speculative Memory Checkpointing (SMC).

Knowing exactly which pages are going to be written after a checkpoint is a difficult problem, which is addressed by SMC through approximation, similar to working set estimation (WSE). Pages that are expected to receive writes are considered to be hot and not write-protected but eagerly copied when hitting a checkpoint. In “copy old data” approaches, they are copied and discarded on the next checkpoint, while in “copy new data” approaches, they are copied into the full memory snapshot. The speculative approach followed by SMC can be examined based on accuracy and performance.

Accuracy.
A speculative approach is accurate when it can continuously determine the pages that will be written during a checkpoint. Missing hot pages triggers page faults and degrades performance. We refer to such errors as undercopying. Respectively, marking rarely written pages as hot leads to more copying than needed, also degrading performance. We refer to these errors as overcopying.

Performance.
Three key factors affect the performance: the overhead of the algorithm that speculates the set of hot pages, the number of undercopying errors, and the number of overcopying errors. Obviously, a very accurate prediction algorithm can reduce the number of errors, but if that comes with an elevated cost, then it overshadows the lack of errors. Similarly, a large number of errors can make SMC more expensive than traditional incremental approaches (e.g., if none of the hot pages are actually written).

Design.
To guide the design of SMC, we carefully evaluated the impact of common operations performed by traditional incremental checkpointing techniques. Table 1 presents our results. An immediately evident result is the substantial overhead introduced by checkpointing strategies using COW pages from user space. This requires forking a new process, managing it, and terminating it when taking a new checkpoint, while the kernel takes care of copying a page when it is written. The latter is quite fast taking only 4016 CPU cycles, while forking, etc. requires 139,576 cycles (see lines 1 and 2 in Table 1). Shredding this overhead is an important factor for high-frequency checkpointing which involves more and potentially shorter (in duration) checkpoints. For this reason, our SMC framework bases its operations in a kernel module that exports checkpointing primitives to user space. A complete user-space solution would have otherwise incurred significantly larger overhead at runtime, mainly due to the cost of managing memory and the MMU bits [9].

To estimate the benefits from using SMC, we compare the time taken to perform a single write when checkpointing with the different incremental checkpointing strategies we described above (see lines 1, 3, and 6 in Table 1). Under (accurate) SMC, the page would just be copied once correctly placed in the writable pages hot set, and the write would complete normally. When using COW, the kernel would make a copy of the page, before the write completes. Finally, with soft dirty bits, the write completes normally but we then need to read the dirty bits to identify the updated page and save it. The process takes 492, 4016, and 16136 CPU cycles respectively. Note that in practice there are other costs involved with these strategies as well, like calculating the hot pages, marking all pages as COW in the beginning, and clearing the dirty bits (Table 1, line 6).

We notice that managing soft dirty bits can be very expensive, and it is preferable to use a page’s checksum to identify updated pages, when we are examining a small number of pages. Most importantly, the direct cost of saving a page when checkpointing is only a small part of the whole process, which involves many indirect costs, like fault handling, managing dirty bits, etc. As a result, a perfectly accurate speculation algorithm incurs eight times less overhead per-page, compared to COW (last of Table 1). We also establish that undercopying and overcopying errors do not cost the same, as the first will result in a COW (approx. 4016 cycles), while the latter leads to a wasted copy (approx. 492 cycles). Thus, on modern architectures, the cost for 1 undercopying error is comparable to 8 overcopying errors.

Table 1: Microbenchmarks that test the various operations performed by incremental checkpointing. The table lists the average number of CPU cycles consumed after running each test 1000 times.
Finally, a “copy old pages” approach is more favorable because it requires less memory space for each checkpoint (no full snapshot). Other than guiding the design and implementation of SMC, we later use these findings to derive the cost factors for our genetically-inspired GSpec WSE strategy.

4. FRAMEWORK OVERVIEW

Figure 1 depicts the high-level architecture of our SMC framework. To deploy SMC, users install a small kernel module (ksmc) and link their programs against a user-level library (libsmc). The library offers convenient memory checkpoint/restore primitives to programs and forwards all their invocations to ksmc through a fast and dedicated SMCall interface that requires no recompilation or restart of the running operating system kernel. Our kernel module can handle requests from a large number of programs in parallel and be safely unloaded when no longer needed, which ensures a fast and safe deployment of SMC. Also note that programs not using speculative checkpointing functionalities are unaffected by the presence of ksmc.

When a user program issues a memory checkpoint request via libsmc, our kernel module checkpoints the current memory image of the calling process and returns control to user space. This event marks the beginning of a new checkpointing interval, terminated only by the next checkpoint (or restore) request. The data (and metadata) associated with every checkpoint is maintained in an in-kernel journal by the core checkpointing component (CKPT) of ksmc. The journal stores a maximum predetermined number of memory pages that should be eagerly/lazily copied before returning control to user space. A copy of these pages is immediately stored in the current checkpoint, eliminating the need for explicit memory tracing mechanisms in the forthcoming checkpointing interval. All the other cold memory pages, in turn, are explicitly tracked and their data copied lazily at first modification.

4.1 Checkpointing Component

The checkpointing component implements the core memory checkpointing functionalities in the ksmc kernel module. Its operations and interface are deliberately decoupled from the main kernel as much as possible. Its internal structure is fully event-driven with a number of well-defined entry points. The main entry point provides user programs with access to a simple control interface via the libsmc library. Each user program can register itself with the checkpointing component—that is enter “SMC mode”—and specify the desired SMC configuration, including the speculation strategy to adopt and the memory regions to checkpoint. By default, the entire memory image is considered for checkpointing, but user programs may limit checkpointing operations to specific memory areas—for example, to implement an SMC-managed heap for a specialized memory allocator that supports application-level backtracking. The control interface also allows primitives to checkpoint/restore the predetermined memory areas or reset/collect SMC statistics—for example, average number of pages copied eagerly/lazily per checkpointing interval.

For each process in SMC mode, ksmc maintains a process descriptor—with process-specific configurations—a set of memory area descriptors, and a journal of checkpoint descriptors. Each checkpoint descriptor maintains a number of page entries with the address and a copy of the original page to restore the saved memory image starting from the next checkpoint in the journal—or the current memory image in case of the most recent checkpoint.

When a process enters SMC mode, ksmc creates new process and memory area descriptors as well as an implicit first checkpoint using a full-coverage memory tracing strategy akin to incremental checkpointing. This is done by write-protecting the page table entries associated with all the memory pages in the virtual address space of the calling process and intercepting all related page faults to save a copy of the soon-to-be modified pages.

Page fault events represent the second important entry point in ksmc, allowing SMC’s memory tracing strategy to create new page entries in the current checkpoint descriptor, notify the speculation component of the event, and allow the kernel to simply copy and unprotect the faulting page and resume user execution. To avoid slowing down the normal execution of the main kernel’s page fault handler, ksmc supports efficient lookups of process and memory area descriptors to quickly return control to the main kernel if the last faulting page is not currently being tracked by SMC. A similar strategy is used when intercepting process termination events—the third entry point in ksmc—which the checkpointing component tracks to automatically garbage collect all the descriptors and page entries associated with each terminating SMC process.

When a new checkpoint operation is requested, ksmc marks the current checkpoint descriptor as completed—note that this is always possible even at the first application-requested checkpoint by construction—and creates a new checkpoint descriptor for the forthcoming checkpointing interval. It subsequently iterates over the page entries in the last checkpoint descriptor and requests the speculation component to...
determine the optimal copying strategy for each page. For each memory page subject to an eager copying strategy, \texttt{ksmc} immediately creates a new page entry in the new checkpoint descriptor. For other pages, \texttt{ksmc} write-protects the page and delegates the checkpointing operations to page fault time.

When a new restore operation is requested, \texttt{ksmc} walks the checkpoint descriptors in reverse order—starting from the current one and ending with the one requested by the user—and incrementally restores all the contained page entries. It subsequently evicts all the visited checkpoint descriptors (and associated entries) from the journal and notifies the speculation component of the event.

### 4.2 Speculation Component

The speculation component enhances the basic incremental checkpointing strategy implemented by the standalone checkpointing part with a working set estimation-driven speculative checkpointing technique at the beginning of every checkpointing interval. While currently integrated in \texttt{ksmc}, the speculation component is strictly decoupled from the checkpointing component and provides a generic speculation framework suitable for both user-level and kernel-level checkpointing solutions. The speculation component requires the external checkpointing solution to provide a number of platform-specific callbacks, including memory allocation, debugging, and configuration primitives. In SMC, our kernel module implements all the relevant callbacks suitable for kernel-level execution.

Internally, our speculation component shadows many of the data structures described in the previous subsection—descriptors and page entries—but also supports writable working set contexts for the benefit of the individual speculation strategies implemented in our framework. Each context stores all page entries associated with the current writable working set, which our speculation component uses to determine the memory pages subject to our eager copying strategy when initializing a new checkpoint descriptor. The current working set context is established at the beginning of every checkpointing interval based on user-defined policies.

Each speculation strategy has unrestricted read and write access to the current writable working set context and can register hooks to manipulate the context for all the events contained by our checkpointing module: page fault, checkpoint, restore, etc. The most conservative speculation strategy would simply produce empty writable working sets never populated with any page entries, an approach that would effectively degrade SMC to traditional incremental checkpointing. More effective speculation strategies, including our genetic speculation and other more traditional working set estimation strategies, are discussed in the following sections.

### 5. SPECULATION STRATEGIES

In the course of this work, we have considered a number of speculation strategies for SMC, drawing from classic working set tracking techniques and black box optimization algorithms. We now discuss these strategies in more detail.

#### 5.1 Classic WSE Strategies

**Scanning-based techniques.**

Scanning-based strategies periodically scan all the memory pages of a running process and determine the current writable working set from the recently modified pages. Scanning-based strategies are generally too expensive for short scanning intervals—strategies involving lightweight dirty page sampling have suggested using intervals of around 30 seconds [66]—due to high costs associated with frequent reference bit manipulation. The latter also suffers from the deployability limitations evidenced in Section 2. These shortcomings hinder the applicability of scanning-based strategies to high-frequency SMC.

**Active-list-based techniques.**

Active-list-based techniques divide all memory pages into two lists: \texttt{active} and \texttt{inactive}. On first access, pages are put on the active list, which are considered hot, that is eagerly copied at the beginning of a new checkpoint interval. On the contrary, inactive pages are copied on demand triggering a COW event. We implemented two active-list-based techniques, \texttt{Active-RND} and \texttt{Active-CKS}, which mainly differ in their active list eviction strategy.

\texttt{Active-RND} relies on dynamically determining the size of the WWS through periodic sampling. This is done by write-protecting the whole address space during the sampling runs, whereas the WWS size is calculated as the running average of the number of pages accessed during these runs. Whenever the active list has reached the estimated size and a new page faults in, \texttt{Active-RND} randomly evicts a page from the list. We chose a random page replacement strategy over other well known page replacement algorithms, like FIFO or the LRU-like CLOCK algorithm [12] and its variations [8,29], because the latter either performed significantly worse in early experiments (FIFO), or require dirty page tracking or page-table entry reference-bit manipulation.

\texttt{Active-CKS} relies on the observation that copying and calculating a checksum is still significantly cheaper than copying a page in COW fashion. While pages also enter the active list when first accessed, \texttt{Active-CKS} will only evict a page when its checksum did not change during the last N checkpoint intervals, with $N = 5$ (the top performer in our experiments).

**Oracle.**

The \texttt{Oracle} strategy considers all the pages that will be accessed during the next interval as \texttt{hot}. Since this strategy is directly based on knowledge of the future (and due to the lack of a time machine), SMIC implements only an optimistic approximation of this algorithm based on profiling data. For each checkpoint $c$, it logs the number of modified pages $N_c$ offline and pre-copies $N_c$ dummy pages online. While this strategy lacks correctness, it gives a good estimate of what performance improvements can be expected by SMC given an ideal speculation strategy.

#### 5.2 Genetic Speculation

Our genetic speculation strategy—or \texttt{GSpec}—aims to estimate the current writable working set using a methodology inspired by genetic algorithms [45]. Such algorithms provide a widely employed blackbox optimization method for problems with a large set of possible solutions. Genetic algorithms are inherently self-tuning and self-adapting, matching the stringent accuracy and adaptivity requirements of high-frequency memory checkpointing. Inspired by biological evolution, such algorithms allow candidate solutions, also called individuals, to compete against each other. In our case an individual represents a set of \texttt{hot} pages, whereas the information of
which pages are considered to be hot is encoded in the individuals’ chromosomes—typically represented by a bit string. All current individuals form a population. They are periodically evaluated using a cost function, which measures their respective fitness. After each evaluation period, a new generation of the population is formed by selecting the most fit individuals (selection) and recombining their chromosomes (crossover). Over time the population’s solutions are meant to converge to a minimum of the cost function.

**Chromosome representation and cost function.**

GSpec maintains a global list of all the memory pages currently known by the algorithm, ordered by page appearance. Each individual’s chromosomes represent a set of candidate memory pages, stored in a WWS bitmap—a generic bit string. If a bit in the WWS bitmap is set, the corresponding page is marked as hot, that is, part of the writable working set—otherwise the page is considered cold. Whenever a memory page is marked as cold by all the individuals, the page is removed from the global page list, that is, the algorithm forgets about the page.

GSpec models its cost function based on the memory copying costs caused by a given individual. Each memory page copied during a checkpointing interval contributes to the total cost associated with the current individual. Memory pages copied lazily are weighted more to reflect the memory tracing costs associated with the COW semantics. Although weighted less, pages copied eagerly are still assigned a nonzero cost, preventing GSpec from greedily copying all the known memory pages. The cost values are directly derived from our analysis in Table 1, with a value of 1 and 8 accounted for every page copied eagerly and lazily, respectively.

**Speculation phase.**

The population has a predetermined size of $N = 5$ individuals, a standard value adopted in prior work on micro-genetic algorithms to ensure an efficient and fast-converging implementation [35]. For each checkpointing interval, GSpec selects one individual from the population in a round-robin fashion and requests the checkpointing component to copy all the hot pages eagerly. The costs for the eagerly copied pages (1) are attributed to the current individual. For each page that faults during the current interval, the respective cost (8) is assigned to the current individual. If a faulting page is currently not in GSpec’s global list, it is added to the WWS bitmap of the current individual with unbiased probability $p = 0.5$.

**Forming a new generation.**

After every individual had its turn, GSpec computes a new generation of individuals to evolve the current population. Each new individual thereby inherits the combined genetic information from selected parent individuals of the current population. Common selection strategies adopted by traditional genetic algorithms are tournament selection [44] and roulette wheel selection [41].

Both strategies select two parent individuals $P_1$ and $P_2$ to generate each individual in the new generation. GSpec implements a roulette wheel selection strategy, which yields a simpler implementation and is known to accurately model many real-world problems [18]. This strategy stochastically selects individuals with a higher probability for lower cost values. GSpec, achieves that by keeping track of the lowest cost $C_{\text{min}}$, in the population and selecting a random individual $I_R$ with a cost $C_R$ as parent with a probability of $p = C_{\text{min}} / C_R$. This process is repeated until two parents are assigned to each individual of the new generation.

Once the parent individuals for the next generation have been selected, GSpec mixes the writable working sets of each parent pair $P_1$ and $P_2$ to generate each new individual. This operation is commonly referred to as crossover, with two dominant strategies used in the literature: n-point crossover and uniform crossover [45].

GSpec opts for a uniform crossover strategy, which generally yields an unbiased and more efficient exploration of the search space in practice [59]. This strategy selects each chromosome bit from $P_1$ (instead of $P_2$) with a predetermined probability $p$. GSpec selects the individual chromosome bits with the standard probability $p = 0.5$ commonly adopted in prior work in the area [59].

To avoid local minima, genetic algorithms occasionally mutate the recombined chromosomes after the crossover phase. GSpec implements a simple bit-flip mutation strategy, flipping the individual chromosome bits with a predetermined probability $p$. In the current implementation, GSpec opts for a bit-flip mutation probability $p = 0.01$, again, a value commonly adopted in the literature [45].

6. IMPLEMENTATION

We implemented SMC in an architecture-independent loadable kernel module for the Linux kernel. Our implementation initially targeted Linux 3.2, comprising a total of 2227 LOC\(^1\) for the checkpointing component and 1466 LOC for the speculation component—implementing our genetic speculation strategy and the alternatives (Active-RND, Active-CKS, and Oracle) considered in the paper. We subsequently tracked all the mainline Linux kernel changes until the recent 3.19 kernel release and, despite the fast-paced evolution of the Linux kernel interfaces, we added a total of only 20 extra LOC to our original implementation. This acknowledges our efforts into decoupling SMC from the mainline kernel, relying on a minimal and stable set of kernel APIs—currently a total of 45 common kernel routines for memory allocation, page table manipulation, interfacing, and synchronization.

Driven by the same principles, we implemented SMC’s page fault interception mechanism using kernel probes [4], the standard Linux kernel instrumentation facility which allows modules to dynamically break into any kernel routine—handle_mm_fault, for our purposes—in a safe and nondisruptive fashion. We adopted the same mechanism to intercept process termination events—the do_exit and do_execve kernel routines—and automatically perform all the necessary process-specific cleanup operations. To implement SMC’s dedicated SMCall interface, in turn, we allowed our kernel module to export a new kernel parameter accessible via the sysctl system call from user space. Our user-level libsmc library—implemented in one header file of 114 LOC—hides the internals of the syscall-based communication protocol with the kernel module to user programs.

To support common request-oriented recovery models with minimal user effort [22, 39], SMC is also equipped with a profiler that automatically identifies suitable checkpointing locations at the top of long-running request loops and a trans-

\(^1\)Source lines of code reported by David A. Wheeler’s SLOC-Count.

---

1. Source lines of code reported by David A. Wheeler’s SLOC-Count.

---
We evaluated our SMC framework implementation on a workstation running Linux v3.12.36 (x64) and equipped with a dual-core Intel Pentium G6950 2.80 GHz processor and 16 GB RAM. To evaluate the real-world impact of SMC, we selected five popular server programs—a common target for memory checkpointing applications in prior work in the area [20, 53, 54, 58, 70]—and allowed our deployed SMC framework to checkpoint the memory image of their worker processes at every client request, following the common request-oriented checkpointing model [22, 39]. For our analysis, we considered the three most popular open-source web servers—Apache httpd (v.2.2.23), nginx (v0.8.54), and lighttpd (v.1.4.28)—a popular RDBMS server—PostgreSQL (v9.0.10)—and a widely used DNS server—BIND (v9.9.3).

To evaluate the impact of SMC on our server programs, we performed tests using the Apache benchmark (AB) [1] (web server programs), the Sysbench benchmark [6] (PostgreSQL), and the queryperf tool [2] (BIND). To investigate the SMC-induced performance impact in memory-intensive application scenarios and its sensitivity to the checkpointing frequency, we further evaluated our solution on hammer, a popular scientific benchmark. Finally, in order to directly compare SMC with recent instrumentation-based memory checkpointing techniques [65] that naturally do not cover uninstrumented shared libraries, we focus our evaluation on a program-only analysis and briefly report on the performance impact of shared libraries when extending the checkpointing surface to the entire address space.

To prepare our test programs for request-oriented memory checkpointing, we allowed our dynamic profiler to automatically identify all the long-running request loops in preliminary test runs and instrument the top of each loop with a checkpoint call into the libsmc library. We configured all of our test programs with their default settings and instructed the Apache httpd web server to serve requests with the prefork module with 10 parallel worker processes. We repeated all our experiments 11 times (with negligible variations) for each of the speculation strategies presented in Section 5 and report the median.

Our evaluation focuses on five key questions: (i) Performance: Does SMC yield low run-time overhead in high-frequency memory checkpointing scenarios? (ii) Checkpointing frequency impact: How sensitive is SMC performance to the checkpointing frequency? (iii) Accuracy: What is the accuracy of our WSE-based speculation strategies? (iv) Memory usage: How much memory does SMC use? (v) Restore time: Does SMC yield low restore time increase?

### 7.1 Performance

To evaluate the run-time performance overhead of SMC on real-world applications, we tested our server programs running in “SMC mode” and compared the resulting throughput against the baseline. To benchmark our web server programs, we configured the Apache benchmark to issue 25,000 requests through the loopback device, using 10 parallel connections, 10 requests per connection, and a 1KB file. To benchmark BIND, we configured the queryperf tool to issue 500,000 requests for a local resource using 20 parallel threads. To benchmark PostgreSQL, we configured the Sysbench benchmark to issue 10,000 OLTP requests using 10 parallel threads and a read/write workload. In all our experiments, we verified that our programs were fully saturated by the benchmarks.

Figure 2 shows the SMC-induced throughput degradation for our server programs, as observed during the execution of the program and shared libraries.

---

**Table 2: Throughput degradation (geomean) induced by different SMC speculation strategies (program only).**

<table>
<thead>
<tr>
<th>Server</th>
<th>Requests per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>20,887</td>
</tr>
<tr>
<td>httpd</td>
<td>28,002</td>
</tr>
<tr>
<td>lighttpd</td>
<td>22,602</td>
</tr>
<tr>
<td>nginx</td>
<td>20,089</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>20,887</td>
</tr>
<tr>
<td>BIND</td>
<td>30,848</td>
</tr>
</tbody>
</table>

**Table 3: Number of requests per second handled by our server programs (baseline, no checkpointing).**

<table>
<thead>
<tr>
<th>Server</th>
<th>Requests per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>20,887</td>
</tr>
<tr>
<td>httpd</td>
<td>28,002</td>
</tr>
<tr>
<td>lighttpd</td>
<td>22,602</td>
</tr>
<tr>
<td>nginx</td>
<td>20,089</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>20,887</td>
</tr>
<tr>
<td>BIND</td>
<td>30,848</td>
</tr>
</tbody>
</table>

---

Figure 2: Throughput degradation induced by different SMC speculation strategies (program only).
of our macrobenchmarks. The absolute number of requests handled by the individuals servers without checkpointing can be found in Table 3. As expected, our speculation strategies generally yield a lower run-time performance overhead than traditional COW-style incremental checkpointing (COW in Figure 2) implemented by our checkpointing component in absence of any speculation strategy—note that our COW-based implementation is already much faster than traditional fork-based implementations used in much prior work. Compared to COW, our speculation strategies reported an average (geometric mean) overhead reduction of 4.44-14.24 percentage points (p.p.). GSpec, in particular, was consistently the top performer across all our server programs (14.24 p.p. average overhead reduction compared to COW, geometric mean). In some scenarios, the GSpec-reported improvements over traditional memory checkpointing are more significant—for example, 18 p.p. overhead reduction for nginx—due to higher checkpointing frequency and a more stable working set.

Active-RND is the second best-performing strategy—with an average performance overhead of 34.2% compared to GSpec’s 30.8% and COW’s 44.9% (geometric mean)—but we experienced its performance rapidly dropping as we deviated from the best-performing RND-N value. We found that altering GSpec’s core parameters from the values commonly adopted in the genetic algorithms literature, in contrast, had only marginal (if any) performance impact. Furthermore, Active-CKS reported the worst speculation performance, with an average overhead of 35.02% across all our server programs. Finally, the Oracle strategy reported, as expected, a consistently lower overhead compared to all our speculation strategies (15.65% geometric mean), providing a promising theoretical lower bound for the performance overhead of any future SMC strategy. Encouragingly, GSpec consistently follows the Oracle strategy across all our server programs and its overhead even comes relatively close to the Oracle for programs with a fairly stable writable working set—for example, 32.1% compared to 17.83% on BIND.

We now compare our results with recent compiler-based memory checkpointing techniques (LMC) [65]. For servers with good speculation performance, SMC performance is comparable or better than that of compiler-based techniques (e.g., GSpec’s 12.9% vs. LMC’s 15.3% on Apache httpd). When speculation is less effective, compiler-based techniques tend to outperform SMC (e.g., GSpec’s 56.9% vs. LMC’s 32.2% on PostgreSQL). On average, SMC induces an extra performance impact of 10-15 p.p. across programs. Nevertheless, we found our results very encouraging, given that unlike compiler-based techniques, SMC’s checkpointing strategy is source code-agnostic and can thus operate on legacy binaries.

Finally, Table 2 shows that, when extending the checkpointing point to the entire address space, we observed an additional performance impact (due to shared library checkpointing) in the range of 12-15 p.p. We also note that the general trend is consistent and speculation equally effective, e.g., 17 p.p. average performance improvement with GSpec.

### 7.2 Checkpointing Frequency Impact

In the previous subsection, we investigated the SMC-induced performance impact on server request-oriented memory checkpointing, a scenario which, in our experiments, yielded a checkpointing frequency of 9K-20k checkpoints/sec across all our server programs.

To investigate the frequency impact, we evaluated our best-performing (GSpec and Active-RND) speculation strategies on hmmer, a memory-intensive scientific benchmark. For our purposes, we instrumented hmmer to invoke the checkpoint call into the libsmc library at each task loop iteration, and forced our library to forward the calls to ksnc only every F predetermined invocations. This allowed us to emulate different checkpointing frequencies, ranging from roughly 700 checkpoints/sec—when checkpointing at every iteration—to 40 checkpoints/sec—when checkpointing every 16 iterations.

Figure 3 depicts the SMC-induced run-time overhead on hmmer across all the checkpoint frequencies considered. Results shown in the figure provide a number of interesting insights. First, checkpointing at every loop iteration yields comparable results to our performance experiments on servers program, with GSpec (2.6%) and Active-RND (2.4%) improving over COW (5.8%). Finally, as expected, for lower memory checkpointing frequencies, the memory tracing costs incurred by traditional COW become more amortized throughout the execution and the performance benefits of SMC become less evident—e.g., less than 1 p.p. overhead reduction when checkpointing every 16 iterations.

### 7.3 Accuracy

To evaluate the accuracy of our speculation strategies, we implemented support for a “meta speculation” strategy in SMC. The meta speculation strategy relies only on standard COW-style incremental checkpointing, but also transparently exposes each observed page-fault event only to the other speculation strategies that have assumed the faulting page not to be in the current writable working set. This allows all strategies to operate normally, while the meta speculation strategy gathers accuracy statistics based on the number of memory pages dirtied by the running program.

Table 4 reports the accuracy statistics produced by the meta speculation strategy when analyzing our server programs. Statistics are gathered on a per-checkpoint interval basis during the execution of our macrobenchmarks and averaged using the mean. The number of mispredicted pages (MP), that is, the sum of overcopied pages (OP) and undercopied pages (UP), represents the total number of dirty memory pages that a given speculation strategy failed to predict according to its internal writable working set estimates. The weighted mispredicted pages (WMP), in turn, weigh undercopied pages—inducing COW events—more than...
overcopied pages, also taking into account the additional costs for computing the checksums in Active-CKS. WMP is computed as $WMP = C_{OC} \cdot N_{OC} + C_{UC} \cdot N_{UC}$, with $N_{OC}/N_{UC}$ and $C_{OC}/C_{UC}$ being the number and cost factor of overcopied/undercopied pages (respectively). Based on the numbers in Table 1, we assume $C_{OC} = 1$ for GSpec and Active-RND, and $C_{OC} = 2.5$ for Active-CKS. We also assume $C_{UC} = 8$ for all our strategies.

The number of unweighted mispredictions (MP) alone seems to suggest that Active-CKS, with 8.7 mispredicted pages on average, is together with Active-RND the best speculation strategy. However, its high accuracy is overshadowed by the checksumming costs ($WMP = 25.7$), especially as Active-CKS tends to overcopy ($OP = 8.0$) nearly as much as GSpec ($OP = 10.5$).

GSpec, in turn, reported 21.2 weighted mispredicted pages on average, outperforming the runner-up Active-RND—that is, 24.3 WMP on average—with a similarly efficient working set estimation implementation. This result acknowledges the effectiveness of GSpec’s cost-driven speculation strategy empowered by genetic algorithms compared to the random strategy provided by Active-RND. This is also reflected in the lower WMP values reported by GSpec across all our server programs.

Overall, we can observe that the WMP predicts the performance results of the respective speculation mechanisms well and further shows the importance of carefully balancing accuracy and efficiency of the underlying working set estimation algorithm when designing a speculation strategy.

### 7.4 Memory Usage

As checkpoints also include overcopied pages, the accuracy of a speculation strategy has a direct impact on checkpoint size and overall memory usage. In our experiments, we observed our speculation strategies introducing an average checkpoint size increase compared to COW of 44%-66% across all our server programs (geometric mean). Programs with a larger writable working set—for example, PostgreSQL—or more diverse memory access patterns across checkpointing intervals—for example, lighttpd—yield the highest checkpoint size compared to traditional incremental checkpointing across all our speculation strategies, with a maximum increase of 133% and 107% (respectively). Programs with more rigorous memory usage, in turn—that is, Apache httpd and BIND—yield a more limited amount of overcopying, with a maximum increase of only 19% and 12% across all our speculation strategies. GSpec’s checkpoint size increases are comparable to the other speculation strategies, only occasionally yielding higher increases that reflect a more aggressive overcopying strategy—for example, for Apache httpd. Even nontrivial increases in checkpoint sizes (e.g., 133% for PostgreSQL), however, do not typically result in significant increases in physical memory usage overhead compared to COW. To quantify the latter, we computed the average overhead induced by memory checkpointing on the Resident Set Size (RSS).

Using COW, we reported a worst-case RSS overhead induced by memory checkpointing of only 3.6% (lighttpd). The same scenario resulted in a maximum RSS overhead of 7.6% across all our speculation strategies. This, thereby, translates to a maximum RSS increase of only 4 p.p. induced by SMC.

### 7.5 Restore Time

Overcopying errors introduce an excessive number of pages in a checkpoint, thus also increasing the restore time. For the program (PostgreSQL) with the largest checkpoint size (28 pages), this results in roughly doubling the number of pages to be restored (58 pages). The worst-case relative increase across our server programs is, thus, small, with only 558 extra CPU cycles required to restore 58 pages (2840 cycles) instead of 28 pages (2282 cycles)—measured using a synthetic microbenchmark. As the total time is still small and restore operations are generally much less frequent than checkpoint operations (e.g., at error recovery time), we believe this additional cost to be negligible in practice.

### 8. RELATED WORK

#### Incremental checkpointing techniques.

Several incremental checkpointing variations and applications are described in literature, with implementations at the user level [3,16,20,51,54–56,60], kernel level [5,21,24,36,37,49,58,63], or virtual machine monitor level [11,34,50,67]. User-level techniques can be easier to deploy, but incur significant run-time overhead because memory management at the application-level is more costly than from within the kernel [9]. Other user-level approaches, rely on compiler-based program instrumentation [10,14,39,40,64,65,76], which require source-code and recompilation of the target programs and all used libraries. Using dynamic instrumentation at the binary level [53,70] can provide checkpointing for unmodified binaries but incurs even higher performance overheads. Finally, approaches that require hardware support are not practical on commodity systems [17]. For this reason, SMC adopts a kernel-only checkpointing strategy implemented in a small kernel module, allowing for easier deployment compared to prior kernel-level work relying on dedicated kernel patches [5,21,36] or complex modules implementing fully-blown memory containers [49,58]. Furthermore, in

<table>
<thead>
<tr>
<th></th>
<th>OP</th>
<th>UP</th>
<th>MP</th>
<th>WMP</th>
<th></th>
<th>OP</th>
<th>UP</th>
<th>MP</th>
<th>WMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache httpd</td>
<td>7.9</td>
<td>1.5</td>
<td>9.4</td>
<td>19.7</td>
<td>6.1</td>
<td>2.9</td>
<td>9.0</td>
<td>29.5</td>
<td></td>
</tr>
<tr>
<td>nginx</td>
<td>10.1</td>
<td>0.6</td>
<td>10.8</td>
<td>6.8</td>
<td>2.5</td>
<td>0.8</td>
<td>3.3</td>
<td>8.7</td>
<td></td>
</tr>
<tr>
<td>lighttpd</td>
<td>22.7</td>
<td>3.2</td>
<td>25.9</td>
<td>48.3</td>
<td>10.5</td>
<td>6.1</td>
<td>16.6</td>
<td>59.6</td>
<td></td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>30.0</td>
<td>4.7</td>
<td>34.8</td>
<td>68.0</td>
<td>19.2</td>
<td>6.3</td>
<td>25.6</td>
<td>70.4</td>
<td></td>
</tr>
<tr>
<td>BIND</td>
<td>2.4</td>
<td>0.9</td>
<td>3.3</td>
<td>9.8</td>
<td>3.3</td>
<td>0.6</td>
<td>3.9</td>
<td>7.8</td>
<td></td>
</tr>
<tr>
<td>geomean</td>
<td>10.5</td>
<td>1.6</td>
<td>12.5</td>
<td>21.2</td>
<td>6.3</td>
<td>2.2</td>
<td>8.7</td>
<td>24.3</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Accuracy of the different SMC speculation strategies, with the average numbers of overcopied pages (OP), undercopied pages (UP), mispredicted pages (MP), and weighted mispredicted pages (WMP).
stark contrast to SMC, these techniques make no attempt to eliminate direct and indirect memory tracing costs in high-frequency memory checkpointing scenarios.

**Checkpointing optimizations.**

A common trend in prior work is to explore strategies to reduce the amount of checkpointed data. Some approaches propose checkpoint compression [28, 40], others rely on block-level checksumming [19, 46] to improve the granularity of incremental checkpointing techniques [3, 19, 46, 51, 56, 63], or, seek to discard redundant memory pages from the checkpointed data [25, 47, 50]. These approaches are well-suited to space-efficient process checkpointing on persistent storage, but are generally less useful to improve the memory checkpointing performance. SMC demonstrates that, in high-frequency memory checkpointing scenarios, memory over-copying can actually be beneficial to minimize the impact of indirect costs on the run-time performance.

Researchers also have explored program analysis techniques to select optimal checkpointing locations [40] or checkpointed data [14, 22, 32]. While complementary to our work, these techniques may help select checkpointing intervals with minimal working set size or provide useful heuristics to improve the accuracy of our working set estimation algorithms. We plan to explore the impact and the synergies between program analysis techniques and SMC in our future work.

Finally, other researchers have considered prediction-based strategies to improve memory checkpointing techniques. Nicolae et al. [48] propose predicting the order of memory pages modified within the next checkpointing interval to prioritize data to save on persistent storage in an asynchronous fashion. Also, their prediction strategy is tailored to reducing the number of copy-on-write events—each memory page is write-protected until asynchronously flushed to persistent storage. Unlike SMC, however, their focus is on reducing copy-on-write events to minimize memory usage and their prediction strategy is only effective in asynchronous checkpointing scenarios. Other researchers have proposed combining copy-on-write semantics with dirty page tracking—using dirty bits [67] or memory diffing [42]—to predict (and precopy) the pages modified at the next checkpointing interval. Their prediction strategy, however, is limited to consecutive checkpointing intervals—which reduces the overall prediction accuracy—and relies on expensive tracking mechanisms in high-frequency checkpointing scenarios—which reduces the overall performance. SMC, in contrast, generalizes these simple prediction strategies to the writable working set model, with a larger window of observation and stronger performance-accuracy guarantees.

**Working set estimation.**

Researchers have investigated working set estimation algorithms for a broad range of application domains, ranging from garbage collection [26, 69, 72], dynamic memory balancing [13, 30, 42, 43, 66, 78], and efficient memory management in general [79], to fast program startup [31], VM migration [71], and page coloring problems [75]. To our knowledge, however, SMC represents the first application of working set estimation algorithms to the memory checkpointing domain. Prior work on working set-driven restore of checkpointed virtual machines [73, 74] comes conceptually close, but, in such context, the working set estimation is performed relatively infrequently and offline—at checkpointing time—and the information gathered only later used to efficiently prefetch data from persistent storage—at restore time. SMC, in contrast, relies on online WSE algorithms that assist and exploit synergies with high-frequency memory checkpointing techniques in real time.

Working set size estimation techniques rely either on dirty page sampling [66, 75], monitoring memory statistics exported by the operating system [13, 26, 43, 72], or incrementally constructing LRU-based miss ratio curves (MRC) [30, 42, 69, 71, 77–79]. The latter generally provide the most accurate working set estimation method, but their most natural implementation requires expensive memory tracing mechanisms. More efficient implementations adopt an intermittent MRC tracking strategy that closely follows the phase behavior of common real-world programs [77] or rely on working set tracking to avoid tracing frequently accessed pages [69, 78, 79], typically at the cost of reduced accuracy [7].

However, traditional working set tracking techniques impose important performance and deployability limitations when applied to high-frequency memory checkpointing. Our genetically-inspired blackbox optimization algorithm, in turn, seeks to minimize the ad-hoc tuning effort generally required by prior techniques, automatically adapting the estimates to different workloads and matching the high accuracy and responsiveness required in high-frequency memory checkpointing scenarios.

9. CONCLUSION

Traditional incremental memory checkpointing is generally perceived as sufficiently fast for several typical real-world programs. In this paper, we challenged this common perception in the context of high-frequency memory checkpointing, by demonstrating that “hidden” costs generally deemed marginal in periodic checkpointing solutions significantly increase the run-time overhead when checkpoints are frequent. To substantiate our claims, we presented an in-depth analysis of the direct and indirect memory tracing costs associated with incremental checkpointing and uncovered limitations of prior frameworks in high-frequency checkpointing scenarios.

To address such limitations, we presented SMC, a new low-overhead technique suitable for high-frequency memory checkpointing. To minimize the direct costs associated with the checkpointing activity, our SMC framework relies on non-intrusive kernel-level specialization implemented in a loadable kernel module. To minimize the indirect costs associated with the checkpointing activity, our framework relies on algorithms for estimating the writable working set to copy speculatively those memory pages that are most likely to change in the next checkpointing interval, and in so doing reduce the memory tracing surface required by traditional incremental checkpointing.

We also demonstrated that our genetically-inspired blackbox optimization algorithm (GSpec) provides an effective working set estimation strategy for SMC, continuously adapting the working set to the workload driven by only program-agnostic cost factors. This strategy provides better accuracy, performance, and self-tuning guarantees than traditional working set estimation techniques. Overall, our experimental results show that SMC is both time- and space-efficient in the practical cases of interest, demonstrating that low-overhead high-frequency memory checkpointing is a realistic option and opening up opportunities for new programming abstractions empowered by fast checkpointing techniques.
Acknowledgments

We would like to thank the reviewers for their comments. This work was supported by the European Commission through project H2020 ICT-32-2014 “SHARCS” under Grant Agreement No. 644571 and by the European Research Council through project ERC-2010-StG 259108 “Rosetta”.

10. REFERENCES

