AI-Ckpt: Leveraging Memory Access Patterns for Adaptive Asynchronous Incremental Checkpointing

Bogdan Nicolae, Franck Cappello

▶ To cite this version:

Bogdan Nicolae, Franck Cappello. AI-Ckpt: Leveraging Memory Access Patterns for Adaptive Asynchronous Incremental Checkpointing. HPDC '13: 22th International ACM Symposium on High-Performance Parallel and Distributed Computing, Jun 2013, New York, United States. pp.155-166, 10.1145/2462902.2462918 . hal-00809847

HAL Id: hal-00809847
https://inria.hal.science/hal-00809847

Submitted on 10 Apr 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

AI-Ckpt: Leveraging Memory Access Patterns for Adaptive Asynchronous Incremental Checkpointing

Bogdan Nicolae
IBM Research
Dublin, Ireland
bogdan.nicolae@ie.ibm.com

Franck Cappello
Joint Laboratory for Petascale Computing
INRIA, France
University of Illinois at Urbana-Champaign, USA
fci@lri.fr

ABSTRACT

With increasing scale and complexity of supercomputing and cloud computing architectures, faults are becoming a frequent occurrence, which makes reliability a difficult challenge. Although for some applications it is enough to restart failed tasks, there is a large class of applications where tasks run for a long time or are tightly coupled, thus making a restart from scratch unfeasible. Checkpoint-Restart (CR), the main method to survive failures for such applications faces additional challenges in this context: not only does it need to minimize the performance overhead on the application due to checkpointing, but it also needs to operate with scarce resources. Given the iterative nature of the targeted applications, we launch the assumption that first-time writes to memory during asynchronous checkpointing generate the same kind of interference as they did in past iterations. Based on this assumption, we propose novel asynchronous checkpointing approach that leverages both current and past access pattern trends in order to optimize the order in which memory pages are flushed to stable storage. Large scale experiments show up to 60% improvement when compared to state-of-art checkpointing approaches, all this achievable with an extra memory requirement of less than 5% of the total application memory.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability

General Terms

Design, Performance, Experimentation

Keywords

scientific computing, high performance computing, cloud computing, fault tolerance, checkpoint restart, asynchronous checkpointing, adaptation to access pattern

1. INTRODUCTION

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
HPDC'13, June 17–21, 2013, New York, NY, USA.
Copyright 2013 ACM 978-1-4503-1910-2/13/06 ...$10.00.

Scientific and data-intensive computing have matured over the last years in all fields of science and industry. They provide an indispensable tool for new insight and solutions to complex problems through modeling, simulation and data analysis. From private-owned data-centers to leadership-class supercomputing facilities, the drive for more computational capabilities has made petascale architectures a reality [2], with predictions of reaching exascale by the end of this decade [22].

Such an explosion of scale introduces many challenges, among which a crucial challenge is *fault tolerance*. With failure rates predicted in the order of tens of minutes [22] and applications running for extended periods of time over a large number of nodes, an assumption about complete reliability is highly unrealistic. Thus, one must consider failures as rather the norm than the exception. Furthermore, since application processes are tightly coupled and depend on each other to make progress with the computation, the failure of one process eventually leads to the failure of all processes. Thus, for the class of problems that we consider, fault tolerance becomes particularly difficult.

Checkpoint-Restart (CR) [14] is a popular approach to provide fault-tolerance for scientific applications. Fault tolerance is achieved by saving recovery information periodically during failure-free execution and restarting from that information in case of failures, in order to minimize the wasted computational time and resources. Although alternatives to CR based on redundancy [7, 30]) have been considered before, such approaches have rarely been adopted in practice for scientific applications due to high performance and resource overhead.

Faced with increasing scale, achieving efficient CR becomes a challenging task. Simple approaches such as synchronous checkpointing become unfeasible: due to high checkpointing frequency, the application would spend the majority of time taking checkpoints rather that running useful computations, with dump times predicted by Jones et al. [21] in the order of several hours. Thus, it becomes increasingly important to devote attention to asynchronous mechanisms that parallelize the checkpointing and computations in order to hide the checkpointing latency. This however is a non-trivial task: it implies capturing and storing the the state of a computation while allowing the computation to progress at the same time. Since this state is mostly composed of allocated memory (used henceforth to refer to the state itself), the checkpointing process and the computation will

compete for this memory, which implies the need to minimize potential conflicts.

An important factor that augments this issue is *the need to operate with scarce resources*. Although large-scale datacenters have a lot of memory, this is a precious resource that is expensive to leverage for potential memory copies that help diminish the conflicts between the computation and checkpointing. This happens for several reasons. First, problem sizes attacked by modern applications have been growing fast, causing a declining memory bytes-to-FLOP ratio: from 0.85 for the No. 1 machine on Top500 in 1997 to 0.01 for projected exa-flop machines [3]. Thus, more and more memory is needed for the computation itself, leaving little extra memory for other operations. Second, extra memory generates operational costs either in terms of energy consumption or direct extra charges to users. The latter aspect is of particular interest in the context of HPC cloud platforms, which are increasingly considered as a cost-effective alternative for running HPC applications. Under such circumstances, virtual machine (VM) instances are more expensive the more memory they provide, prompting the need to provide a configuration where the application uses as little extra memory as possible besides what is absolutely need.

In this paper we propose *Adaptive Incremental Checkpointing* (AI-Ckpt), an asynchronous checkpointing runtime specifically designed to operate with scarce extra memory/local storage. Unlike conventional approaches, we introduce a novel checkpointing strategy that is highly versatile: it dynamically adapts to the access pattern of the application and predicts future memory accesses in order to minimize the interference of the background checkpointing process on the application.

We summarize our contributions below:

- We present a series of design principles that facilitate efficient asynchronous checkpointing. In particular, we leverage the assumption that for the iterative application class we consider, first-time writes to memory regions that need to be checkpointed asynchronously generate the same kind of interference as for the previous iterations and thus it is possible to commit the checkpointing data to stable storage in an optimized order that avoids memory access contention between the application execution and the checkpointing process.

- We show how to materialize these design principles in practice through a series of algorithmic descriptions, that are then applied to implement a checkpointing runtime library capable of tracking both user-defined memory contents explicitly or dynamic memory allocations implicitly (Sections 3.3 and 3.4).

- We evaluate our approach in a series of experiments, using both synthetic benchmarks and two real-life scientific applications. These experiments demonstrate significant improvement in overall checkpointing time, while reducing at the same time the negative impact of checkpointing overhead on the performance of the application (Section 4).

2. RELATED WORK

The simplest way to deal with CR is to leave the this issue to the application developer, which is known as *application-level checkpointing*. In this case, checkpointing can be hand-optimized by leveraging application-specific properties, however at the cost of added complexity that can become prohibitively expensive [24]. At the other extreme is *system-level checkpointing*. In this case, checkpointing is completely transparent with respect to the application, however it is inherently difficult to make feasible because of much larger state size [28] that needs to be saved and the need to employ a checkpointing protocol [14] to guarantee consistency. To fill the gap between the two, an alternative is to provide checkpointing through a run-time library: the checkpoint contents and the places where checkpoints should be taken are application defined, but the how to save the checkpointing data is the responsibility of the system. This is called *user-defined* checkpointing and is employed in several approaches [6, 25].

Regardless of the employed approach, the checkpointing data needs to be saved in a persistent fashion to stable storage that can survive failures. Given the huge amount of checkpointing data that needs to be saved and the widening gap between computational capabilities and I/O bandwidth, this can quickly lead to unacceptable overheads and poor scalability.

One direction that can be explored in order to alleviate this issue is how to reduce the checkpoint sizes. In this context, *incremental checkpointing* was proposed: it is based on the idea that checkpointing data does not fully change from one checkpoint to another, thus storing only incremental differences is enough to restart. Incremental approaches can be broadly classified into two categories: *page-based* and *deduplication-based*. Page-based approaches [31, 17] trap writes to memory in order track all changes and build a set of dirty pages that need to be saved. De-duplication based approaches [4] on the other hand identify differences by means of computation (most often hashing). It is also possible to combine these approaches into hybrid schemes, e.g. hybrid page-based/deduplication-based schemes [16] or hybrid incremental/full checkpointing schemes [32]. Furthermore, de-duplication can be extended beyond the scope of a single process by identifying memory pages with identical content across groups of processes [27]. In either case, incremental checkpointing can complemented with compression techniques [26] to further reduce the checkpoint sizes.

Another direction that helps alleviate the overhead of checkpointing is to depart from synchronous checkpointing. One idea in this context is to design quasi-synchronous checkpointing algorithms that prevent contention to stable storage [23]. Another idea is to use multi-level checkpointing [6, 25, 12], i.e. dump the checkpointing data on fast local storage and then asynchronously flush this data to global storage. Dorier et al. [13] have shown significant benefits of this idea for multi-core architectures, however at the expense of using a large shared memory buffer where the checkpointing data from all cores is aggregated. To limit the memory usage and overhead of copies, a third possible idea is to avoid blocking the application during checkpointing altogether, by using asynchronous techniques such as *copy-on-write*.

Extensive related work has also been undertaken in the area of *live migration*, in particular *pre-copy* [10] and its derivatives. Like asynchronous checkpointing, precopy aims to transfer the memory contents from a source to a destination while the source continues execution and potentially changes the contents. However, the goal here is to converge

to a state where both source and destination have identical memory contents in order to be able to transfer control to the destination and continue execution from there. The convergence does not imply any ordering constraints and in fact it is beneficial to delay transferring frequently changed contents [10, 20, 29]. On the other hand, in the case of asynchronous checkpointing, the memory contents cannot be overwritten before it was transferred to the checkpoint, which introduces additional ordering constraints and thus makes the problem more difficult.

Our own work focuses on efficient asynchronous checkpointing in spite of limited extra available memory. We believe the key to do so is to adapt to the access pattern of the application in order to leverage the little spare extra memory as efficiently as possible for copy-on-write. Although there are several established ways to reason about memory access patterns, notably the working set model [11] and by extension to our context the writable working set [10], the synchronization issues raised by asynchronous checkpointing shift the focus to ordering rather than frequency of use (in particular, only first time writes between checkpoint requests need to be considered - see Section 3.1). To our best knowledge, we are the first to formulate the assumptions and explore the benefits of adapting to such specific access pattern requirements and ordering constraints.

3. OUR APPROACH

This section presents the general design principles with an algorithmic description and shows how to implement and integrate them in a typical large scale distributed architecture.

3.1 Design principles

Define and manage protected memory regions. AI-Ckpt enables both user-defined checkpointing and transparent checkpointing. In case of the former, it is the responsibility of the user to define what memory regions are important and needed on restart, which is achieved by using specific memory allocation primitives. In case of the latter, AI-Ckpt automatically captures all memory allocations and considers the requested regions important for restart. Regardless of how such memory regions were defined, they are directly managed by AI-Ckpt, as detailed below. Henceforth we call such memory regions "protected".

Leverage dirty-page tracking to capture write access pattern and checkpoint increments asynchronously. We leverage dirty page tracking in order to simultaneously enable both incremental and asynchronous checkpointing. This works as follows: whenever a checkpointing request is received, all memory regions managed by AI-Ckpt are marked as read-only. At the same time, in background, all pages that were modified so far (i.e. marked as "dirty") are flushed to stable storage. Initially, any new protected memory region is marked as read-only. The application does not block during the checkpoint request. Instead, any write to protected memory regions is trapped and handled in a special fashion. First of all, the corresponding page is marked as "dirty" for the next checkpoint request. When this next checkpoint request eventually arrives, it is trivial to establish what pages were modified since the last checkpoint by simply checking their dirty status. Second, it can happen that the background checkpointing has not managed to flush the page yet. In this case, it is not possible to simply continue with the write, as doing so will corrupt the content that is expected to be flushed. To deal with this issue, there are two options: either wait until the page was successfully flushed or employ copy-on-write. Finally, once the page has been properly handled, the write protection is lifted and the write to the page (and any other subsequent writes) can continue normally.

Note that asynchronous checkpointing is susceptible to jitter, which for HPC applications is particularly problematic [19] due to the tightly-coupled nature that leads to accumulation of delays. In our context, it is not enough to hide the latency of transfers between memory and persistent storage using traditional approaches such as dedicated I/O cores, because an important source of jitter is the delays caused by waiting. Thus, for the rest of this section we concentrate on this aspect.

Use bounded copy-on-write to avoid wait delays. A popular technique to avoid waiting is to simply create a private copy of the page and apply the write there, which is known as *copy-on-write*. In our case, is not possible to employ copy-on-write in the traditional sense, because this would disrupt the address space expected by the application. Thus, the only possibility left is to create a private copy for the background checkpointing process and perform the actual write on the original page. This works only if the original page is not already in the process of being flushed, otherwise a wait is necessary. Furthermore, this approach cannot be abused indefinitely to avoid waiting: besides the obvious limitation of spare memory available other than for application needs, too much copy-on-write could potentially have a high negative impact on performance that offsets the benefits of avoiding a wait. Thus, we limit the amount of memory available for copy-on-writes to a fixed value that can be configured by the user before launching the application.

Adapt dirty page flushing to access pattern. In its simplest form, the strategy used to flush dirty pages follows a predefined order that is independent of the access pattern. Although quite popular and easy to implement, such an approach is not feasible if operating under a limited amount of memory reserved for copy-on-write. This results from the fact that once the copy-on-write memory buffer is full, the application has to wait until either the page it is trying to write has been flushed or enough free space was released back to the copy-on-write buffer (which enables it to perform a new copy-on-write and thus avoid waiting).

Obviously, a strategy that does not care about the access pattern can hit on the worst case scenario where it flushes all other pages except the page that is waited for or any of the pages that triggered a copy-on-write. To eliminate this effect, we propose a strategy that adapts to the access pattern. More specifically, if the application is waiting for a page, we reschedule that page such that it is flushed as soon as possible. Even if the application is not waiting for any page, we still prefer pages that triggered copy-on-write, as this keeps the buffer free for "dark times" when frequent copy-on-writes might quickly fill it up.

Leverage access pattern history to optimize flushing. Scientific high performance computing applications are typ-

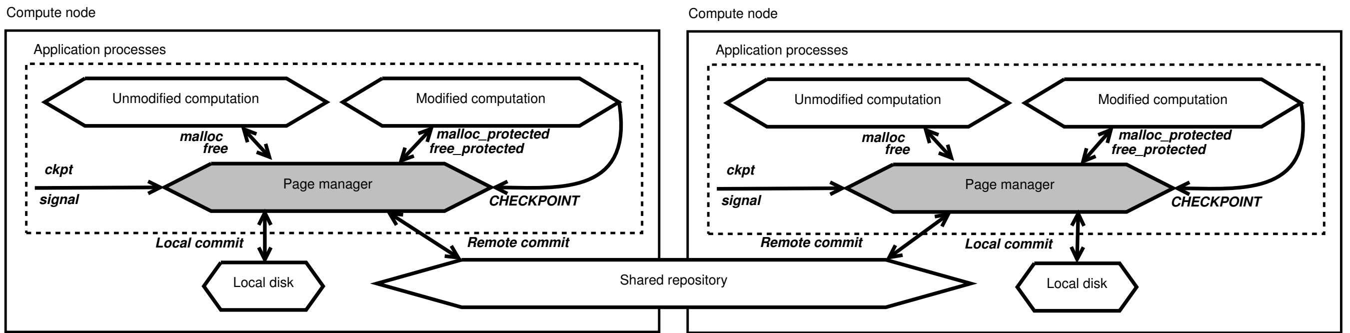


Figure 1: Distributed architecture that integrates our approach via the page manager (dark background).

ically iterative: each process first synchronizes with other processes (typically by message passing), then performs a complex computation on some in-memory data (which involves an alteration of a significant part of it). Most commonly, after a predefined amount of iterations was completed, all processes are checkpointed.

Given this repetitive nature, processes tends to generate highly similar access patterns between checkpoints. Thus, we propose to leverage this fact to further enhance the ability to adapt to the access pattern. Since only first-time writes to memory pages can introduce delays between consecutive checkpoint requests, we propose to record for all such first time writes *when* they happened and under *what circumstances*.

Although we could record the exact timestamp, for simplification reasons we consider that the access order is enough to estimate the "when". With respect to the circumstances, we are interested in the type of interference that this access caused to the asynchronous checkpointing process. This translates to several possible scenarios: (1) a copy-on-write was performed for the page; (2) the application had to wait for the page to be flushed first (because no more copy-on-write slots were available or because the page was in the process of being written to storage); (3) the page was accessed while the checkpointing was still in progress but it didn't trigger any copy-on-write or wait because it was flushed before; and finally (4) the page was accessed after the checkpointing has completed.

We maintain this information for the whole duration between two consecutive checkpoint requests, which we will herein refer to as *epoch*. Armed with this knowledge, we aim to minimize the interference that the application and the background checkpointing process experience. Based on the assumption that page accesses exhibit a similar behavior to that recorded in the previous epoch, we first commit to storage preferentially those pages that have history of "bad behavior" (i.e. caused waits or copy-on-writes) and only then proceed to commit those pages that had a "good behavior". Furthermore, when multiple pages exhibit a similar behavior, we prefer the page that was accessed the earliest.

A detailed algorithmic description of how this works is presented in Section 3.3. Note that for simplification reasons, we describe here only the case when the epoch matches the iterative behavior of the application (i.e. checkpoints requested every N iterations). In order to deal with "unaligned" epochs, one potential solution is to maintain enough past first-time writes to cover a full iteration, then use this information to "align" the epoch to the iterative behavior (e.g. based on its first few writes). However, providing such a solution is outside the scope of this work.

3.2 Architecture

A simplified distributed architecture that integrates our approach is depicted in Figure 1.

Each compute node runs the *application processes*, which include either a *modified* or *unmodified* computation. In the first case, the application directly controls what memory regions are protected (using malloc_protected and free_protected). In the second case, memory management is handled transparently by capturing all malloc/calloc/realloc as well as free calls.

All memory pages that correspond to the protected memory regions are monitored by the *page manager*, which is the central actor of our approach and is responsible to implement our adaptive asynchronous checkpointing approach. The moment when to initiate the checkpointing is determined either explicitly by the application (which can directly call the CHECKPOINT primitive) or by any other external process (by sending a signal to the page manager, which will then internally call the CHECKPOINT primitive).

The page manager is designed in a modular fashion such that it is easy to plug in different storage backends where the dirty pages can be committed. These can range from POSIX-enabled filesystems (local file systems or parallel file systems deployed remotely, e.g. *PVFS* [9]) to specialized high-availability cloud repositories (such as *Amazon S3* [5]). Furthermore, it can easily complement dedicated checkpointing repositories designed for specific roles (such as virtual disk snapshotting [28]).

Also note that although local storage is particularly attractive as a place to store checkpoints (because it is much faster and more scalable compared to parallel file systems or other conventional remote storage options), it is prone to failures and thus unreliable. However, there are several options to overcome this issue, with data replication on different nodes being the most straight-forward. More cost-effective solutions based on erasure codes are also possible in order to reduce both performance overhead and storage space requirements, as demonstrated by our previous work [18].

3.3 Zoom on the page manager

The *page manager* consists of two independent modules that run concurrently and compete for access to the mon-

itored memory pages: (1) a module that asynchronously commits all dirty pages accumulated at the moment when a checkpoint is requested (ASYNC_COMMIT); and (2) a module that traps all first writes during application runtime after the checkpoint was requested (PROTECTED_PAGE_HANDLER).

The application can initiate a new checkpoint by calling the CHECKPOINT primitive (Algorithm 1). Upon receipt of this request, the page manager first checks if the previous checkpoint is still in progress ($CheckpointInProgress =$ **true**), waiting for it to complete if this is the case. Once the previous checkpoint was successfully committed to storage, it initializes three data structures used by the two modules to synchronize: *Dirty*, which represents the set of all dirty pages, accessed after the CHECKPOINT call, $AT[p]$ which represents the type of access that was triggered by a page p in *Dirty* and finally $Index[p]$ which represents the order of p in *Dirty*.

Algorithm 1 Initiate a new checkpoint

```
 1: procedure CHECKPOINT
 2:     if CheckpointInProgress then
 3:         wait until CheckpointInProgress = false
 4:     end if
 5:     LastDirty ← Dirty
 6:     LastAT ← AT
 7:     LastIndex ← Index
 8:     Dirty ← ∅
 9:     AccessOrder ← 0
10:     for all p ∈ Pages do
11:         write protect p
12:         AT[p] ← UNTOUCHED
13:         Index[p] ← 0
14:     end for
15:     for all p ∈ LastDirty do
16:         State[p] ← PAGE_SCHEDULED
17:     end for
18:     CheckpointInProgress ← true
19:     notify ASYNC_COMMIT
20: end procedure
```

The type of access triggered by a page p can take one of the following values: *UNTOUCHED*, which is the initial value and means the page was not yet accessed; *COW*, which means p has triggered a copy-on-write; *WAIT*, which means PROTECTED_PAGE_HANDLER had to wait until p was committed; *AVOIDED*, which means p was accessed after it was committed, but before the checkpointing process has finished; and finally *AFTER*, which means p was accessed after the checkpointing process has successfully completed.

Each of *Dirty*, *AT* and *Index* has a corresponding data structure prefixed by *Last*, which has the same semantics as the original except for the fact that it represents the statistics of previous epoch rather than the current one. Finally, a fourth data structure $State[p]$ describes the state of p, which can be one of the following values: *PAGE_PROCESSED*, which is the initial value and means p was already processed by the checkpointing process (either already committed or untouched); *PAGE_SCHEDULED*, which means p is dirty and needs to be committed (but this was not already done); and finally *PAGE_INPROGRESS*, which means p was locked and is in the process of being committed.

Obviously, CHECKPOINT needs to reset the access type of all pages to *UNTOUHCED* and then write protect them in order to trap all future modifications. After resetting *Dirty*, all pages that were modified since the last checkpoint (now in *LastDirty*) will be marked as *PAGE_SCHEDULED*. Once this step is complete, ASYNC_COMMIT can proceed to commit the dirty pages.

Algorithm 2 Handle a write to a protected page

```
 1: procedure PROTECTED_PAGE_HANDLER(p)
 2:     if State[p] = PAGE_SCHEDULED ∧ |CowPage| <
    Threshold then
 3:         CowPage[p] ← copy of p
 4:         AT[p] ← COW
 5:     else if State[p] = PAGE_PROCESSED then
 6:         if CheckpointInProgress then
 7:             AT[p] ← AVOIDED
 8:         else
 9:             AT[p] ← AFTER
10:         end if
11:     else
12:         WaitedPage ← p
13:         while State[p] ≠ PAGE_PROCESSED do
14:             wait for notification from ASYNC_COMMIT
15:         end while
16:         WaitedPage ← nil
17:         AT[p] ← WAIT
18:     end if
19:     Dirty ← Dirty ∪ {p}
20:     AccessOrder ← AccessOrder + 1
21:     Index[p] ← AccessOrder
22:     remove write protection from p
23: end procedure
```

In the mean time, any modification to a write protected page p will be trapped by PROTECTED_PAGE_HANDLER. This process is detailed in Algorithm 2. More specifically, if p was scheduled but not yet committed and there are enough copy-on-write slots available ($|CowPage| < Threshold$), then a new copy-on-write slot can be used. Otherwise, if p was already committed, nothing needs to be done except setting its access type to *AVOIDED* or *AFTER*. Finally, the only possibility left is that p is in progress or there are not enough copy-on-write slots left. In this case, we need to wait until p was committed. In order to avoid waiting as much as possible, a hint is created for ASYNC_COMMIT, by assigning a special marker $WaitedPage$ to p. This marker will be used by ASYNC_COMMIT to maximize the priority of p (i.e. commit it as soon as possible). Once p has been successfully handled, it is added to the *Dirty* set, after which its access order index is set and finally its write protection is removed.

The dirty pages are committed in an iterative fashion, as detailed in Algorithm 3. As long as there are still dirty pages left in *LastDirty*, ASYNC_COMMIT selects one such page p (using SELECT_NEXT_PAGE). If p resulted in a copy-on-write, its copy is committed and the corresponding slot released. Otherwise, p is locked (i.e. *PAGE_INPROGRESS*), directly written to storage and then unlocked (i.e. marked as *PAGE_PROCESSED*). In either case, p is removed from *LastDirty* and the next iteration is started.

The central aspect of this iterative process is how to select the next page to be committed, which is detailed in Algorithm 4. Obviously, if $WaitedPage \neq$ **nil**, then $WaitedPage$ must be committed as soon as possible, in order to be able to unblock PROTECTED_PAGE_HANDLER and thus enable the

Algorithm 3 Commit modified pages asynchronously to storage

```
1: procedure ASYNC_COMMIT
2:     while true do
3:         wait for notification from CHECKPOINT
4:         while LastDirty ≠ ∅ do
5:             p ← SELECT_NEXT_PAGE
6:             if AT[p] = COW then
7:                 commit CowPage[p] to storage
8:                 release slot of p in CowPage
9:             else
10:                State[p] ← PAGE_INPROGRESS
11:                commit p to storage
12:                State[p] ← PAGE_PROCESSED
13:                notify PROTECTED_PAGE_HANDLER
14:            end if
15:            LastDirty ← LastDirty \ {p}
16:        end while
17:        CheckpointInProgress ← false
18:    end while
19: end procedure
```

application to continue. Furthermore, if there are pages that triggered copy-on-write, they will be preferentially committed in order to release copy-on-write slots as soon as possible. If neither of these two cases applies, then a page is selected based on the access pattern exhibited by the application before the checkpoint request, under the assumption that it will reflect the future access pattern. More specifically, preference is given to the pages that were marked *WAIT*, then those pages that were marked *COW* and finally those pages that were marked *AVOIDED*. This way, the pages that could have the potentially worst interference are committed first, thus minimizing the chance of future interference. Pages that are marked *AFTER* are given the least priority, as they are likely to keep this status until the next checkpoint request and thus are not likely to generate interference. In either case, if more than one page has the same access type, preference is given to the page that was accessed the earliest before the checkpoint request (i.e. smallest *LastIndex*).

Algorithm 4 Select next page to commit to storage

```
1: function SELECT_NEXT_PAGE
2:     if WaitedPage ∈ LastDirty then
3:         return WaitedPage
4:     end if
5:     if ∃p ∈ LastDirty | AT[p] = COW then
6:         return p
7:     end if
8:     if ∃p ∈ LastDirty | LastAT[p] = WAIT then
9:         return p | LastIndex[p] is minimal
10:    end if
11:    if ∃p ∈ LastDirty | LastAT[p] = COW then
12:        return p | LastIndex[p] is minimal
13:    end if
14:    if ∃p ∈ LastDirty | LastAT[p] = AVOIDED then
15:        return p | LastIndex[p] is minimal
16:    end if
17:    return any remaining p ∈ LastDirty
18: end function
```

3.4 Implementation

We implemented *AI-Ckpt* in form of two libraries. The first library implements the *page manager*, while exposing the CHECKPOINT primitive to the application. It also exposes two specific memory allocation/deallocation routines: malloc_protected and free_protected. These routines can be used to control directly at application-level what memory contents needs to be checkpointed.

For the case when transparency is desired, we implemented a second library that traps all dynamic memory allocations performed by the application and automatically reports all involved pages to the page manager. To this end, we built our own custom memory allocator on top of *jemalloc* [15], a scalable high performance malloc implementation designed to efficiently support concurrent allocations. The application itself needs not necessarily be linked against this second library, as it is enough to preload the library in order to replace the standard system malloc implementation. This is particularly useful when the application

The page manager was implemented from scratch using the *Boost* C++ collection of libraries, which introduces several optimized implementations of hash tables and balanced trees that we used to adopt the algorithms presented in Section 3.3 with minimal overhead.

In order to trap writes to memory, we rely on the mprotect system call to mark specific pages as read only. If the application attempts to write to such pages, the kernel will trigger a SIGSEGV signal, which we trap using a custom signal handler that implements PROTECTED_PAGE_HANDLER (Algorithm 2). This mechanism involves certain non-trivial aspects that require closer consideration. In particular, if the application passes read-only memory regions to certain system calls that are supposed to write to the memory (for example read), these system calls will not trigger a SIGSEGV but rather fail. To circumvent this issue, we trap such system calls and artificially trigger the necessary SIGSEGVs before launching the system call itself.

4. EVALUATION

After briefly describing the experimental setup and methodology, we evaluate in this section our approach both in synthetic and real life settings.

4.1 Experimental setup

The experiments were performed on *Grid'5000*, an experimental testbed for distributed computing that federates nine sites in France, as well as *Shamrock*, an experimental platform of the Exascale Systems group of IBM Research in Dublin, Ireland.

For the Grid'5000 experiments, we used 42 nodes of the Rennes site, each of which is equipped with a quadcore Intel Xeon X5570 x86_64 CPU, local disk storage of 500 GB (access speed $\simeq$55 MB/s using SATA II ahci driver) and 24 GB of RAM. The nodes are interconnected with Gigabit Ethernet (measured 117.5 MB/s for TCP sockets with MTU = 1500 B with a latency of $\simeq$0.1 ms). Each node is powered by recently updated Debian Sid distribution where OpenMPI 1.4.3 was installed and set up. In this setting, we store the checkpoints in a "conventional" fashion by using a parallel file system. To this end, we reserve 10 nodes to act at storage elements and deploy the *PVFSv2* [9] parallel file system on them. The rest of 32 nodes are used to run

our MPI applications and have access to the PVFS deployment through the POSIX interface made available through the PVFS FUSE module.

The Shamrock testbed consists of 160 nodes interconnected with Gigabit Ethernet, each of which features an Intel Xeon X5670 CPU (6 cores, 12 hardware threads), HDD local storage of 1 TB and 128 GB of RAM. For the purpose of this work, we used a reservation of 28 nodes. Each node runs the Red Hat 6.2 Enterprise Linux distribution, while the MPI library installed is MPICH2 1.4.1. In this case, all nodes are reserved for running the applications, while the checkpoints are written to local storage. This setting has a potential for higher I/O scalability (as discussed in Section 3.2) and thus pushes our approach to the limits, as there are fewer opportunities to take of long I/O delays.

For the rest of this paper, we will refer to the two experimental setups simply as Grid'5000 and, respectively, Shamrock. In both setups, the memory page size used throughout our experiments is fixed at 4 KB, the default of the operating system.

4.2 Methodology

We compare three approaches throughout our evaluation:

Asynchronous incremental checkpointing using our approach.

In this setting we use AI-Ckpt to capture all dynamic memory allocations performed by the application and treat all CHECKPOINT requests according to the strategy presented in Section 3.3. We denote this setting our−approach for the rest of the paper.

Asynchronous incremental checkpointing without adaptation to access pattern.

We compare our approach with the case when the access pattern generated before the CHECKPOINT request is not taken into consideration while dumping the checkpointing data to storage. More specifically, this setting is similar to the previous one (i.e. all memory write accesses are trapped for the purpose of building the set of dirty pages that needs to be dumped to storage), except for the fact that the dirty pages are simply dumped in ascending order of their address. For the rest of this paper, we refer to this setting as async−no−pattern.

Synchronous incremental checkpointing.

The third setting we compare our approach with is a synchronous checkpointing approach that blocks inside the CHECKPOINT primitive until all checkpointing data has been successfully dumped to storage. In this setting, dirty page tracking is still used for the purpose of identifying the incremental changes since the last checkpoint, however this mechanism is greatly simplified due to the fact that the application and the checkpointing process do not compete for the dirty pages. For the rest of this paper, we refer to this setting as sync.

These approaches are compared based on the following metrics:

- *Impact on application performance*: is the performance degradation perceived by the application during checkpointing, compared to the case when no checkpointing is performed. For the purpose of this work, we are interested in the impact on the total runtime of various memory-intensive benchmarking scenarios and a real HPC scientific application.
- *Checkpointing time*: is the time elapsed between the moment when the CHECKPOINT primitive has been called and all dirty pages have been successfully committed to storage. For sync, this corresponds to the duration of time during which the application blocked in the CHECKPOINT call. For the other two approaches, the duration is directly reported by *AI-Ckpt*.
- *Access type statistics*: we are interested in statistics about the types of accesses that were triggered by the page faults, as these can explain the various observable differences in checkpointing time and performance overhead. In particular, it is desirable to have as few as possible WAITs, as they are the main source of delays for the other two metrics.

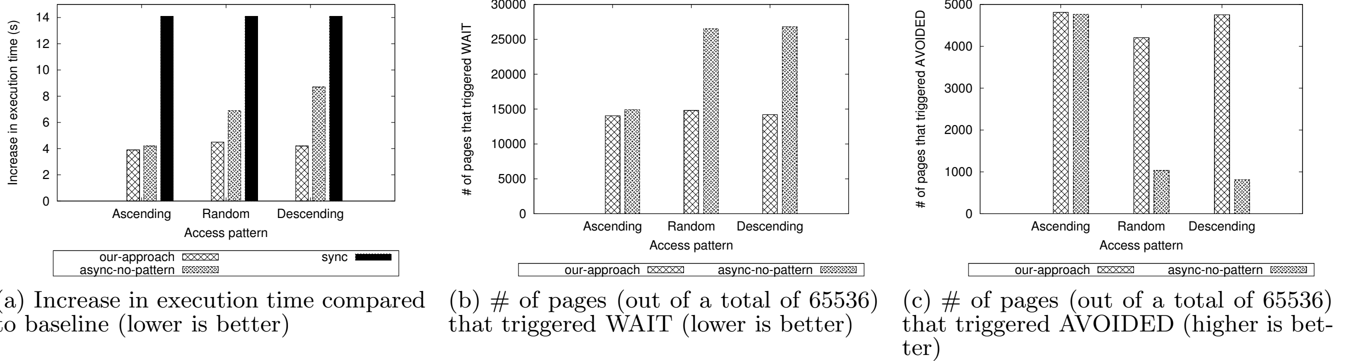
4.3 Checkpointing performance of memory-intensive benchmarks

Our first series of experiments aims to gather insight into how the memory access pattern can influence asynchronous checkpointing. To this end, we have developed a memory-intensive benchmark that allocates a large memory region and then runs a number of iterations, each of which touches the full memory content byte-by-byte in a specified order. Each time a fixed number of iterations has been completed, the CHECKPOINT primitive is called. For the purpose of this work, we fixed the number of iterations to 39, with a checkpoint request issued every 10 iterations (for a total of 3 checkpoints, each of which is overlapping with the benchmark and competes for memory accesses).

In order to understand how the order of memory writes impacts the checkpointing performance, we implemented three access patterns: *Ascending* (i.e. the memory region is accessed page-by-page from the beginning towards the end), *Random* (i.e. a random permutation of the indexes of all pages is generated and used as a fixed access order for all iterations) and finally *Descending* (i.e. the memory region is accessed page-by-page from the end towards the beginning). For each page, a simple transformation is performed: all bytes are incremented by one.

Each experiment consists in running our benchmark on one of the Grid'5000 nodes, while recording the completion time and statistics about the access types triggered by the pages. The size of the memory region is fixed at 256 MB, while the size of the copy-on-write buffer is fixed at 16 MB.

The increase in execution time for the benchmark, compared to the baseline (i.e when no checkpointing is performed), is illustrated in Figure 2(a). As expected, sync has the highest overhead of all three approaches, which is maintained at constant level regardless of access pattern. Comparing our approach to async−no−pattern reveals only small differences for the *Ascending* access pattern, which is understandable considering the fact that the actual order in which the pages are accessed matches the static order in which the pages are selected by async−no−pattern. However, when this is not the case any longer (i.e. for *Random* and *Descending*), significant differences start to appear, reaching as high as 33% and respectively 50% lower overhead in favor of our approach. Compared to sync, our approach exhibits up to 72% lower overhead.



(a) Increase in execution time compared to baseline (lower is better)

(b) # of pages (out of a total of 65536) that triggered WAIT (lower is better)

(c) # of pages (out of a total of 65536) that triggered AVOIDED (higher is better)

Figure 2: Performance results and statistics about access types triggered by pages for a memory-intensive benchmark

In order to understand these findings better, we illustrate statistics about the types of accesses triggered by the pages during the runtime of the benchmark. All statistics are measured between two consecutive checkpoints and the average for the three checkpoints is reported. Since the whole memory region is changed between two consecutive checkpoints, the total number of pages that is flushed to storage remains constant at 65536 for all three checkpoints. As can be noticed, the strategy used by our approach to adapt to the access pattern brings clear advantages over async−no−pattern, especially for *Random* and *Descending*. For these last two access patterns, our approach waits on almost 50% less pages (Figure 2(b)), while managing to avoid both waits and copy-on-writes in a proportion of more than 4x the level of async−no−pattern (Figure 2(c)).

4.4 Case study: Checkpointing performance of CM1

Our next series of experiments illustrates the behavior of our proposal in real life. For this purpose we have chosen *CM1*, a three-dimensional, non-hydrostatic, non-linear, time-dependent numerical model suitable for idealized studies of atmospheric phenomena. This application is used to study small-scale processes that occur in the atmosphere of the Earth, such as hurricanes.

CM1 is representative of a large class of HPC stencil applications that model a phenomenon in time which can be described by a spatial domain that holds a fixed set of parameters in each point. The problem is solved iteratively in a distributed fashion by splitting the spatial domain into subdomains, each of which is managed by a dedicated MPI process. At each iteration, the MPI processes calculate the values for all points of their subdomain, then exchange the values at the border of their subdomains with each other. After a certain number of iterations have been successfully completed, each MPI process triggers a checkpoint, then followed by a barrier to synchronize with all other MPI processes and finally it resumes execution.

Since CM1 is written in Fortran, it was not possible to directly use malloc_protected and free_protected. However, thanks to our custom memory allocator we were able to intercept all dynamic memory allocations triggered by the allocatable data structures, which cover all checkpointing data that needs to be saved. In order to expose the CHECKPOINT call, we implemented a minimalist wrapper library for Fortran. Using this library, we replaced the hand-optimized synchronous checkpointing implemented in CM1 with a simple call to CHECKPOINT.

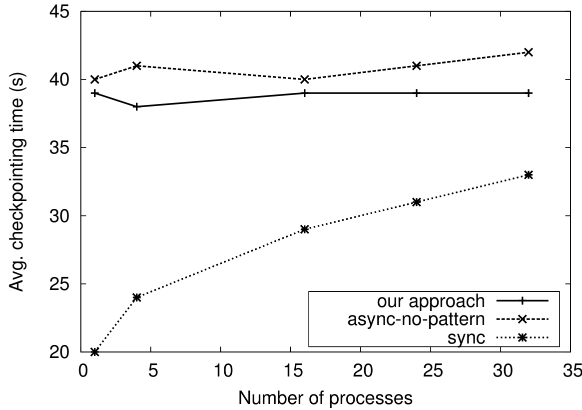
For the purpose of this work, we have chosen as input data a 3D hurricane that is a version of the Bryan and Rotunno simulations [8]. We run the simulation of this 3D hurricane on Grid'5000, with each MPI process deployed on a dedicated compute node. The checkpointing frequency is set at 50 seconds of simulated time, which for this configuration results in approx. 400 MB worth of memory content that is changed, out of a total of 728 MB. We fix the total simulation time to 180s, which is enough to trigger three checkpoints.

We aim to study two aspects: (1) how well our approach scales compared to the other two approaches and (2) how the size of the copy-on-write buffer impacts the performance of our approach compared to the other two approaches. These aspects are detailed below.

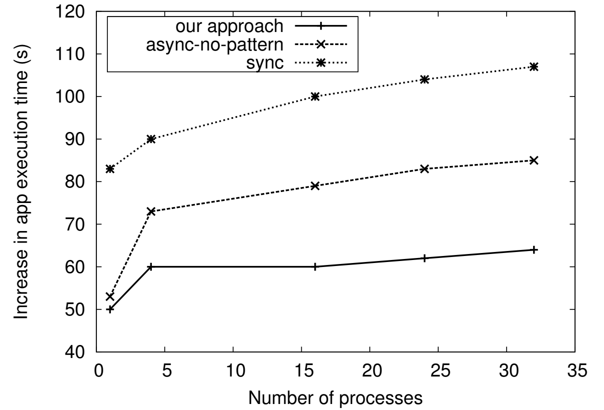
4.4.1 *Weak scalability*

Our first experiment studies the weak scalability of our approach by solving the same problem using a different precision, in such way that the size of the subdomain solved by each process remains constant at 200x200. The experiment consists in deploying an increasing number of processes, starting from one and going up to 32. We let the application run until completion and record the increase in execution time (compared to the baseline, i.e. an execution with checkpointing deactivated) as well as the average checkpointing time for the second and third checkpoint (we omit the first checkpoint as it is a full checkpoint). The copy-on-write buffer size is fixed at 16 MB.

Results are shown in Figure 3. With respect to checkpointing time (Figure 3(a)), a sharp increase can be observed in the case of sync. This effect is caused by two factors: (1) an increasing I/O pressure is generated on the storage nodes that host PVFS as the number MPI processes increases; and (2) the small system page size causes multiple concurrent small writes to PVFS, which increases the number of multiple connections that the storage servers have to handle in parallel, thus the high overhead. On the other hand, our approach and async−no−pattern are much more scalable



(a) Avg. checkpointing time (lower is better)



(b) Increase in execution time compared to baseline (lower is better)

Figure 3: Weak scalability of CM1: 400 MB/728 MB worth of incremental memory changes/process, 1 process/node

with respect to checkpointing time. Although in absolute terms the checkpointing time is higher, this is not surprising considering that the checkpointing runs in parallel with the application, and thus has to compete with it for network bandwidth. Furthermore, this also explains the much better scalability: instead of concentrated bursts of I/O as generated by sync, both asynchronous approaches distribute the I/O more evenly between checkpoints, thus reducing the I/O pressure on the storage nodes. Finally we observe that our approach marginally reduces the checkpointing time compared to async−no−pattern, which is a consequence of the interference between the network traffic generated by the application and the network traffic generated by the checkpointing.

The benefit of access pattern adaptation becomes clearly visible in the increase of execution time compared to the baseline (Figure 3(a)). In this case, our approach avoids waiting for around 6000 less pages/checkpoint when compared to async−no−pattern. This directly reflects on the increase in execution time: compared to our approach, async−no−pattern is almost 33% slower when considering the extreme of 32 processes. Since sync has to wait for all pages to be committed, it is not surprising that it exhibits the worst performance: compared to our approach, it is almost 67% slower.

4.4.2 *Impact of copy-on-write buffer size on checkpointing performance*

Our next experiment studies the impact of copy-on-write buffer size on the performance of checkpointing. Like in the previous setting, the size of the subdomain solved by each process remains constant at 200x200, however, this time we fix the size of the problem at the maximum of 32 processes and range the buffer size from 0 MB to 256 MB. For each buffer size, we let the application run until completion and record the execution time and statistics about the number of pages that were waited for.

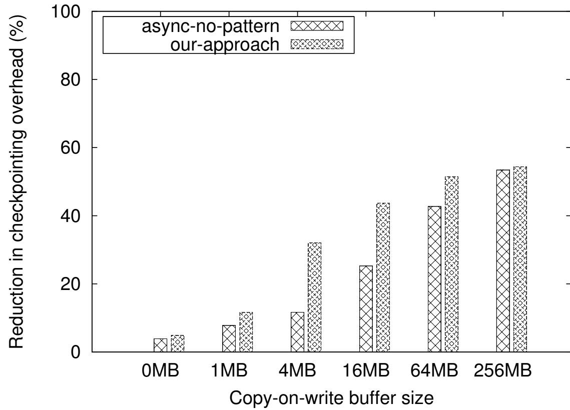
Results are shown in Figure 4(a). We illustrate the percentile increase in checkpointing overhead for both our approach and async−no−pattern when compared to sync (that is, the difference in completion time between the asynchronous approach and baseline is divided by the difference in completion time between sync and baseline, after which it is subtracted from one and multiplied by one hundred).

As can be observed, when the copy-on-write buffer size is 0 (i.e. copy-on-write is deactivated), both asynchronous approaches perform very closely and exhibit a rather small benefit over synchronous checkpointing, barely reaching 5%. We traced back this result to the fact that of all pages that were committed, most of them had to be waited for. Thus, it seems the lack of a copy-on-write buffer limits the ability of our approach to keep up in sync with the rate of memory changes and/or survive deviations from the access pattern of the previous epoch. However, when gradually increasing the copy-on-write buffer size, a higher reduction in checkpointing overhead is noticeable for both asynchronous approaches. This reduction is especially dramatic for our approach, more than doubling at each step and keeping way ahead over async−no−pattern, which starts to see a significant reduction only beginning with 16 MB. As the copy-on-write buffer gets larger, the difference between our−approach and async−no−pattern gradually gets smaller, eventually evening out at 256 MB when the number of copy-on-write slots is high enough to avoid all page waits. According to these observations, we conclude that adaptation to access pattern has an advantage over no adaptation in all cases, with the most dramatic differences occurring for small copy-on-write buffer sizes, which gives our approach the upper edge.

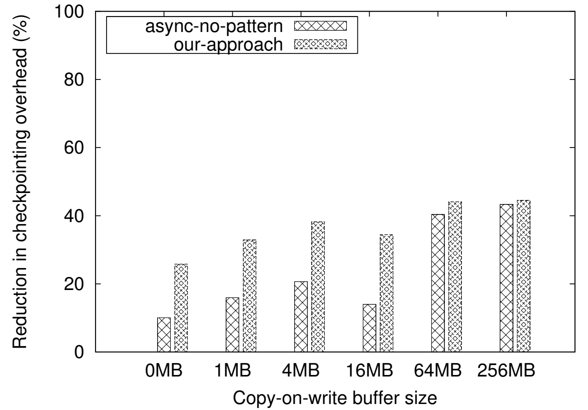
4.5 Case study: Checkpointing performance of MILC

Our second case study focuses on another high performance computing application: *MIMD Lattice Computation (MILC)*. This application is particularly useful in the field of quantum chromodynamics (QCD), which describes the interactions of the quarks and gluons that form particles such as protons, neutrons and mesons.

MILC treats continuum space-time as a four-dimensional



(a) CM1: 32 processes (higher is better)



(b) MILC: 280 processes (higher is better)

Figure 4: Impact of copy-on-write buffer size on the performance of checkpointing: reduction in overhead compared to sync

hypercube lattice that is called lattice QCD. In this discretization, lattice sites carry fields representing quarks and the links between lattice sites carry gluon fields. Each link between nearest neighbors in this lattice is associated with a 3-dimensional SU(3) complex matrix for a given field. The fields evolve using an iterative procedure (configuration generation phase), and after a sufficient number of steps the system has changed enough that the new configuration is archived for further analysis. Many different physics projects can use this configuration. To speed-up this process, the lattice is split into subdomains and distributed among MPI processes.

For the purpose of this work, we adapted the NERSC-6 procurement version of the MILC benchmark [1] in order to use AI-Ckpt. To minimize modifications to the benchmark, we use our custom memory allocator to transparently intercept all memory allocations. Our only modification was to add a call to the CHECKPOINT primitive at the end of the computation of each trajectory. We run the benchmark on Shamrock, with each node running 10 MPI processes (leaving two spare I/O cores). We fix the number of trajectories to 3, which corresponds to three evenly spaced checkpoints throughout the runtime. In this scenario, each process touches approx. 830 MB out of a total allocated memory of 868 MB.

As with CM1, we are interested in both the scalability of our approach compared to the other two approaches and the impact of the copy-on-write buffer size.

4.5.1 *Weak scalability*

This studies the weak scalability of our approach by solving an increasingly larger size of the problem in such way that each process solves a fixed subdomain of the lattice that is 20x32x32x18 large. We deploy an increasing number of processes starting from 10 (1 node) up to 280 (28 nodes). We record the average checkpointing time and the increase in execution time (compared to the baseline, i.e. an execution with checkpointing deactivated). The copy-on-write buffer is deactivated for this scenario.

Unlike the case of CM1, we used local storage in order to persist the modified memory pages of the checkpoints. This enables better overall I/O scalability considering the much larger scale of the problem (i.e. 20 times more checkpointing data compared to CM1) and better reflects more recent trends. On the other hand, it pushes our approach to the limits due to less I/O contention and more homogeneous I/O delays that give sync better scalability and thus better overall chances to compare more favourably to the two asynchronous approaches.

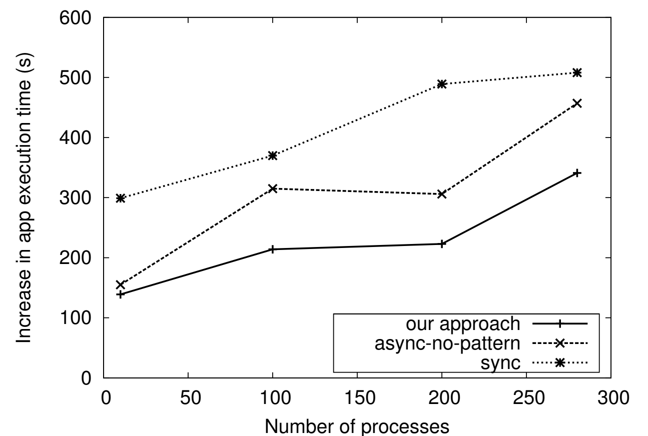


Figure 5: Weak scalability of MILC with 830 MB/866 MB worth of incremental memory changes/process, 10 processes/node: increase in execution time compared to baseline (lower is better)

Under these circumstances, for all three approaches the average checkpointing time remains almost constant and fluctuates around $210s \pm 10s$, with a slight advantage for sync due to the fact that no overlapping occurs with the application runtime. This shows that the impact of I/O distribution and contention is of less importance as in the case of CM1.

To understand how the access pattern adaptation man-

ages to reduce the checkpointing overhead and improve the time-to-completion for the application, we depict the increase of execution time compared to the baseline in Figure 5. As can be observed, our approach outperforms sync by more than 25%, which is more than double the improvement experienced by async−no−pattern (11%). This result is highly significant considering the use of local storage: it shows that even when I/O delays are short and do not increase with larger scale, access pattern adaptation maintains its benefits compared to sync and is more scalable than async−no−pattern, whose advantage over sync experiences an overall decreasing trend.

4.5.2 *Impact of copy-on-write buffer size on checkpointing performance*

Similar to the case of CM1, we study the impact of copy-on-write buffer size on the performance of checkpointing for MILC: we fix the problem size to its maximal extent (280 processes) but vary the copy-on-write buffer size.

Results are shown in Figure 4(b). Again, we illustrate the percentile increase in checkpointing overhead for both our−approach and async−no−pattern when compared to sync. Unlike the case of CM1, when the copy-on-write buffer size is 0 (i.e. copy-on-write is deactivated), a large benefit of our−approach over async−no−pattern is already visible. Thus, in this case the rate at which memory is changed and/or deviations from the access pattern of the previous epoch are much smaller. When gradually increasing the copy-on-write buffer size, both approaches experience a higher reduction in checkpointing overhead. As can be observed, our−approach constantly outperforms async−no−pattern by more than 100% up to 64 MB, when the difference between the two approaches gradually gets smaller and evens out at 256 MB. As in the case of CM1, adaptation to access pattern always outperforms no adaptation, with the largest differences observable for for small copy-on-write buffer sizes.

5. CONCLUSIONS AND FUTURE WORK

Checkpoint-Restart (CR) is a key method to provide fault tolerance for large-scale scientific applications. With increasing scale, CR faces an additional challenge: besides the implicit goal of minimizing the performance overhead during fault-free execution, it has to operate with limited extra available memory besides the one allocated by the application for computational needs.

In this paper, we have proposed *AI-Ckpt*, a runtime environment that enables asynchronous incremental checkpointing. Unlike other state-of-art CR approaches, our proposal is specifically optimized to both adapt to the current memory access pattern and learn from the previous access pattern in order to flush dirty pages during background checkpointing with minimal impact on the running application.

We demonstrated the benefits of our approach through experiments that involve dozens of nodes and hundreds of processes, using both benchmarks and real applications. Compared to naive asynchronous incremental checkpointing, we show up to 30% less checkpointing performance overhead in the real world, which grows up to 60% when compared to synchronous checkpointing. All these benefits are achievable for small copy-on-write buffers that represent less than 5% of the total application memory.

Overall, we conclude that leveraging the current and past memory access patterns during asynchronous checkpointing can significantly lower the overhead on application performance, especially for those that exhibit an iterative nature and generate repetitive memory access patterns. This idea can be further enhanced with small copy-on-write buffers to better handle unexpected deviations from the access pattern between consecutive checkpoints and thus further reduce the overhead of asynchronous checkpointing. Although this effect is naturally present even when there is no awareness of access pattern, much larger overhead reductions are possible at much smaller copy-on-write buffer sizes when leveraging the access pattern. This is a double-win scenario: the application both finishes faster and consumes less memory.

Based on these results, we plan to further explore how to leverage adaptation to the access pattern in the context of CR. In particular, for simplification reasons we scheduled the flushing of dirty pages based on access order, without taking into account the temporal aspect. Thus, one interesting direction to explore is whether introducing timestamps makes room for further optimizations.

Acknowledgments

This work was supported in part by the Joint Laboratory for Petascale Computing, an initiative of INRIA, UIUC, NCSA and Argonne National Laboratory. The experiments presented in this paper were carried out using the Shamrock cluster of IBM Research, Ireland and the Grid'5000/ ALADDIN-G5K experimental testbed, an initiative of the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see http://www.grid5000.fr/).

6. REFERENCES

[1] Nersc 6 procurement benchmark. http://www.nersc.org.

[2] Top 500 supercomputing sites. http://top500.org.

[3] DOE Exascale Initiative Technical Roadmap. Technical report, US Department of Energy, 2009.

[4] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *ICS '04: Proceedings of the 18th Annual International Conference on Supercomputing*, pages 277–286, St. Malo, France, 2004. ACM.

[5] Amazon Simple Storage Service (S3). http://aws.amazon.com/s3/.

[6] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. FTI: High Performance Fault Tolerance Interface for Hybrid Systems. In *SC '11: Proceedings of 24th International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 32:1–32:32, Seattle, USA, 2011. ACM.

[7] R. Brightwell, K. Ferreira, and R. Riesen. Transparent redundant computing with mpi. In *EuroMPI'10: Proceedings of the 17th European MPI user's group meeting conference on recent advances in the message passing interface*, pages 208–218, Stuttgart, Germany, 2010.

[8] G. H. Bryan and R. Rotunno. The maximum intensity of tropical cyclones in axisymmetric numerical model simulations. *Journal of the American Meteorological Society*, 137:1770–1789, 2009.

[9] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, USA, 2000.

[10] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI'05: Proceedings of the 2nd Symposium on Networked Systems Design & Implementation*, pages 273–286, Boston, USA, 2005.

[11] P. J. Denning. Working sets past and present. *IEEE Trans. Softw. Eng.*, 6(1):64–84, Jan. 1980.

[12] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi. Hybrid checkpointing using emerging nonvolatile memories for future exascale systems. *ACM Trans. Archit. Code Optim.*, 8(2):6:1–6:29, June 2011.

[13] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf. Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O. In *CLUSTER '12 - Proceedings of the 2012 IEEE International Conference on Cluster Computing*, Beijing, China, 2012.

[14] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34:375–408, September 2002.

[15] J. Evans. A scalable concurrent malloc(3) implementation for FreeBSD. In *Proceedings of BSDCan 2006*, Ottawa, Canada, 2006.

[16] K. B. Ferreira, R. Riesen, R. Brighwell, P. Bridges, and D. Arnold. libhashckpt: hash-based incremental checkpointing using gpu's. In *EuroMPI'11: Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, pages 272–281, Santorini, Greece, 2011. Springer-Verlag.

[17] 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 *SC '05: Proc of 18th International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 9:1–9:14, Seattle, USA, 2005.

[18] L. B. Gomez, B. Nicolae, N. Maruyama, F. Cappello, and S. Matsuoka. Scalable Reed-Solomon-based Reliable Local Storage for HPC Applications on IaaS Clouds. In *Euro-Par '12: 18th International Euro-Par Conference on Parallel Processing*, pages 313–324, Rhodes, Greece, 2012.

[19] T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *SC '10: Proceedings of the 23rd ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, New Orleans, USA, 2010.

[20] K. Z. Ibrahim, S. Hofmeyr, C. Iancu, and E. Roman. Optimized pre-copy live migration for memory intensive applications. In *SC '11: 24th International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 40:1–40:11, Seattle, USA, 2011.

[21] W. M. Jones, J. T. Daly, and N. DeBardeleben. Application Monitoring and Checkpointing in HPC : Looking Towards Exascale Systems. In *ACM-SE '12: Proceedings of the 50th Annual Southeast Regional Conference*, pages 262–267, Tuscaloosa, USA, 2012.

[22] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, W. Stanley, and K. Yelick. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical report, DARPA, 2008.

[23] D. Manivannan, Q. Jiang, J. Yang, and M. Singhal. A quasi-synchronous checkpointing algorithm that prevents contention for stable storage. *Inf. Sci.*, 178(15):3109–3116, Aug. 2008.

[24] P. McGrath and B. Tangney. Scrabble: A distributed application with an emphasis on continuity. *Softw. Eng. J.*, 5(3):160–164, July 1990.

[25] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC '10: Proceedings of the 23rd International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, New Orleans, USA, 2010.

[26] B. Nicolae. On the Benefits of Transparent Compression for Cost-Effective Cloud Data Storage. *Transactions on Large-Scale Data- and Knowledge-Centered Systems*, 3:167–184, 2011.

[27] B. Nicolae. Towards Scalable Checkpoint Restart: A Collective Inline Memory Contents Deduplication Proposal. In *IPDPS '13: The 27th IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, Boston, USA, 2013.

[28] B. Nicolae and F. Cappello. BlobCR: Efficient Checkpoint-Restart for HPC Applications on IaaS Clouds using Virtual Disk Image Snapshots. In *SC '11: 24th International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 34:1–34:12, Seattle, USA, 2011.

[29] B. Nicolae and F. Cappello. A Hybrid Local Storage Transfer Scheme for Live Migration of I/O Intensive Workloads. In *HPDC '12: 21th International ACM Symposium on High-Performance Parallel and Distributed Computing*, pages 85–96, Delft, The Netherlands, 2012.

[30] S. Rajagopalan, B. Cully, R. O'Connor, and A. Warfield. SecondSite: Disaster Tolerance as a Service. In *VEE '12: Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pages 97–108, London, UK, 2012. ACM.

[31] M. Vasavada, F. Mueller, P. H. Hargrove, and E. Roman. Comparing different approaches for incremental checkpointing: The showdown. In *Linux'11: The 13th Annual Linux Symposium*, pages 69–79, 2011.

[32] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Hybrid Checkpointing for MPI Jobs in HPC Environments. In *ICPADS '10: Proc. of the 16th International Conference on Parallel and Distributed Systems*, pages 524–533, Shanghai, China, 2010.