HAL
archives-ouvertes.fr

Towards Scalable Checkpoint Restart: A Collective Inline Memory Contents Deduplication Proposal

Bogdan Nicolae

▶ To cite this version:

Bogdan Nicolae. Towards Scalable Checkpoint Restart: A Collective Inline Memory Contents Deduplication Proposal. IPDPS '13: The 27th IEEE International Parallel and Distributed Processing Symposium, May 2013, Boston, United States. pp.19-28, 2013, <10.1109/IPDPS.2013.14>. <hal-00781532v2>

HAL Id: hal-00781532
https://hal.inria.fr/hal-00781532v2

Submitted on 2 Jun 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.

Towards Scalable Checkpoint Restart: A Collective Inline Memory Contents Deduplication Proposal

Bogdan Nicolae
Exascale Systems Group
IBM Research, Ireland
bogdan.nicolae@ie.ibm.com

***Abstract*—With increasing scale and complexity of supercomputing and cloud computing architectures, faults are becoming a frequent occurrence. For a large class of applications that run for a long time and are tightly coupled, Checkpoint-Restart (CR) is the only feasible method to survive failures. However, exploding checkpoint sizes that need to be dumped to storage pose a major scalability challenge, prompting the need to reduce the amount of checkpointing data. This paper contributes with a novel collective memory contents deduplication scheme that attempts to identify and eliminate duplicate memory pages before they are saved to storage. Unlike previous approaches that concentrate on the checkpoints of the same process, our approach identifies duplicate memory pages shared by different processes (regardless whether on the same or different node). We show both how to achieve such a global deduplication in a scalable fashion and how to leverage it effectively to optimize the data layout in such way that it minimizes I/O bottlenecks. Large scale experiments show significant reduction of storage space consumption and performance overhead compared to several state-of-art approaches, both in synthetic benchmarks and for a real life high performance computing application.**

***Keywords*-scientific computing; high performance computing; fault tolerance; checkpoint restart; memory checkpointing; deduplication; I/O load balancing**

I. Introduction

Scientific and data-intensive computing have matured over the last couple of 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. Unsurprisingly, this prompted a rapid development of the computing infrastructure and the application ecosystem that makes all this possible. From private-own data-centers to cloud computing centers and leadership-class supercomputing facilities, the drive for more computational capabilities has made petascale architectures a reality [1], with predictions of reaching exascale by the end of the decade [2].

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 [3] and applications running for extended periods of time over a large number of nodes, an assumption about complete reliability is highly unrealistic. Unlike loosely-coupled application where failed tasks can be simply restarted, scientific applications have tight dependencies between tasks, which makes fault tolerance highly challenging. *Checkpoint-Restart (CR)* [4] 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.

Faced with increasing scale, achieving efficient CR becomes a challenging task: due to exploding checkpoint sizes and 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. [5] in the order of several hours. This is not only a problem of performance overhead, but also of resource utilization: not only does checkpointing consume excessive storage space, but it also consumes bandwidth, which causes I/O bottlenecks and generates extra operational costs. Although there have been attempts to mitigate I/O bottlenecks by introducing local storage (i.e. HDDs, SSDs, NVMs, etc.) with growing number of cores per node even this is still insufficient to handle the I/O pressure. Thus, it becomes increasingly important to attack the problem from a different angle, i.e. attempt reduce the checkpoint sizes. While there are several techniques proposed in this direction, recent studies [6] point out that *deduplication* (i.e. identifying identical copies of data and storing only one copy) shows promising potential, with reported reductions of up to 70%.

This paper contributes with a collective inline deduplication technique that identifies and eliminates duplicate memory contents at page level granularity both locally and between *different* application processes. Our approach relies on the idea of complementing local deduplication with a distributed strategy that identifies and eliminates the most frequently appearing memory pages globally across all processes. To this end, it introduces a scalable hash-based reduction algorithm that not only builds the set of such frequently appearing pages, but also performs I/O load balancing by evenly distributing the responsibilities of who stores the full copy and who stores only a reference throughout the system. These aspects have a crucial impact in making CR scalable, both with respect to performance overhead and resource utilization. We summarize our contributions as follows:

- We present a series of design principles that facilitate efficient collective inline deduplication. Unlike conventional approaches that operate on local checkpoints, our proposal is able to identify and eliminate duplicated data at global level between *different* application processes.

(Section III-A)
- We show how to materialize these design principles in practice through a series of algorithmic descriptions, that are then applied to implement a checkpointing run-time library capable of dealing with both user-defined memory contents explicitly or dynamic memory allocations implicitly. (Sections III-C and IV)
- We evaluate our approach in a series of experiments, conducted on two different experimental testbeds, using both synthetic benchmarks and a real-life HPC application. These experiments demonstrate a large reduction of both the overall checkpoint size and the performance overhead of checkpointing. (Section V)

II. Related work

There are roughly three major directions of research on improving the scalability of CR.

One such direction explores the possibility of desynchronizing checkpoints through uncoordinated checkpointing protocols, which previously received little attention in practice due the cost and complexity introduced by message logging and the risk of domino effects [7]. To this end, several assumptions about behavior of certain HPC applications (such as send-determinism) were demonstrated to reduce message logging overhead and thus increase the feasibility of uncoordinated checkpointing [8].

Another direction is to leverage local storage and asynchronous techniques. In this context, multi-level checkpointing [9], [10] dumps the checkpointing data on fast local storage and then asynchronously flushes this data to persistent storage (e.g. a parallel file system). Dorier et al. [11] have shown significant benefits of this idea for multi-core architectures. Still, dumping all checkpointing data to a storage service is expensive, even if handled asynchronously: it steals bandwidth away from the application and creates background jitter. To avoid these negative effects, several proposals aim to directly make local storage resilient, e.g. through erasure codes [12], [13].

Finally, reduction of checkpoint sizes is another major direction. Besides compression [14], another possible solution is deduplication. Deduplication techniques are broadly classified into *static* and *content-defined* approaches. Static approaches split the input data into equally sized chunks, which are then compared among each other (either byte-by-byte or, for increased performance, based on their hash values) in order to identify and eliminate duplicates. While simple and fast, static approaches suffer from misalignment issues (i.e insertions or deletions lead to the impossibility to detect duplicates). To deal with such misalignment issues, *content defined* approaches [15] were proposed. Essentially, they involve a sliding window over the data and that hashes the window content at each step using Rabin's fingerprinting method [16]. When a particular condition is fulfilled, a new chunk border is introduced and the process is repeated until all input data was processed, leading to a collection of variable-sized chunks. This approach was used in several storage systems [17], [18]. With respect to when deduplication is performed, there are two possibilities: either *inline*, i.e. before data is committed to storage or *offline*, i.e. asynchronously at a later point.

To our best knowledge, deduplication techniques were applied so far to reduce the size of checkpoints dumped by the same process only, which we henceforth refer to as *local deduplication*. It can be combined with *incremental checkpointing*, which 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 [19]. An alternative to deduplication in order to detect changes is *page-tracking*: it traps writes to memory in order track all changes and builds a set of dirty pages that will be saved, as demonstrated by several approaches [20], [21]. It is also possible to combine these approaches into hybrid page-based/deduplication-based schemes [22].

Our own work focuses on inline deduplication techniques that go beyond local contents and are able to identify and eliminate duplicated checkpointing data belonging to different processes. To our best knowledge, *we are the first to explore the benefits of deduplication under such circumstances*.

III. Our approach

A. Design overview

Our proposal relies on three key design principles:

Apply inline deduplication at memory page level: Offline deduplication is one possible solution to deduplicate checkpointing data. This however has two major drawbacks in the context of CR. First, it involves unnecessary writes of data that will be eliminated later on. This in itself has several negative consequences: it delays the moment when the deduplication can begin and it consumes I/O bandwidth unnecessarily, which besides adding to operational costs can potentially lead to further delays due to I/O competition. Second, the way the checkpointing data is written may alter the original layout, leading to different alignments of the memory contents. This may make identical data look different from the perspective of deduplication algorithms (or at least much more expensive to identify as such).

To address these drawbacks, we propose an inline deduplication scheme that applies the deduplication *before* any checkpointing data is committed to storage. This not only eliminates any delay before the deduplication can begin, but also avoids any unnecessary writes and thus their negative consequences as well. The deduplication itself is performed by each process at page level granularity in a similar fashion as static chunking by considering each page a chunk. Working at page level granularity has two advantages: (1) it analyses the original layout of data, which for processes that exhibit similar behavior in terms of memory allocation and access (which the case for HPC applications) means less chances to suffer from misalignments; (2) it is very fast because is does not have to deal with such misalignments.

Complement local deduplication with a collective inter-process scheme: Local deduplication is a highly scalable and relatively cheap operation to perform, because no synchronization between processes is necessary. However, on the downside its search space is limited to a single process. At the other extreme, one may consider a perfect deduplication that has knowledge about all pages of all processes. However, such a perfect deduplication is not feasible in practice: even if only hash values instead of full memory contents were gathered from all processes, their number would quickly explode at large scale. This would lead to an unacceptably high communication time and bandwidth utilization required to disseminate the information about the pages, not to mention the analysis time required to identify duplicates. Thus, for best results one must strive to achieve a trade-off between local deduplication and an ideal global deduplication.

To achieve this trade-off, we propose a collective scheme that aims to gather the set of $Threshold$ most frequently appearing pages among all processes. This is done under the assumption that such pages have the best potential to further improve local deduplication: for each frequently appearing page only one process becomes the owner, while the rest uses a reference to the owner. The collective scheme itself is based on a hierarchic parallel reduction: initially it starts from the result of the local deduplication of each process and then gradually merges these results, keeping only the most $Threshold$ frequently appearing pages for each merge. This step is performed in parallel until eventually no merge is possible and thus a global set of at most $Threshold$ most popular pages remains.

Reduce I/O contention and staggering through load balancing: The benefits of collective deduplication are highly dependent on the assignment of the references. To better illustrate this point, consider an extreme scenario where the same set of memory pages needs to be checkpointed by all processes (i.e. the memory contents is replicated for each process). This is an advantageous scenario for collective deduplication, because overall only one copy of each memory page has to be stored. However, if the assignment of the references were done in such way that a single process owned all pages while the rest referenced it, this would lead to a bottleneck: except for the owner of the pages, all processes would quickly finish but then have to wait for the staggering process to complete writing the full contents. Besides staggering, an unbalanced distribution of references can lead to I/O competition: some processes that are in a relationship to another (e.g. they share the same node) may become more loaded than other groups of processes and will start competing for the same I/O resources (e.g. network bandwidth, local storage, etc.).

For this reason, it is crucial to evenly distribute the I/O load among the processes in order to avoid such situations. To this end, we introduce a load balancing strategy that is embedded inside the collective deduplication scheme. This approach minimizes the impact on performance (as opposed to an independent scheme that is applied later on). The basic idea behind it is to determine the "unavoidable" load of each process during each merge, i.e. count the number of non-duplicate pages each process has to store. Using this knowledge, each duplicate page can then be assigned to the least loaded process that holds a copy, who becomes its owner. Such an assignment is not permanent: a page can change its owner multiple times before the end of the reduction phase, dynamically adapting to the new load balancing requirements as the reduction converges towards the end result. A detailed algorithmic description of how this works in provided in Section III-C.

B. Architecture

The simplified schema of a 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 allocated memory regions are monitored by the *page manager*, which exposes the CHECKPOINT primitive to the application. The application process calls this primitive whenever it wants to dump the protected memory regions to storage as part of a new checkpoint. For transparency reasons, it is possible to trigger the CHECKPOINT primitive from outside of the application, by raising a specific signal for which the page manager registers a handler.

The page manager is designed in a modular fashion such that it is easy to plug in different storage back-ends where pages can be committed. These can range from more conventional approaches like parallel file systems (e.g. *PVFS* [23]) or cloud storage repositories (e.g. *Amazon S3* [24]) to specialized storage systems [25] and even local storage. Local storage is particularly attractive, because it is much faster and more scalable compared to conventional approaches. However, it is prone to failures and thus unreliable. Nevertheless, there is a wide range of proposals that can complement our proposal with a scalable persistency solution: erasure code protection schemes [12], multi-level checkpointing [9], virtual disk snapshotting [26], etc.

C. Zoom on the deduplication process

The application can request a new checkpoint at any moment using the CHECKPOINT primitive (listed in Algorithm 1), which initiates a three-stage procedure.

In the first stage, a local deduplication is performed over the set of memory pages that the application needs to save ($MemoryPages$). This is done by hashing the contents of each page into the $LocalHashes$ set, which holds unique hash values corresponding to unique pages with high probability.

Each page has a corresponding $Reference$ entry, which points to the rank of the process that "owns" the page, i.e. needs to store a physical copy (whose reference is by convention **nil**). Initially, after the local deduplication step,

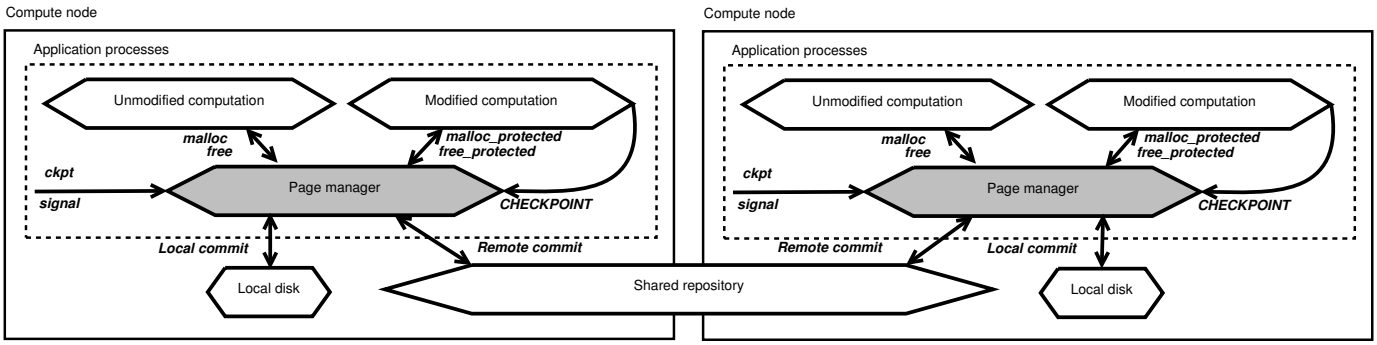


Fig. 1. Distributed architecture that integrates our approach via the page manager (dark background).

Algorithm 1 Checkpoint the memory contents given by the $MemoryPages$ set

```
1:  procedure CHECKPOINT
2:      LocalHashes ← ∅
3:      for all p_i ∈ MemoryPages do
4:          h_i ← HASH(p_i)
5:          if h_i ∉ LocalHashes then
6:              LocalHashes ← LocalHashes ∪ {h_i}
7:              Reference[p_i] ← nil
8:          else
9:              Reference[p_i] ← MyRank
10:         end if
11:     end for
12:     HashFreq ← ∅
13:     for all h_i ∈ LocalHashes do
14:         HashFreq ← HashFreq ∪ {(h_i, 1, MyRank)}
15:     end for
16:     TopK ← ALL_REDUCE(HashFreq,HASH_MERGER)
17:     for all (h_i, c_i, r_i) ∈ TopK do
18:         if h_i ∈ LocalHashes and MyRank ≠ r_i then
19:             Reference[p_i] ← r_i
20:         end if
21:     end for
22:     for all p_i ∈ MemoryPages do
23:         if Reference[p_i] = nil then
24:             write (HASH(p_i), Contents[p_i])
25:         else
26:             write (HASH(p_i), Reference[p_i])
27:         end if
28:     end for
29: end procedure
```

each reference is either **nil** or the rank of the local process ($MyRank$).

Once this step has completed, the collective deduplication stage is initiated. The goal of this stage is to build the set of $Threshold$ most frequently occurring pages throughout all processes (denoted $TopK$), establishing for each such page its owner. There are two key challenges here: (1) how make this stage as efficient as possible; (2) how to achieve load balancing (i.e. evenly distribute the amount references among the ranks).

To this end, we use tuples of the form (h_i, c_i, r_i) to describe elements of $TopK$, where h_i is the hash corresponding to the contents of the page, c_i is the number of times h_i was encountered by all processes and r_i is the rank responsible to store it. Initially, we build a set of tuples $(h_i, 1, MyRank)$ (denoted $HashFreq$) for each process out of their corresponding $LocalHashes$. Then, we reduce all $LocalHashes$ into $TopK$, while performing the load balancing between the ranks during the reduce and finally disseminate $TopK$ to all ranks so that it can be used for further deduplication.

To address the first challenge, we optimize the reduce by performing it as a collective ALL_REDUCE operation among all ranks. This is a popular pattern, functionally equivalent to a reduce operation on a root process (using for example recursive doubling) which is then followed by a broadcast of the end result to all other processes. Active research is done [27], [28] to efficiently deal with this pattern depending various parameters (such as message size, whether the number of processes is a power of two or not, etc.), however a contribution in this direction is outside the scope of this paper.

To address the load-balancing challenge, we propose a dedicated merging algorithm ((listed in Algorithm 2) to be used in the reduce. This algorithm merges two sets of tuples ($Left$ and $Right$) of the form (h_i, c_i, r_i) into a single set ($Result$) that has a maximum of $Threshold$ elements. Eventually, the final result of the reduce becomes $TopK$.

To keep track of the load assigned to each process, we use the $PageLoad[rank]$ structure, which is initially 0 for each process. Obviously, there is no choice of assignment for those entries that do not appear both in $Left$ and $Right$. For this reason, they are added to $Result$ and the $PageLoad$ of their corresponding rank is incremented. Next, the entries that appear both in $Left$ and $Right$ are processed: if there is a choice (i.e. their rank in $Left$ and $Right$ differs), the final rank to store the page is the one whose $PageLoad$ is minimal. Note that the order of doing this is crucial: if the pages for which there was a choice were processed first, this would have left limited freedom for the rest and thus would have potentially caused load imbalance. Finally, after all entries have been processed, in a final step $Result$ is truncated to hold only the top $Threshold$ entries according to decreasing c_i.

Algorithm 2 Determine the $Threshold$ most frequent pages and the node responsible for each page

```
1: function HASH_MERGER(Left, Right)
2:     for all i ∈ Ranks do
3:         PageLoad[i] ← 0
4:     end for
5:     Result ← ∅
6:     for all (h_i, c_i, r_i) ∈ Left| ∄(h_i, _, _) ∈ Right do
7:         Result ← Result ∪ (h_i, c_i, r_i)
8:         PageLoad[r_i] ← PageLoad[r_i] + 1
9:     end for
10:    for all (h_i, c_i, r_i) ∈ Right| ∄(h_i, _, _) ∈ Left do
11:        Result ← Result ∪ (h_i, c_i, r_i)
12:        PageLoad[r_i] ← PageLoad[r_i] + 1
13:    end for
14:    for all (h_i, c_i, r_i) ∈ Left|∃(h_i, c_j r_j) ∈ Right do
15:        if PageLoad[r_i] < PageLoad[r_j] then
16:            r ← r_i
17:        else
18:            r ← r_j
19:        end if
20:        Result ← Result ∪ (h_i, c_i + c_j, r)
21:        PageLoad[r] ← PageLoad[r] + 1
22:    end for
23:    while |Result| > Threshold do
24:        Result ← Result \ {(h_i, c_i, r_i)|c_i is minimal }
25:    end while
26:    return Result
27: end function
```

After all processes received $TopK$, in a final stage the $Referece$ of each memory page that has a corresponding entry in $TopK$ is adjusted to match the rank from that entry. With the final $Reference$ established for each page, the pages can now be dumped to storage. For those pages that hold a non-nil reference, only the rank and hash value is stored. This is enough to fetch a full copy of the page on recovery.

For the rest of this section, we briefly analyze the complexity of the proposed algorithms. Since both the first and third stage iterate through the memory pages, their asymptotic upper bound is $O(np)$, where np is the number of pages (we assume HASH and **write** take $O(1)$ as the page size is a constant). With respect to the second stage, first let's analyze the asymptotic upper bound of HASH_MERGER. Since its result is bounded by $Threshold$, the union of $Left$ and $Right$ is limited (except for the first time) to a total of $2 \cdot Threshold$ elements in the most defavorable case. Furthermore, selecting the top $Threshold$ most frequent elements out of $2 \cdot Threshold$ elements involves either partial sorting or insertions in heaps/balanced trees, thus we obtain $O(lg\ Threshold \cdot Threshold)$. Since ALL_REDUCE is logarithmic in the total number of processes (denoted n), for the second stage we finally obtain $O(lg\ n \cdot lg\ Threshold \cdot Threshold)$. Thus, the overall asymptotic upper bound for CHECKPOINT is $O(np + lg\ n \cdot lg\ Threshold \cdot Threshold)$.

D. Dealing with collisions

The algorithms presented in the previous section make an important assumption: it is enough to compare the hash values of two memory pages in order to conclude that their contents is the same and thus save only one copy. Obviously, this assumption does not hold, as the range of hash values is much smaller than the range of all possible page contents. Thus the scenario where two pages have different contents but equal hash values is possible. Such a scenario is called a *collision*. Since collisions cause our approach to save different checkpointing data than originally intended, it is important to analyze their impact. In this section we show that collisions are seldom enough for our assumption to hold in practice.

The problem of collisions is well studied in the field of cryptography. It can modeled through the *birthday problem*, an actively researched mathematical problem [29] with a broad spectrum of applications. The birthday problem raises a simple question with a non-trivial answer: what is the probability that at least two out of n people share the same birthday? Obviously, if we have more than 365 people, we can be sure to find two that share the same birthday. Otherwise, it can be shown that the probability is

$$p(n) = 1 - \prod_{k=1}^{n-1}(1 - \frac{k}{365})$$

Surprisingly, it takes far less than 365 people to reach a probability of 99%: only 57 are necessary. This alarming result threatens our assumption that collisions are seldom. However, luckily for our context, we can afford a much broader range of values for hashes than there are days in a year. For example, if we assume that the HASH function used in the previous section generates a uniform spread of hash values and the hash size is 256 bits, then the hash space is $2^{256} \approx 1.2 \cdot 10^{77}$. Under these circumstances, we need no less than $4.8 \cdot 10^{37}$ pages to reach a probability as small as 1%. Considering that the default page size of the operating system is 4 KB, we need $1.92 \cdot 10^{23}$ Exabytes worth of checkpointing data for collisions to happen in practice. Not only does this outmatch the capabilities of an exascale machine by 23 orders of magnitude, but at such high amounts of data, silent errors such as bit-flips in memory are more likely to lead to erroneous results than a restart from a checkpoint where a collision happened, which prompts the need for additional protection mechanisms at application level anyway.

Thus, we argue that working with hash values instead of memory page contents is safe enough in practice to eliminate the need of dealing with collisions. Nevertheless, for theoretical completeness it is worth mentioning that our algorithms can be easily extended to check for collisions at the expense of checkpointing overhead.

IV. Implementation

Our prototype implements the *page manager* as a library that exposes the CHECKPOINT primitive to the application.

The application will make use of specialized dynamic memory allocation/deallocation routines, i.e. malloc_protected and free_protected. These routines can be used to control directly at application-level what memory contents is needed on restart, enabling our approach to work in user-defined checkpointing mode.

On top of this library, we implemented a second library that transparently traps normal malloc and free calls in order to deal with applications for which user-defined checkpointing is difficult to manage or for which memory management is outside of user control (e.g. because they are written in higher level languages). To this end, we built or own custom memory allocator on top of *jemalloc* [30], a scalable high performance malloc implementation designed to efficiently support concurrent allocations. The application itself needs not necessarily be recompiled and linked against this second library, as it is enough to preload the library in order to replace the standard system malloc implementation.

The page manager was implemented 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 III-C with minimal overhead. For the implementation of the ALL_REDUCE abstraction we rely on MPI through its corresponding Boost interface for collective operations. Finally, we have chosen a conservative hash function to obtain digests for the contents of the memory pages: *SHA1*. Depending on the application, a better speed vs. collision chance trade-off might be desired, which is fully supported by our approach without further modifications.

V. Evaluation

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

A. Experimental setup

We use two experimental testbeds for our evaluation: *Shamrock* and *Grid'5000*. Shamrock is the experimental platform of the Exascale Systems group of IBM Research in Dublin, Ireland. It 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 25 nodes. Grid'5000 is an experimental testbed for distributed computing that federates nine sites in France. We used 32 nodes (interconnected with Gigabit Ethernet) of the Reims site, each of which is equipped with 2 AMD Opteron 6134 x86_64 CPUs (12 cores), for a total of 24 cores per node. Furthermore, each node is equipped with 48 GB of RAM (2 GB/core) and HDD local storage of 217 GB.

With respect to the software configuration, we use OpenSSL 1.0.1 for SHA1 calculations and Boost 1.51 for all our data structures and MPI calls. The MPI library installed is MPICH2 1.4.1. Finally, the memory page size used throughout our experiments is fixed at 4 KB, the default of the operating system (up-to-date Debian Sid distribution).

B. Methodology

We compare four checkpointing approaches throughout our evaluation:

a) Checkpointing through local and collective deduplication: This is our approach. It implements all functionalities of the CHECKPOINT request described in Section III-C), performing a local deduplication of memory pages followed by a collective inter-process deduplication step. Each process dumps the remaining set of unique pages to local storage. For the rest of the paper, we refer to this setting as our−approach.

b) Checkpointing through local deduplication only: This setting is very similar to our−approach except for the collective inter-process deduplication step. Implementation wise, this is achieved by omitting the call to ALL_REDUCE inside the CHECKPOINT request. A comparison with this approach is highly relevant because it enables us to isolate the impact of ALL_REDUCE and analyze its benefits. For the rest of the paper, we refer to this setting as local−dedup.

c) Full checkpointing: In this setting, each process dumps its full set of memory pages to local storage before returning control to the application. This is the most common form of checkpointing used in practice and thus an important approach to compare against. For the rest of the paper, we refer to this setting as full−dump.

d) Incremental checkpointing through page-tracking: In this setting, each process monitors write accesses to memory pages between consecutive checkpoints in order to build an incremental set of differences that is dumped to local storage. Unlike the previous setting, untouched memory pages will not unnecessarily be saved. Monitoring is achieved by write-protecting all memory pages between consecutive checkpoints (or the beginning) and trapping first time writes through the generated SEGFAULTs. After the first time write is recorded, the write-protection is removed and the application is allowed to continue. Incremental approaches are increasingly being adopted in practice and thus make a comparison relevant. We denote this setting as inc−tracking for the rest of this paper.

These approaches are compared based on the following metrics:

- *Average amount of checkpointing data per process*: is the storage space consumed by each process on the average to save its memory pages (unique or not). This metric reflects the overall efficiency of each of the approaches in identifying and avoiding unnecessary saves of memory pages. To obtain the total amount of checkpointing data saved by all processes, simply multiply this by the number of processes. Obviously, smaller values are better.
- *Maximal amount of checkpointing data per process*: is the storage space consumed by the most loaded process to save its memory pages (unique or not). This metric is important because it shows to what extend I/O load balancing succeeds in distributing the I/O load across the processes. A small value means better I/O load balancing,

and thus less performance penalty due to staggers (i.e. slow processes that take a long time to checkpoint due to heavy load).

- *Impact on application performance*: is the performance degradation perceived by the application with checkpointing enabled, as compared to the baseline (i.e. the case when no checkpointing is performed). This metric reflects the end-benefits of paying for the techniques employed by each of the approaches to avoid unnecessary saves of memory pages. Again, smaller values are better.

C. Synthetic benchmarks

Our first series of experiments aims to push our−approach to the limit in order to better understand its behavior and the different trade-offs involved. To this end, we implemented a simple benchmarking process that allocates a fixed-sized memory region (1 GB) through malloc−protected, fills it with a predefined pattern, and then invokes CHECKPOINT. Such benchmarking processes are replicated on an increasing number of cores belonging to the Shamrock cluster in order to study the scalability. The number of benchmarking processes is increased in multiples of 12, so as to fill the capacity of each compute node completely and thus generate the maximum amount of I/O pressure to local storage. To isolate the behavior of CHECKPOINT and guarantee that it is called at the same moment by all benchmarking processes, we introduced a barrier before its call.

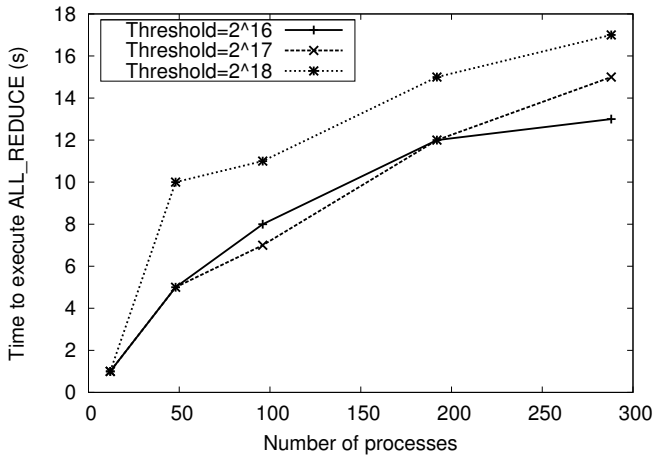


Fig. 2. Scalability of ALL_REDUCE in the worst case scenario: Each process fills its memory region in a fashion that guarantees globally unique pages and then checkpoints to local storage (1 GB per process, 12 processes per node)

Since most of the performance penalty incurred by our approach is generated by the ALL_REDUCE stage, in a first step we evaluate its worst-case scenario scalability, i.e. when all memory pages written by all benchmarking processes are guaranteed to be unique, which forces the result of HASH_MERGER to reach the $Threshold$ for each call. We vary the $Threshold$ for three different values: 2^{16}, 2^{17} and 2^{18}. The results obtained for this configuration are illustrated in Figure 2. As expected, with increasing number of processes and increasing $Threshold$, the performance penalty grows. However, the shape of the curves exhibit a slower growth with increasing number of processes, which is consistent with our algorithm analysis presented Section III-C, where we show a logarithmic asymptotic trend (considering a fixed $Threshold$).

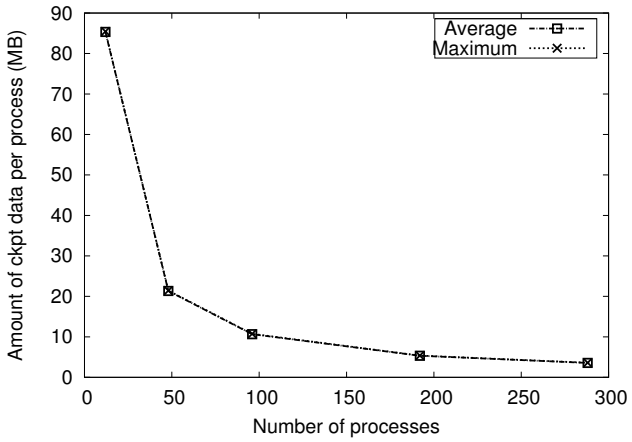
Next, we evaluate the impact of collective deduplication in a favourable scenario: when each benchmarking process writes the same set of unique pages (i.e. each page is globally duplicated by all processes but has no local duplicate). In this case, the result of HASH_MERGER remains constant at 2^{18} tuples at all times (i.e. one tuple for each page of the 1 GB region). Thus, we fixed $Threshold$ to this value. The cost of ALL_REDUCE is in this case the same as for the worst-case scenario presented above, hence it was not explicitly illustrated. More interesting however is the overall cost of CHECKPOINT. As can be observed in Figure 3(b), all other approaches except our−approach perform very closely and have a near-constant cost in terms of execution time, which can be traced back the same amount of checkpointing data that they write, a step that overshadows page-tracking or hash calculation costs.

On the other hand, our−approach successfully identifies and eliminates the duplicates across all processes, drastically reducing I/O pressure on each node and thus obtaining consistent reduction in execution time of at least 85%. This is consistent with the trend observed in Figure 3(a), where the average amount of checkpointing data per node experiences a sharp drop due to deduplication. However, on its own this is not enough to explain the reduction of I/O pressure: if only one process were to save the pages while the others used references, the total execution time would not have differed much compared to the other approaches. It is at this point where it becomes obvious that the load balancing strategy of our−approach plays a major role in this success: we can observe a perfect overlap between the maximum amount of checkpointing data per node and the average amount, which demonstrates an even distribution of the I/O workload among the processes.

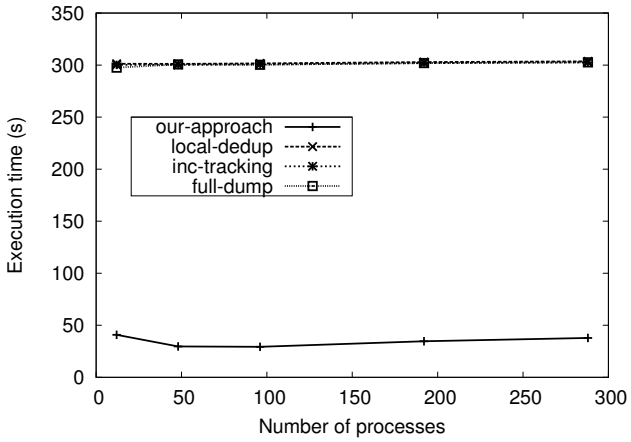
D. Case study: CM1

Our next series of experiments focuses real life high performance computing applications. We illustrate the benefits of our approach for one such application: *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



(a) Average and maximum amount of checkpointing data saved per process



(b) Total time for all benchmarking processes to finish execution

Fig. 3. Scalability in the best case scenario: Each process fills its memory region in the same predefined fashion and then checkpoints to local storage (1 GB per process, 12 processes per node, $Threshold = 2^{18}$)

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.

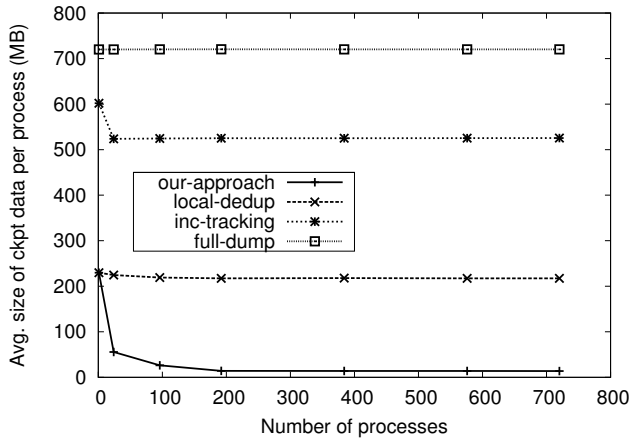
Since CM1 is written in Fortran, we had to implement a minimalist wrapper library for Fortran. Using this library, we replaced the hand-optimized synchronous checkpointing implemented in CM1 with a simple call to CHECKPOINT, while relying on our library to transparently capture all memory allocations.

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 [31]. The experiment studies the weak scalability of our approach on the Reims cluster 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. To this end, we gradually increase the number of MPI processes in multiples of 24, so as to fill the capacity of each compute node completely (i.e. one MPI process per core with no spare cores left). Given this configuration, each MPI process allocates 728 MB, out of which approx. 500 MB of memory content is changed. We set the checkpointing frequency to 30s of simulated time and the total simulation time to 70s, which is enough to trigger two checkpoints. For our−approach, we fixed the $Threshold$ to 2^{17}.

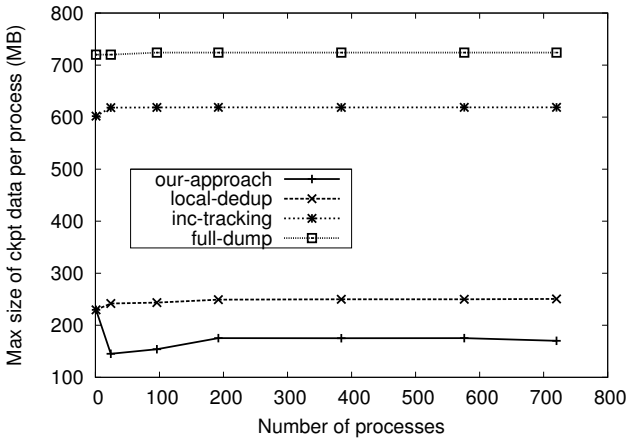
Results are shown in Figure 4. With respect to the average amount of memory saved per process due to checkpointing (Figure 4(a), as expected we observe a constant trend for full−dump, because all processes write the whole pre-allocated memory region responsible to hold their subdomain and auxiliary data structures. Analyzing the curve for inc−tracking, we observe that with increasing number of processes, the average amount stabilizes at 71% of the full memory allocation per process. What this means is that a big portion of in-memory data remains read-only throughout the computation, which enables inc−tracking to obtain substantial reductions of checkpoint sizes. Interesting to note is the jump from one to many processes: a drop in average checkpoint size is noticeable due to less management overhead (i.e. extra data structures for rank 0) on each node. The same effect is noticeable for local−dedup, however almost to an imperceptible extend: this happens because deduplication already succeeds at eliminating a large quantity of duplicated data, stabilizing at average amount of 28% of the full memory allocation per process.

This hints to an important conclusion: a large number of memory pages that are modified by the application as a result of its computation either do not change from the previous iteration or they change "collectively" to the same content, which makes sense if nothing interesting is happening in the subdomain (e.g. the hurricane is moving to other parts of the analyzed space). In support of this comes our−approach: thanks to collective inter-process deduplication, we are able to take advantage of this behavior globally and thus reduce the average amount of checkpointing data per process to almost 2% of the original memory allocation when reaching 788 processes. Notice how fast this reduction occurs: while for one process our−approach starts at the same level as local−dedup, for 192 processes the average is already 15x smaller compared to local−dedup alone. Putting everything in perspective at global level, our−approach obtains for 788 processes a total checkpointing size of just 11 GB, compared to 170 GB obtained by local−dedup, 469 GB obtained by inc−tracking and 564 GB obtained by full−dump.

Analyzing the maximum amount of checkpointing data saved per process (Figure 4(b), we observe a steady tendency for full−dump, inc−tracking and local−dedup. Except full−dump, a noticeable increase is observable with an increasing number of processes, due to differences in the result of computations (i.e. some processes are involved in compu-



(a) Average amount of checkpointing data saved per process (lower is better)



(b) Maximal amount of checkpointing data saved per process (lower is better)

Fig. 4. Weak scalability of CM1: Two consecutive checkpoints are saved to local storage (728 MB of memory per process and 24 processes per node)

tations where something "dramatic" happens in their subdomain). This effect is particularly observable for our−approach, where we see the maximum stabilizing at 50% of local−dedup. This result demonstrates the effectiveness of our load balancing strategy: thanks to even distribution of the I/O workload for the duplicated memory pages, it is possible to drastically reduce the pressure on the processes that are responsible for more intensive subdomains.

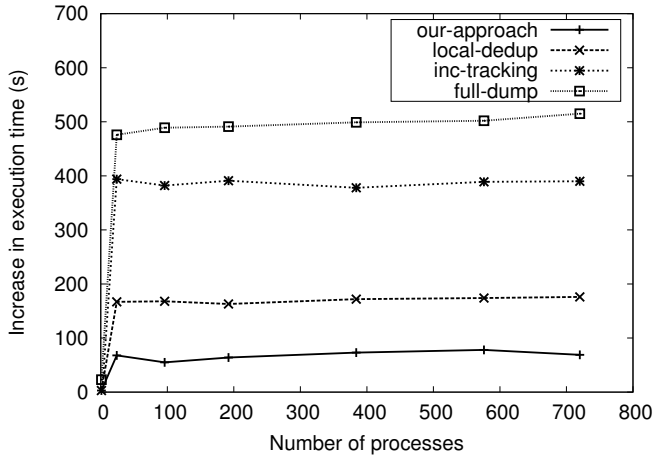


Fig. 5. Weak scalability of CM1: Increase in execution time due to checkpointing compared to baseline (lower is better)

Ultimately, both the global reduction of checkpointing data and the load balancing strategy have a decisive role in the overall reduction of the checkpointing overhead perceived by the application. This is shown in Figure 5, where the total increase in application execution time (compared to the scenario where CM1 runs without checkpointing) is depicted. As can be observed, all four approaches exhibit a stable trend and thus are scalable. Except our−approach, the explanation is simple: we use local storage and thus the I/O pressure remains constant. In the case of our approach, this result is even more interesting: despite rising cost of collective deduplication, thanks to our logarithmic proposal and increasing duplicates that are identified and eliminated, a stable tendency is observable too. When compared to local−dedup, inc−tracking and full−dump, this ultimately translates to an overall performance reduction of 60%, 80% and 85% respectively.

VI. Conclusions

Checkpoint-Restart (CR) is a key method to provide fault tolerance for large-scale HPC applications. With increasing scale, checkpointing data explodes and thus leads to high CR performance overhead and resource utilization. Thus, it is important to attempt to reduce the checkpoint sizes.

In this paper, we have proposed a run-time system for distributed HPC applications that checkpoints dynamically allocated memory in a scalable fashion by leveraging deduplication. Unlike previous approaches that are confined to the memory space of a single process, we introduced a inline collective deduplication scheme that is able to identify and eliminate identical pages across *different* processes in a balanced fashion that evenly distributes the I/O load globally for the remaining unique pages that need to saved.

We demonstrated the benefits of our approach through experiments that involve dozens of nodes equipped with local storage and a high core count, using both benchmarks and a real HPC stencil application. Compared to three state-of-art approaches, in the real world our approach reduces the storage requirements between 93%-98% and achieves excellent load balancing, with at least a 50% less difference between the maximum and average I/O load for each process. This ultimately translates to 60%-85% less performance overhead. To align with recent trends, all these benefits were demonstrated using local storage: we predict even better results when using parallel file systems or other remote shared storage systems. Interesting to note is the access pattern that made these results

possible: the deduplicated contents does not mostly originate from immutable data such as copies between iterations (which is why incremental page tracking showed limited success), but rather from computations that generate identical results even across different processes (which is why local deduplication was not enough).

Encouraged by these results, we plan to broaden the scope of our work and experiment in future work with more applications. We are also looking at the possibility of dynamically adapting the threshold to better match run-time requirements. Furthermore, we explored deduplication only in space and not in time. Thus, extending our approach to look at past checkpoints for duplicated contents has the potential to improve our results even further.

ACKNOWLEDGMENTS

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/).

REFERENCES

[1] "Top 500 supercomputing sites," http://top500.org.

[2] S. Ashby, P. Beckman, J. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, P. Messina, T. Mezzacappa, P. Moin, M. Norman, R. Rosner, V. Sarkar, A. Siegel, F. Streitz, A. White, and M. Wright, "The Opportunities and Challenges of Exascale Computing," US Department of Energy, Tech. Rep., 2010.

[3] 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," DARPA, Tech. Rep., 2008.

[4] 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.*, vol. 34, pp. 375–408, September 2002.

[5] 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*, Tuscaloosa, USA, 2012, pp. 262–267.

[6] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel, "A Study on Data Deduplication in HPC Storage Systems," in *SC '12: 25th International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, USA, 2012.

[7] L. Alvisi and K. Marzullo, "Message Logging: Pessimistic, Optimistic, Causal, and Optimal," *IEEE Trans. Softw. Eng.*, vol. 24, no. 2, pp. 149–159, Feb. 1998.

[8] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic MPI Applications," in *IPDPS '11: 25th IEEE International Parallel and Distributed Processing Symposium*, 2011, pp. 989–1000.

[9] 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*. New Orleans, USA: IEEE Computer Society, 2010, pp. 1–11.

[10] 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.*, vol. 8, no. 2, pp. 6:1–6:29, Jun. 2011.

[11] 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, pp. 155–163.

[12] 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*. Seattle, USA: ACM, 2011, pp. 32:1–32:32.

[13] 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*, Rhodes, Greece, 2012, pp. 313–324.

[14] D. Ibtesham, D. Arnold, K. B. Ferreira, and P. G. Bridges, "On the viability of checkpoint compression for extreme scale fault tolerance," in *Euro-Par Workshops (2)*, Bordeaux, France, 2011, pp. 302–311.

[15] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 174–187, Oct. 2001.

[16] M. Rabin, "Fingerprinting by random polynomials," Center for Research in Computing Technology, Harvard University, Tech. Rep. TR-CSE-03-01, 1981.

[17] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*. San Jose, USA: USENIX Association, 2008, pp. 18:1–18:14.

[18] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, "Hydrastor: a scalable secondary storage," in *FAST '09: Proccedings of the 7th conference on File and storage technologies*. San Francisco, USA: USENIX Association, 2009, pp. 197–210.

[19] 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*. St. Malo, France: ACM, 2004, pp. 277–286.

[20] 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*, 2011, pp. 69–79.

[21] 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*, Seattle, USA, 2005, pp. 9:1–9–14.

[22] 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*. Santorini, Greece: Springer-Verlag, 2011, pp. 272–281.

[23] 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*, Atlanta, USA, 2000, pp. 317–327.

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

[25] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie, "BlobSeer: Next-generation data management for large scale infrastructures," *J. Parallel Distrib. Comput.*, vol. 71, pp. 169–184, February 2011.

[26] 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*, Seattle, USA, 2011, pp. 34:1–34:12.

[27] B. Tu, J. Fan, J. Zhan, and X. Zhao, "Performance analysis and optimization of mpi collective operations on multi-core clusters," *The Journal of Supercomputing*, vol. 60, no. 1, pp. 141–162, 2012.

[28] A. Mittal, N. Jain, T. George, Y. Sabharwal, and S. Kumar, "Collective algorithms for sub-communicators," in *ICS '12: Proceedings of the 26th ACM international conference on Supercomputing*. New York, NY, USA: ACM, 2012, pp. 225–234.

[29] D. Brink, "A (probably) exact solution to the Birthday Problem," *The Ramanujan Journal*, vol. 28, pp. 223–238, 2012.

[30] J. Evans, "A scalable concurrent malloc(3) implementation for freebsd," in *In Proceedings of BSDCan 2006*, Ottawa, Canada, 2006.

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