Application-Level Differential Checkpointing for HPC Applications with Dynamic Datasets

Kai Keller and Leonardo Bautista Gomez
Barcelona Supercomputing Center (BSC-CNS)
Barcelona, Spain
{kai.keller, leonardo.bautista}@bsc.es

***Abstract*—High-performance computing (HPC) requires resilience techniques such as checkpointing in order to tolerate failures in supercomputers. As the number of nodes and memory in supercomputers keeps on increasing, the size of checkpoint data also increases dramatically, sometimes causing an I/O bottleneck. Differential checkpointing (dCP) aims to minimize the checkpointing overhead by only writing data differences. This is typically implemented at the memory page level, sometimes complemented with hashing algorithms. However, such a technique is unable to cope with dynamic-size datasets. In this work, we present a novel dCP implementation with a new file format that allows fragmentation of protected datasets in order to support dynamic sizes. We identify dirty data blocks using hash algorithms. In order to evaluate the dCP performance, we ported the HPC applications xPic, LULESH 2.0 and Heat2D and analyze them regarding their potential of reducing I/O with dCP and how this data reduction influences the checkpoint performance. In our experiments, we achieve reductions of up to 62% of the checkpoint time.**

***Index Terms*—Fault Tolerance, Differential Checkpointing, Incremental Checkpointing, Multilevel Checkpointing**

I. Introduction

High-performance computing (HPC) is a major tool for scientific research and fast industrial development. Supercomputers have observed an exponential increase in size and performance over the last couple of decades. Exascale computing (i.e., 10^{18} floating point operations per second) is the next frontier and it promises to bring orders of magnitude more computing power into the hands of scientists. However, the exponential increase in computational power also comes with a certain number of challenges; for instance, power consumption and resilience are among the most pressing issues that need to be addressed to reach such extreme computing scales. Indeed, the increasing number of components in large-scale systems makes the machine more prone to failures, reducing the mean time between failures (MTBF). At the same time, the amount of data used in large HPC simulations is increasing exponentially. Failures in supercomputers are usually handled through checkpoint and restart, by storing the state of the computation in reliable storage, so that the application can restart from the last saved state upon a failure. Unfortunately, the reduction in the MTBF forces users to checkpoint at a higher frequency to reduce the amount of re-computation to be done in case of failure. Simultaneously, the checkpoint takes more time as the amount of data to save increases. This leads to a steep reduction in system efficiency. In order to maintain high productivity in supercomputers and large data centers, it is important to reduce as much as possible the amount of data to be checkpointed to reliable storage.

Part of the research presented here has received funding from the European Unions Seventh Framework Programme (FP7/2007-2013) and the Horizon 2020 (H2020) funding framework under grant agreement no. H2020-FETHPC-754304 (DEEP-EST). The present publication reflects only the authors' views. The European Commission is not liable for any use that might be made of the information contained therein.

Differential checkpointing has been proposed in order to avoid re-writing checkpoint data that is identical between two consecutive checkpoints (i.e., no change of data). Previous research works have attempted to implement such a technique by tracking dirty memory pages in the system and only updating those within the checkpoint (CP) files. While this method works, it is not always efficient as many applications do re-write the exact same content (e.g., zero) into the same memory cells. From the OS perspective, these memory pages have changed as they are dirty, but in reality the content has not changed. Hashing the memory pages to detect real changes has also been proposed. Unfortunately, this technique also fails to detect unmodified datasets with dynamic sizes (e.g., particles moving between domains) or datasets relocated in memory.

In this paper we have implemented a hash-based strategy in which we partition the application datasets (not the memory pages) in blocks and keep track of the changes of each block by comparing the corresponding hashes. In addition, we introduce a new file format that is capable of recognizing changes in data blocks and simultaneously dynamically adapt to changes in the size of the protected structure. We evaluate the collision robustness of multiple hash algorithms and show that MD5 and CRC32 are viable solutions for differential checkpointing. We integrate our implementation into the multilevel checkpointing library FTI and evaluate it with three HPC applications. In our measurements, we obtain up to 62% reduction in checkpointing time in comparison to traditional checkpointing. Furthermore, we propose a theoretical model that predicts performance gains that could be obtained with our dCP technique.

The rest of this paper is organized as follows. Section II introduces the terminology of this paper. Section III discusses related work. Section IV introduces our implementation of dCP. Section V explores the robustness of different hashing algorithms. Section VI presents our analytical model to predict performance gains. The results of our large-scale evaluation are presented in Section VII. Section VIII discusses the strong points and limitations of this proposed technique and finally, Section IX concludes this paper.

II. Terminology

The term *incremental checkpointing* is used in the literature to denote two different processes. To avoid confusion we would like to clarify what we refer to when we use the terms incremental and differential checkpointing.

a) Definition of Incremental Checkpointing: We refer to incremental checkpointing as to be the *incremental completion of a CP file*. This technique serves primarily to avoid overhead caused by oversaturated network channels. It may be used within applications that provide datasets, that define the current program state, at different times. Thus, instead of writing the whole CP data at once, it is incrementally written during some period of time, which reduces the stress on the network.

b) Definition of Differential Checkpointing: We refer to differential checkpointing as to be the *differential update of a CP file*. That is, the data blocks in the previous CP file that by the time of a subsequent CP differ to the corresponding data block of the current application state, will be replaced by the up-to-date data block. The rest of the blocks (i.e., those that did not change) will not be updated.

III. Related Work

Differential updates of data-states, data-dependencies or workflows exist in several disciplines of HPC. Depending on the case, there are various methods that allow detection and logging of differences in data-structures and workflows. An interesting example for such a logging mechanism and the differential update of CP data in data streaming applications is well described within the web documentation of Apache Flink [1]. The update mechanism is based on a so-called *log-structured-merge*. The data storage is based on a key-value pair. In order to update the CP files, the first and fast in-memory storage layer collects the updated keys inside a *memtable*. After a certain amount of data has been accumulated inside the memtable, the data is flushed to a stable storage. The flushed memtables are now called *sorted-string-tables* (sstables). At a certain point, the various sstables will be merged into one sstable. This is performed asynchronously to the streaming application execution by a dedicated process. The merge consolidates redundant keys from different sstables.

Another important work to mention is the differential (de-)serialization in SOAP implemented by Nayef Abu-Ghazaleh et al. [2]–[4]. SOAP (simple object access protocol) is a messaging protocol adequate for server communications. It can be used to negotiate between different application layer protocols that encode the messages into the XML format (e.g. HTTP, RPC or SMTP). SOAP is a promising candidate to negotiate between independent transfer protocols in high-performance parallel and distributed computing (HPCD) environments [5]. A bottleneck of the messaging workflow is the (de-)serializing of messages. Serializing refers to the conversion of in-memory data to ASCII text messages encoded in the XML format prior to the sending of a message and de-serialization refers to the reverse process after a message has been received. Nayef Abu-Ghazaleh et al. present a mechanism that uses checksums in order to identify redundant information in consecutive messages prior to the de-serialization. The similarities may be inside the message contents or within the encoding XML structure. Using this information, it is possible to skip de-serialization of unchanged message sequences.

Besides the two framework-specific examples from above and other specific implementations, non-specific implementations of dCP, e.g. as linkable libraries, exist at kernel level (compare C. Wang et. al [6] or R. Gioiosa et. al. [7]), i.e. transparent for the application developer and in form of compiler plugins that analyze C/R capabilities inside the application during compile time (see G. Bronevetsky et. al. [8]). However, kernel-level checkpointing is not always efficient and not much exist for HPC applications at user-level.

The library *libckpt* [9] can be operated almost transparently (i.e. without modifying the application code), but, it also provides API functions that enable the user to determine the CP behavior. The API permits to specify the CP data (i.e. explicitly exclude or include certain memory regions) or the CP location (i.e. where inside the application flow) and location of the CP files (i.e. path on the file system). In order to detect data updates between consecutive CPs, libckpt employs the UNIX page protection mechanism. The library has knowledge about the process virtual address space. All memory pages that correspond to the process address space are set to read only via a call to `mprotect` with the `PROT_READ | PROT_EXEC` flags specified. After this, every store operation to one of the protected pages will rise a segmentation fault signal (`SIGSEGV`). This signal is caught by libckpt and appropriately handled. The address of the page is then marked dirty and will be written to disk during the next CP. Every differential CP (Plank denotes this as incremental CP [9]) represents an extra file on the file system. However, the user may specify a parameter in order to restrict to a maximum number of files. When this amount is reached, the files are being merged into one. This strategy has two major drawbacks. First, by protecting the whole address space of the application, one incorporates data that is not necessarily needed for a successful restart. That means that one may expect CP files to be much larger than necessary, hence a higher checkpoint overhead. Second, many applications update continuously all the datasets, which does not imply that the value after the update differs from the one before (e.g., zeros in a domain). In this case, the page protection mechanism will not lead to a significant reduction of data in the dCP files missing the goal of such a feature.

Kurt B. Ferreira et al. [10], [11] developed the library *libhashckpt* on top of libckpt. Before the dirty memory pages are written to disk, the hashes of this pages are compared to the hashes that were generated by the time of the former CP. Only if the hashes differ, the page will be incorporated in the dCP update. The version of libckpt that is provided at [12] is restricted to 32-bit kernels and thus cannot be used on almost any cluster/supercomputer. Also, we were not able to acquire the library libhashckpt by any means in order to compare it with our implementation. Libhashckpt is the closest work to our proposal; nonetheless there are multiple differences. First, libhashckpt is based on classic PFS-based checkpoint-restart and not implemented in a multilevel checkpointing library with asynchronous checkpointing, which involves a number of differences (see Section IV). Second, libhashckpt produces a file for every checkpoint update, having to deal with a high number of files, which applies high stress on the metadata servers. Third, libhashckpt does not adapt well for applications with dynamic dataset sizes. Indeed, when datasets change in size they might be moved to other memory locations or force other datasets to shift in the memory space, this will look like a completely different dataset from the memory page perspective but in reality they are just the same datasets that have either been displaced or changed in size. This will force a complete rewrite of the full checkpoint data, missing once again the goal of differential checkpointing.

This paper addresses all those issues, making this proposal the only general purpose multilevel checkpointing library that implements a version of differential checkpointing that adapts to datasets with dynamic sizes through a user-level interface and that scales for large HPC applications.

IV. dCP Implementation in FTI

In section III we saw several examples of logging mechanisms that may detect and track differences in application states. The mechanisms can be divided into two categories: tracking dirty pages (i.e. pages that was accessed by a store operation) and tracking actual changes of data by checksum comparison. Apache Flink and libckpt apply the first category. SOAP and libhashckpt implement both strategies. Although the hash based strategy has an advantage over the dirty page approach, applying hashes over memory pages still has the disadvantage of lacking the application perspective, offering only a *black box* perspective over the data. Application-level interfaces allow us to see datasets on their own, and detect real changes, reagrdless of whether they move to another memory region.

FTI is an application-level checkpointing library, with an API that provides flexibility and allows user to flag datasets that need to be protected. In addition, FTI is a multi-level CP library that offers 4 levels of increasing reliability and FTI implements a dedicated process that performs post-processing work for the more reliable CP levels asynchronously to the application processes. In our implementation, the virtual address space of the datasets will be partitioned into blocks of size b. We create hashes of these blocks and keep them in memory. The hashes are created from the dataset representation in memory immediately after a successful CP and *before* the application continues its normal execution so that the hashes in memory belong to the state of the dataset that is stored in the CP files. The hashes are also applied before an asynchronous work (e.g., RS encoding) is done. FTI also creates hashes in order to ensure CP file consistency upon restart, but these two types of hashes are unrelated.

We do not adopt the method of creating new files for every dCP update. Instead, we take advantage of the existing FTI *head* feature [13], that is, we may assign work that is related to FTI processes (e.g., RS-encoding, flushing CP files from local storage to the PFS) to a dedicated process that can operate asynchronously to the application flow. In order to update the CP file safely, we create a copy of the file using the head process and update the copy. The former CP file is kept during the update in order to roll back to it when an error occurs during the update. After successful completion of the CP the old CP file will be removed. This procedure prevents the corruption of the CP data in the case of a failure during the update process and minimizes the number of files on the PFS which in turn reduces the stress on the metadata server.

A. Dealing with Dynamic Sizes of Datasets

In order to implement an efficient and scalable dCP mechanism, the FTI protected datasets need to be arranged in immutable blocks in the CP files. For protected datasets with steady sizes, this is accomplished naturally. However, FTI supports datasets with dynamic sizes. Thus, to maintain immutable positions inside the checkpoint file we need to allow fragmentation of the datasets and in order to read the files upon the restart, we need to store the dataset-file mapping.

In the current release, FTI stores metadata that is needed for the restart in separate files. These files are getting parsed using the Iniparser library [14]. We could potentially extend this practice in order to keep track of the data-file mapping. However, FTI writes one metadata file for each group of processes (consult [13] or [15] for details about the FTI library). In order to minimize the overhead, we decided to develop a file format for FTI, *FTI-FF*, that includes the metadata for the owning process within the file structure. The general file structure is shown in figure 1 (For a comprehensive description please visit [16]).

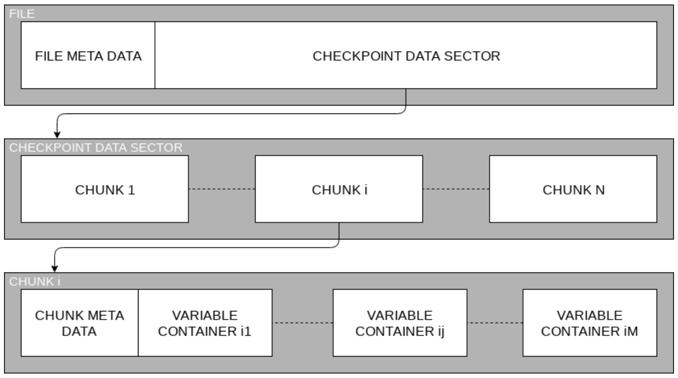

Fig. 1: FTI-FF structure: The *File Meta Data* contains information that is used by FTI for other purposes than differential checkpointing (e.g., RS-encoding). The *Chunk Meta Data* holds the file-mapping metadata for the dataset chunks stored in the current block (i.e., chunk-size, container-size, id)

FTI-FF Structure: The file structure is generated dynamically. By the time of the first successful CP, every dataset has created a virtual container with the current size of the dataset. The container is located at an immutable position in the CP file next to the corresponding metadata block. When a dataset increases its size in one of the successive CPs, hence exceeds the size of the first virtual container belonging to this dataset, another virtual container is created with the excess as its size. The new virtual container is appended at the end of the file, tailing the corresponding metadata block. This mechanism repeats every time a dataset exceeds the total size of the existing virtual containers. Once created, a container never changes its size. The total size of all the virtual containers belonging to the same dataset will be seen as contiguous and will be filled linearly. This may lead to sparse files when the data size shrinks.

B. Updating the CP Files

The *prime directive* we have to meet when implementing any dCP approach is that we must not update the data in the existing CP files directly. This is due to the danger of corrupting the file if an error occurs during the update.

In FTI we meet this goal by creating a copy of the CP file after the successful creation. The duplication is performed by a dedicated process asynchronously to the application run so that the application processes can continue the execution as if they would without dCP functionality. During the next CP, the processes may now update the copy directly. After the successful completion, the former file can be deleted. If the dedicated process cannot create the copy, the application processes will be notified and a complete CP will be created.

C. Tracking the differences

We mentioned earlier that the memory regions of the datasets will be partitioned into blocks of size b and that the content of the blocks is represented by hashes. If two hashes that correspond to the same block differ, we assume that the contents differ (*dirty* blocks) and if the hashes coincide, we assume that the contents are identical (*clean* blocks). We have to distinguish between blocks that are old (*valid*), i.e. present in the CP file, and blocks that are new (*invalid*), i.e. not present in the CP file (for instance, by the time of the first CP, all blocks are invalid). Invalid blocks will be added to the CP file without hash comparison.

In order to decide which data needs to be updated in the CP files, we apply the following set of rules:

(I) mark new blocks as invalid.
(II) identify dirty blocks during the dCP update.
(III) update the CP file with dirty or invalid blocks.
(IV) crate/update hashes for invalid/dirty blocks.

I: When datasets are exposed to FTI, the corresponding blocks are marked invalid to ensure that new datasets will be included in the CP file. The same applies when datasets increase their size and new data blocks are exposed to FTI. However, the hashes for the blocks will not be created yet.

II: During the dCP update, the processes request contiguous dirty regions by calling the function `FTI_ReceiveDcpChunk()`. A dirty region is the accumulation of adjacent dirty blocks. The function takes a pointer to the origin of the dataset and a size argument and compares sequentially the hashes of blocks with size b (user defined granularity). The function returns 1 and updates the pointer with the base address of a dirty region and sets the size of the region. 0 is returned if no dirty region was found. Invalid blocks cannot be compared in that sense since they do not have a representation inside the CP file. Hence, invalid blocks will be included in dirty regions ad-hoc.

III: `FTI_ReceiveDcpChunk()` is called inside a while loop and the CP copy is updated with the dirty regions returned by the function. The loop continues until the function returns 0, signaling that the dataset is now again up-to-date in the CP file. At the first CP, all blocks will be written.

IV: After the successful completion of the dCP update, the hashes that correspond to dirty or invalid blocks will eventually be updated (or created for invalid blocks) so that the hashes represent the actual state of the datasets in the current CP file[1]. Clearly, we keep the hashes of blocks that are neither dirty nor invalid untouched.

[1]One could think it would be more efficient to update the hash array during II, however, this would violate the prime directive since we cannot assure that the dCP update will be indeed successful.

The process is visualized in figure 2. The figure is divided in three sections separated by a dashed line. The left section corresponds to I and is implemented in function `FTI_Protect`. The function is used in FTI to register datasets in order to include them into the CP files. FTI creates metadata related to the dataset within this function. After the first call to `FTI_Protect`, all blocks of the corresponding datasets are marked invalid. After a subsequent call in order to increase the size of a dataset, blocks in the memory region that exceed the former size are new to FTI and thus consequently marked invalid as well. This ensures that new blocks are automatically included in the CP files. The middle section of the figure corresponds to II and III and the third section corresponds to IV. Hash creation (if invalid) or the update (if dirty), only happens in the third section after the successful completion of the dCP update (which corresponds to a full CP at the first call).

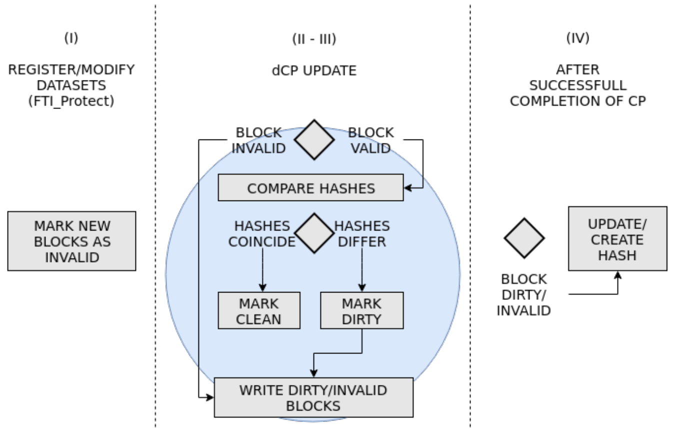

Fig. 2: dCP detection and update scheme. Processes left to the blue circle happen before and processes to the right after the dCP update. The circle indicates the dirty region request loop

The relevant metadata for the dCP mechanism is kept inside an array of the structure `struct FTIT_HashBlock`. The array has N elements, where N is the next greater integer of the dataset size divided by the block size (or this very value if the dataset size is a multiple of the block size). Every array element corresponds to one block of the partitioned dataset. The structure has three members: A boolean that indicates if the hash is valid, a boolean that indicates if the hash is dirty and the hash digest (either a 32-bit unsigned integer for CRC32 or a 128-bit unsigned char buffer for MD5).

V. Choice of the Hash Algorithm

Depending on the size of the protected datasets, the hash arrays might get significantly large. For instance, the MD5 digest length is 128 bits (16 bytes). Assuming a hash-block size of 128 bytes and 1GB of protected data per rank, we have to reserve 128MB of RAM for the hash metadata. In order to reduce this size, we can either increase the block size or decrease the digest size. The former may decrease the dCP performance due to the coarser resolution (i.e., more dirty block updates) and the latter may increase the risk for inconsistent CP files due to higher collision rates of the hash algorithm (i.e., when a collision occurs the block is considered clean despite the fact that the data in the block has changed).

In order to provide a small digest size, we tested three 32-bit hash algorithms (digest size 32 bits) upon performance and reliability. Adler32, Fletcher32 and CRC32. For completeness, we included also MD5 (digest size 128 bits) in the tests although it is considered to be reliable and fast for data integrity checks (despite its flaws in the cryptographic area [17]). The Adler32 and CRC32 checksums were calculated using the zlib data compression library [18], Fletcher32 was implemented using the recommendations in [19] and for MD5 we used the OpenSSL library [20].

Fletcher32 and Adler32 are both significantly faster than CRC32. However, both also have poor collision resistant characteristics for block sizes that are relevant in our case, as we will see below. To obtain a statement about the reliability of the checksums we performed a simple collision test. We focussed on the so-called *avalanche effect* [21], since in real applications it is very possible that elements of the datasets change only very little. The test follows algorithm 1.

Algorithm 1 Count hash collisions of modified buffers

```
repeat
    for all $b$ do
        populate $C_b$ with $N_b$ random u64 integers;
        create hashes $h_{C_b}$ of $C_b$;
        for all $p$ do
            for i=1, $N_b$ do
                $D_{b,i}$ = $C_{b,i} \oplus p$;
                Create hash $h_{D_{b,i}}$ of $D_{b,i}$;
                if $h_{D_{b,i}}$ == $h_{C_{b,i}}$ then
                    $c_{b,p}++$;        ▷ $c_{b,p}$ := Collision Counter
                end if
            end for
        end for
    end for
until N iterations
```

C_b and D_b are buffers that contain random integers, $b = \{2^i \,|\, 7 \leq i \leq 15\}$ denotes the hash block sizes and p denotes the patterns that are used to modify the elements of C_b. For N_b we have $b \bmod (N_b \times 64) == 0$. The elements of p correspond to bit flips of the last 1 ($p_0 = \texttt{0x1}$), 2 ($p_1 = \texttt{0x3}$), 4 ($p_2 = \texttt{0xff}$), 8 ($p_3 = \texttt{0xfff}$) and 16 ($p_4 = \texttt{0xffff}$) bits and to an arbitrary modification (p_5 is an arbitrary pattern) to simulate a random change.

Fletcher32 is commonly implemented with $M = 2^n$ or $M = 2^n - 1$ (M is the modulo value for the checksum. Consider [19] for implementation details). The case $M = 2^n - 1$ leads to identical checksums for buffers that differ only in one or more groups of two consecutive bytes that are all `0x00` in one and all `0xff` in the other buffer. For us, this is reason enough to disqualify the algorithm for its usage in dCP. Nevertheless, we included it in our measurements.

The results of the collision test are listed in table I. Adler32 and Fletcher32 exhibit a significant amount of collisions. Most of the collisions for Adler32 occurred for 1-bit or 2-bit flips and decrease for increasing block sizes. The collisions for Fletcher32 are homogeneously distributed for all modifications and block sizes. We estimate the collision rate of both algorithms, Adler32 and Fletcher32, as being too high in order to provide a sufficient level of reliability. MD5 and CRC32, on the other hand, did not show any collisions. The test we have performed is not appropriate to deliver a solid cryptographic statement about the reliability of CRC32 and MD5, however, it is enough to disqualify Adler32 and Fletcher32 for our purpose. Based on literature about CRC32 and MD5 (CRC32 is used in zlib and other cases to provide data integrity [18], [22], [23]) and based on our results we are quite confident about its application for dCP.

TABLE I: Collision rates (i.e. the probability of collision per iteration) achieved by application of algorithm 1. We did not detect any collision for CRC32 or MD5 and the collision rates for Fletcher32 mod(65535) were almost identical to Fletcher mod(65536). Thus, we do not list the results here. For all cases, the number of iterations have been within 160-180 million.

	p_0	p_1	p_2	p_3	p_4	p_5
b			ADLER32			
128	6.84e-3	1.42e-3	8.56e-5	3.68e-7	6.13e-9	1.23e-8
256	1.70e-3	3.46e-4	2.12e-5	8.59e-8	1.23e-8	1.23e-8
512	4.24e-4	8.69e-5	5.39e-6	1.84e-8	0	6.13e-9
1024	1.06e-4	2.21e-5	5.39e-6	1.84e-8	0	6.13e-9
2048	2.56e-5	5.21e-6	2.58e-7	0	6.13e-9	0
4096	6.23e-6	1.37e-6	9.20e-8	0	6.13e-9	0
8192	1.56e-6	2.70e-7	1.84e-8	6.13e-9	0	0
16384	3.56e-7	4.91e-8	4.29e-8	6.13e-9	0	0
32768	1.41e-7	7.98e-8	1.84e-8	0	0	0
			FLETCHER32 - MOD(65536)			
128	1.54e-5	1.47e-5	1.52e-5	1.52e-5	1.50e-5	1.54e-5
256	1.53e-5	1.55e-5	1.53e-5	1.54e-5	1.56e-5	1.54e-5
512	1.48e-5	1.56e-5	1.53e-5	1.52e-5	1.53e-5	1.52e-5
1024	1.48e-5	1.55e-5	1.51e-5	1.53e-5	1.58e-5	1.56e-5
2048	1.49e-5	1.51e-5	1.49e-5	1.49e-5	1.50e-5	1.56e-5
4096	1.57e-5	1.53e-5	1.57e-5	1.53e-5	1.51e-5	1.50e-5
8192	1.55e-5	1.51e-5	1.49e-5	1.54e-5	1.47e-5	1.54e-5
16384	1.51e-5	1.55e-5	1.52e-5	1.54e-5	1.55e-5	1.52e-5
32768	1.56e-5	1.59e-5	1.48e-5	1.57e-5	1.53e-5	1.52e-5

VI. When is Differential Checkpointing Beneficial?

In order to estimate the threshold at which differential checkpointing becomes beneficial, we construct a cost function from the reduction in CP overhead:

$$\Delta T_s = |N_d t_w - N_t t_w| = (N_t - N_d)\, t_w, \tag{1}$$

and from the additional generated overhead (i.e. the time to determine the differences):

$$\Delta T_o = (N_t + N_d)\, t_h. \tag{2}$$

t_w is the duration to write a block of data with block-size b, t_h the duration of hashing the block, N_d is the number of blocks that differ and N_t is the total number of blocks. The saving in equation 1 corresponds to the absolute value of the time difference between writing all blocks ($N_t t_w$) and writing only the dirty blocks ($N_d t_w$). The overhead in equation 2 corresponds to the time to hash the data blocks. Equation 2 involves both values, N_t and N_d, since, we cannot commit the new hashes for data-blocks that differ prior to the successful completion of the CP, hence we compute these twice[2]. After normalizing to the total number of blocks N_t we get:

$$\tau = (t_h - t_w) + n_d\,(t_w + t_h), \quad n_d = N_d/N_t. \tag{3}$$

Where $\tau := \Delta T/N_t = (\Delta T_o - \Delta T_s)/N_t$. Equation 3 can be considered a cost function that turns into a reduced overhead (speedup) for $\tau < 0$ and to additional overhead for $\tau > 0$. We can infer, that the maximal overhead accounts to $2N_t t_h$ when $n_d = 1$. This corresponds to a maximal relative overhead of $2t_h/t_w$ (i.e., relative to the time without dCP).

We may define the threshold, η, at $\tau = 0$ as:

$$\eta := n_d\Big|_{b,\tau=0} = \frac{t_w - t_h}{t_w + t_h} \approx \frac{1-\rho}{1+\rho}, \quad \rho = \frac{t_h}{t_w}. \tag{4}$$

Let us keep in mind that τ depends on the block size b as well. η corresponds to the threshold ratio of updated CP data (i.e. dirty) to the total amount of CP data below which we can expect a speedup. Equation 4 is defined for $\eta \in [0, 1]$ and behaves monotonic in that regime. The lower the value for ρ the closer η gets to 1, which would correspond to a threshold of $n_d = 1 \mathrel{\widehat{=}} 100\%$ dirty (i.e. no overhead).

We can give an estimation of η by comparing the time that it takes to write and that it takes to hash a block of data. Thus, we measure the time, t_w, to write a block of size b to disk, where we consider the write to be a collective operation. We do so by measuring the total time, T_w, to write a buffer of size $n * b$ and computing $t_w = T_w/n$. T_w is the time for a collective write (i.e. all processes must have finished I/O). And also we measure the time, t_h, that it takes to compute the hash for a block of size b. In contrast to t_w, t_h is computed by $t_h = \sum t_{h,i}/n$, where $t_{h,i}$ is the time to hash the ith block, thus t_h is the average value of all $t_{h,i}$. This reflects in contrast to T_w a local (non-collective) operation.

Since the hash creation is local to the ranks we may expect a perfect scaling behavior for t_h. For t_w instead, we have to consider network bottlenecks that can slow down the I/O processes towards a larger scale. Thus, we expect an increasing speedup for increasing total problem sizes.

Figure 3 shows the results for the measurements we performed for 768 and 2400 processes. In both cases, the total buffer size was 1GB per process which leads to the total problem sizes of 0.75 TB and 2.3 TB respectively.

Note that η is the threshold w.r.t the dirty blocks. In figure 3 we show $1 - \eta$, which corresponds to the threshold w.r.t the clean blocks. We can see that the threshold indeed decreases for a growing problem size. We observe a better performance of MD5 towards CRC32 in all cases. The performance of MD5 depends slightly on the hash-block size. This dependency is less strong at a larger scale. This also applies for the performance difference between CRC32 and MD5. The results show that for $b =$ 32KB and MD5, the threshold is at merely 5% clean data share (i.e. 5% less to write).

VII. Evaluation

In section VI we have seen that even when applications update 95% of the checkpoint data (i.e. we save only about 5% of I/O) dCP can already be beneficial for HPC applications. In order to demonstrate this theoretical result with empirical evidence, we analyze the behavior of dCP in FTI while checkpointing three HPC applications at large scale. We conduct representative experiments that analyze performance and overhead. All experiments were performed on MareNostrum4. Each node is composed by [24], [25]:

[2]We may avoid the redundancy here if we store the hashes for the *dirty* blocks in a separate array, which would lead to a higher memory footprint.

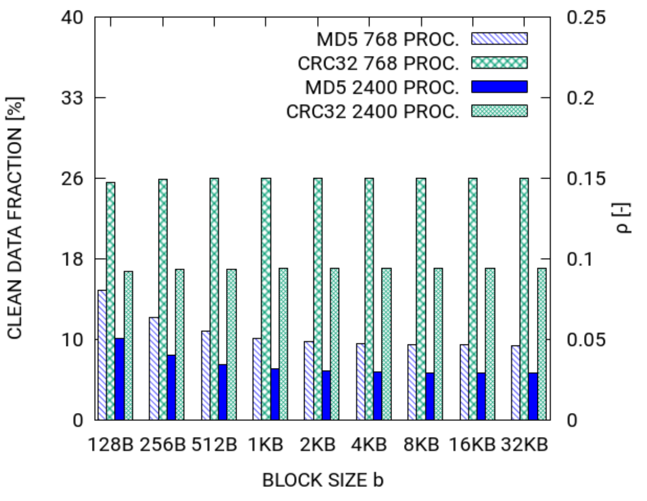

Fig. 3: The bars show the estimated dCP threshold, i.e. the fraction of clean data we need to make the dCP operation beneficial. The left axis shows the clean data fraction (1-η), the right axis shows the value of ρ (ratio between the hash time, t_h, and I/O time, t_w, for block size b) that corresponds to the respective value of $1 - \eta$ on the left axis. The experiment has been performed with 768 and 2400 processes and 1GB per rank.

- 2 Intel Xeon Platinum 8160 CPU (24 cores at 2.10GHz)
- 12 $\times$ 8 GB DDR4-2667 DIMMS (96GB/node)
- 100 Gbit/s Intel Omni-Path HFI Silicon 100 Series PCI-E
- 10Gbit Ethernet
- 200 GB SSD local to the nodes
- SUSE Linux Enterprise Server 12 SP2

A. *HPC Applications*

In this section we introduce the different applications used during our large scale evaluation.

1) LULESH 2.0: Lulesh [26] is part of the Advanced Simulation and Computing (ASC) program from the Lawrence Livermore National Laboratory (LLNL). It simulates a Sedov blast wave propagation within one material in three dimensions [27]. The modeling space is discretized into an unstructured hex mesh. The system state is updated using stencil operations. The purpose of LULESH is to provide a proxy application that possesses the characteristics of an HPC application from this field in order to analyze performance on various platforms and various programming models. That makes it very interesting for us as an example since it represents a broad field of applications.

In order to maximize the checkpoint load, we conducted measurements that determined the highest value that we can pass to LULESH without the risk of a memory overflow on the node. The checkpoint data is serialized, which increases the memory footprint of the application. With a CP size of

430MB per rank, we use about 80GB of the node memory (96GB available) and achieve an aggregate CP size of 725GB.

2) xPic: xPic is an alternative implementation of the physical problem treated in iPic3D [28]. iPic3D and xPic are part of the application co-design in the DEEP-EST project [29]. The application models space plasma simulations. The modeling space is discretized by a rigid mesh. The mesh is defined in the configuration file. The simulation is always initialized to the equilibrium state. In each time step, the particle states and electromagnetic fields are advanced using the Vlasov equation, which couples the equation of motion to the Maxwell equations.

xPic takes its runtime parameters from a configuration file. In order to scale the problem size, we used a combination of the parameters `ntcx` (number of cells in x-direction), `ntcy` (number of cells in y-direction) and `nppc` (number of particles per cell). To control the number of contiguous datasets, we modified the parameters `nblockx` (number of blocks in x-direction), `nblocky` (number of blocks in y-direction) and `nspec` (number of species). We implemented two distinct mechanisms in order to expose datasets to FTI. In the first implementation, xPic-c (c for common), we expose every memory contiguous dataset individually to FTI. Depending on the configuration of xPic, this may lead to a large number of protected variables. In the second implementation, xPic-s (s for serialized), we use BOOSTs `libboost_serialization` library [30] to combine the datasets into one contiguous buffer which is then exposed to FTI.

3) Heat2D: Heat2D is a 2D heat distribution simulation using a 1D domain decomposition. It simulates the transition from a non-equilibrium heat distribution to the equilibrium state. In each time step, the cells of the temperature grid are updated via a 4-point stencil operation that stores the average of the 4 neighbor cells temperatures into the center cell. The ranks exchange adjacent rows of the temperature grids. The simulation runs until the total value of the temperature differences reaches a pre-defined minimal value or exceeds a certain number of iterations. The large majority of memory used by Heat2D is checkpointed which enabled us to perform large scale executions with large checkpoitn sizes, for instance a run with a total problem size of about 2.8TB with 2304 processes on 48 nodes.

B. Variation of the Block Size b

We start by analyzing the impact of the block size over the effectiveness of dCP. We measured the time of a dCP update for various block sizes b and compared the results to an ordinary CP (dCP disabled). All CPs were performed at the same application state. We performed experiments with both MD5 and CRC32, the results were very similar for both hashing algorithms thus we show only the MD5 results for space constrains. By decreasing the block size, we increase the granularity. That means that we have a better chance to get close to the actual percentage of data that did change. This should result in fewer data to write and therefore we expect better performance with smaller blocks.

Table II shows the results for the experiment we performed with the xPic application (see VII-A2 for details). The first column of the table shows the block size and the third column shows the percentage of data written compared to the original checkpoint size. We notice that as the block size increases, the amount of data to write increases as well, due to the lower granularity. However, the overhead (shown in the second column) is incredibly large for high block granularities (i.e., small blocks). To understand this phenomena, we measured the time spent hashing and the time spent writing data for each case. We observe that the large majority of checkpointing time is spent in writing and not hashing. This is caused by the fragmentation of the updates into small chunks. It has been shown in the past (e.g. [31]–[33]), that PFSs have poor performance when small chunk sizes need to be written.

TABLE II: Impact of the block size b on the dCP update time for xPic using MD5. Negative values of τ correspond to a speedup and positive values to overhead. *HASH SIZE* lists the respective memory sizes that the hash tables occupy in memory. The problem size was 1568MB per rank.

b	τ	dCP RATE	SHARE HASH	SHARE WRITE	HASH SIZE [MB]
128B	1333%	52.25%	1.51%	97.67%	196
256B	1106%	53.84%	1.53%	97.39%	98
512B	666%	56.25%	2.10%	96.13%	49
1KB	231%	59.15%	4.40%	91.40%	25
2KB	15%	61.42%	12.82%	73.93%	12
4KB	-32%	62.25%	21.77%	55.07%	6
8KB	-35%	62.41%	22.69%	52.47%	3
16KB	-36%	62.48%	22.66%	52.52%	1.5
32KB	-36%	62.50%	22.67%	52.07%	0.76

For xPic, block sizes of less than 4KB degrade performance and block sizes greater than 4KB improve performance up to 36%. In addition, we measured the amount of memory consumed to store the hash tables. Most of the block sizes have hash tables that represent less than 1% of the memory used by the process. For block sizes of 16KB the hash tables take only 0.1% of the memory used by the application. Based on this analysis, we decided to use block sizes of 16KB during the following measurements.

C. Spatial and Temporal Differences

After finding the right block size to avoid too coarse hashes as well as to fine I/O writes, we investigate the amount of data that is actually being updated between two consecutive checkpoints for the applications presented in Section VII-A.

The results are depicted in Figure 4. The three sub-figures are divided into several temporal regions following the y axis (i.e., dCP taken at iteration 1000, 5000, etc.) and spatial regions following the x axis (i.e., the process rank which is representative of a slice of the domain). First, we observe that LULESH does not change too much data during the first iterations; and as the time passes (up to iteration 20000) the number of ranks where data is actually modified increases. This reflects the shock wave that is simulated by LULESH. This demonstrates that for applications like LULESH, the benefits of dCP might vary depending on time and space.

xPic on the other hand, shows a completely different behavior, the amount of data updated is consistently the same across all the ranks and regardless of the time in the execution. This is explained by the fact that xPic is a plasma simulation in which particles are constantly in movement, even in those changes are minimal, they are enough to trigger updates as they will produce a different block hash. There are a few variables of the application that are read-only and that do not

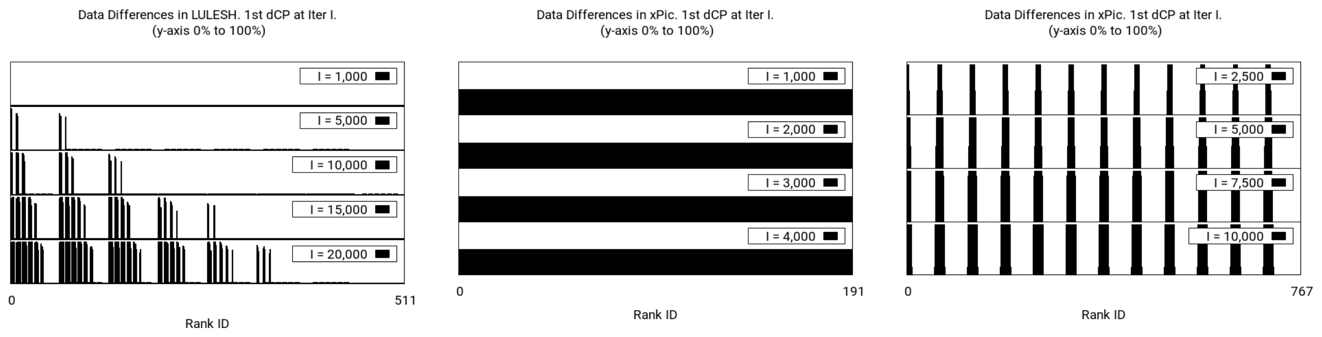

Fig. 4: Data differences after first checkpoint for different ranks (x axis) and different times for checkpointing (y axis).

change through out the simulation, which is why not a 100% of the data is updated at every checkpoint.

Looking into Heat2D, we observe a middle ground between LULESH and xPic. Indeed, Heat2D also increases the data differences as time evolves, but a much lower pace than LULESH, giving it a less dynamic look. We observe that the most affected ranks are organized in strides, which is consistent with the 1D partitioning mentioned in Section VII-A3. However, other initial conditions could translate into a more homogeneous updates across ranks.

D. Overhead reduction on HPC Applications

In this section we evaluate the overhead of dCP in comparison with classic CP for the three applications.

Table III lists the results of our measurements performed with LULESH. The first row represents the full CP and the second a checkpoint with dCP in which only 3% of the data is updated. We have only two rows since we never had updates significantly different to 3%. This result indicates that the propagation of the wave is slow as shown previosly. This great reduction in checkpoint size with dCP in LULESH translates into a 62% reduction in the CP time.

TABLE III: Relative overhead of dCP compared with full CP for LULESH. Negative values correspond to a reduction of the overhead (speedup) and positive values to an increase in the overhead.

Relative checkpoint overhead compared to full CP ($\Delta T/T_0$ [%])			
Data diff. (n_d)	MD5	CRC32	NO dCP
100%	-9±12	-5 ± 13	-5 ± 13
3%	-62 ± 10	-60 ± 8	-

For xPic, we evaluate the non-serialized as well as the serialized implementations (xPic-c and xPic-s, see VII-A2) were each implementation is tested against two distinct configurations (A and B). For configuration A, the FTI protected memory consist of many relatively small contiguous datasets. Configuration B instead has few but rather large contiguous datasets. Table IV summarizes the relevant runtime parameters for both configurations.

The results of our evaluation with these configurations is shown in Table V. First, we observe that the reduction on checkpoint size is the same for executions with and without serialization. Another observation is that the application of dCP for configuration A does not reduce the checkpoint overhead. The reason for this is that configuration A produces a large number of small chunks to be written. A more detailed analysis of this phenomena is done in section VIII. In contrast, we do observe an important overhead reduction for configuration B. The best performance measured is for xPic-s (serialized) using MD5 with up to 35% speedup while writing only 50% of the original checkpoint size.

TABLE IV: Dataset sizes for the various xPic configurations.

	CONFIG. A		CONFIG. B	
	xPic-c	xPic-s	xPic-c	xPic-s
SIZE OF DATASETS [MB]	4.22	1360	168	1344.25
# OF DATASETS	320	1	8	1
CP SIZE / RANK [MB]	1350.32	1360.38	1344.55	1344.80
CP SIZE TOTAL [GB]	760	765	882	883

TABLE V: Relative overhead of dCP compared with full CP for xPic. Negative values correspond to a reduction of the overhead (speedup) and positive values to an increase in the overhead.

Relative checkpoint overhead compared to full CP ($\Delta T/T_0$ [%])				
	Data diff. (n_d)	MD5	CRC32	NO dCP
xPic-c (A)	100%	7 ± 11	6 ± 12	0 ± 9
	50%	9 ± 12	11 ± 9	-
xPic-c (B)	100%	9 ± 16	14 ± 11	-3 ± 9
	62%	-33 ± 6	-28 ± 6	-
xPic-s (A)	100%	7 ± 17	14 ± 9	0 ± 7
	50%	-4 ± 6	0 ± 6	-
xPic-s (B)	100%	5 ± 5	11 ± 7	-2 ± 6
	62%	-35 ± 7	-29 ± 6	-

As mentioned in Section VII-C, the data difference in Heat2D depend significantly on the initial conditions. Heat2D shows a good reduction of checkpoint size, in the regime of 40% to 100%. Table VI lists the results. We can see that MD5 has clearly performance benefits in comparison to CRC32. We notice that almost all of the experiments show an significant reduction on the checkpoint overhead. We observe important speedups of up to 49% for a 40% dCP update using MD5.

Overall, the three applications (although with different behaviours) show substantial improvements thanks to dCP. The reduction in checkpointing overhead goes up to 62%, 35% and 49% for LULESH, xPic and Heat2D respectively.

TABLE VI: Relative overhead of dCP compared with full CP for Heat2D. Negative values correspond to a reduction of the overhead (speedup) and positive values to an increase in the overhead.

Relative checkpoint overhead compared to full CP ($\Delta T/T_0$ [%])			
Data diff. (n_d)	MD5	CRC32	NO dCP
100%	-2 ± 9	1 ± 6	-4 ± 11
99%	-5 ± 7	-2 ± 7	-
95%	-8 ± 6	-7 ± 7	-
87%	-14 ± 6	-12 ± 6	-
79%	-19 ± 8	-17 ± 6	-
71%	-26 ± 6	-22 ± 6	-
63%	-35 ± 5	-30 ± 5	-
56%	-40 ± 5	-37 ± 4	-
40%	-49 ± 5	-46 ± 7	-

VIII. Discussion

In section VI we presented a theoretical model that may be used to estimate the speedup we may achieve using dCP. In this section, we want to check whether the predictions from the model coincide with the measurements or not.

Let us write down the relative time difference, S, of a dCP update towards a conventional CP:

$$S(n_d) = \Delta T(n_d)/T_0 := \begin{cases} < 0 : \text{overhead reduction} \\ > 0 : \text{overhead increase} \end{cases} .$$

T_0 denotes the time for a full CP and dCP disabled. Using equation 3 we may write this as:

$$S(n_d) = \frac{\tau}{t_w} = \rho - 1 + n_d(\rho + 1) \quad , \quad \rho = \frac{t_h}{t_w} \tag{5}$$

We used here that $T_0 = t_w N_t$. We determined t_w and t_h by a separate measurement and the values we use here are:

$$b = 16\text{KB} \tag{6}$$
$$t_w = 1.35 \times 10^{-3} s \tag{7}$$
$$t_h = 3.92 \times 10^{-5} s \quad \text{[MD5]} \tag{8}$$
$$\rightarrow \rho = 0.029 \tag{9}$$

For clarity, we will consider only the results for MD5.

Figure 5 shows the measured relative speedups and the estimation computed by equation 5. The figure shows that in two cases the estimation is near to accurate and in two cases it is not as accurate. Heat2D and xPic with configuration B show both a very good matching to the estimation done with the theoretical model. LULESH shows the highest speedup with 62%, however, using equation 5 we would expect a speedup of about 94%. For xPic-s with configuration A, we measured a 4% speedup but expected about 46%.

Given the disagreement between theoretical prediction and experimental results for LULESH and xPic with configuration A, we performed a more detailed analysis. Figure 6 shows the cumulative density function (CDF) of chunk sizes written contiguously during a dCP update for all four scenarios. The figure reveals a correlation between the size of the chunks and the performance. xPic-s A and LULESH show both less performance than expected and both write mostly chunks of relatively small sizes (4MB - 12MB). On the other hand, we have good performance in xPic B and Heat2D where we observe relatively large chunk sizes (mostly over 200MB).

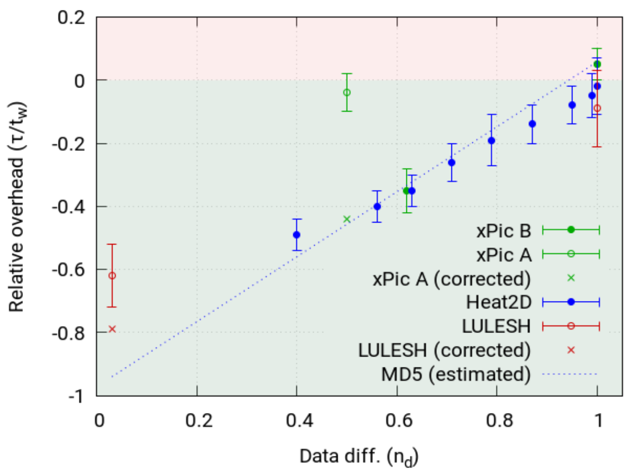

Fig. 5: Measured and estimated speedup/overhead of dCP updates. The green background indicates the region where we have speedup and the red region indicate overhead. $\tau/t_w = 0$ corresponds to the threshold (i.e., the full CP baseline) The datasets with the label *corrected*, refer to measurements that used a buffer to collect small chunks in order to avoid small chunck writes.

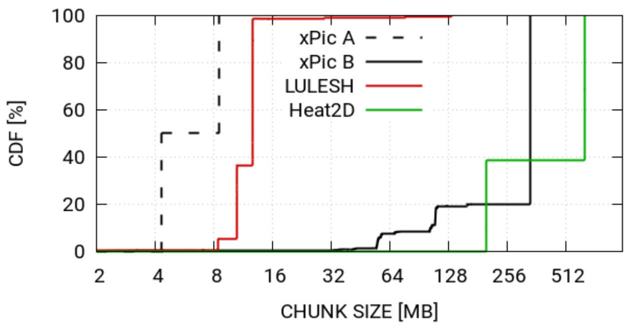

Fig. 6: Cumulative distribution function (CDF) for chunk sizes of contiguous dirty regions during dCP updates.

If the small writes are the explanation for the inaccurate model predictions, one should be able to meet the estimated performance by avoiding I/O operations with small chunks. This can be accomplished by allocating a buffer of sufficient size and collecting small chunks in this buffer until the accumulated chunk size exceeds an appropriate I/O size (e.g., 16MB), to then finally write to a dedicated file. We implement this technique and redo the experiments.

The results for LULESH and xPic after implementing this modification are denoted as *corrected* in figure 6. We can see that after correction xPic A is in very good agreement with the model prediction. LULESH also improved but it is still not as good as the model has predicted. We continue the detailed analysis of LULESH and we noticed that LULESH has about 2-3% updates in all ranks *except* in rank 0. Rank 0 has a dCP share of 80%. A large amount of the data is thus written by only one rank. It appears, that the model performs less well if the distribution of the dCP share is highly anti-symmetric. For all the other cases, we have a very good matching between model prediction and experimental results, if we avoid small chunk sizes in I/O operations.

IX. Conclusion

In this paper we experimented with the UNIX page protection mechanism in order to determine data differences, but this mechanism is not able to differ between assignments that leave the data invariant and assignments that indeed change the data, since every access to a memory address always causes this address to be considered as *dirty* by the operating system. Thus, we implemented a hash-based differential checkpointing mechanism capable to detect real data changes. We tested 4 hash algorithms upon performance and reliability and our conclusion is that CRC32 and MD5 are safe choices to implement dCP.

Another challenge of implementing differential checkpointing is that some applications have datasets that change in size during the execution. To overcome this issue, we developed a new file format (FTI-FF) for FTI that includes its own metadata and is extensible. Our results indicate that the dCP performance, for our prototype implementation, is significantly better in most of the cases. For some applications, it might depend on the chunk size of contiguous dirty blocks that are written to the CP files during the dCP update. Indeed, we observed a better performance towards larger chunk sizes, and dCP becomes clearly inefficient for very small chunk sizes. However, we have demonstrated that this issue can be resolved by a mechanism that collects small chunks into a large block until an appropriate accumulated size is reached before writing to stable storage. We observed a speedup of up to 49% in Heat2D, 35% in xPic and 62% in LULESH.

X. Acknowledgements

This project has received funding from the European Unions Seventh Framework Programme (FP7/2007-2013) and the Horizon 2020 (H2020) funding framework under grant agreement no. H2020-FETHPC-754304 (DEEP-EST); and from the European Unions Horizon 2020 research and innovation programme under the LEGaTO Project (legato- project.eu), grant agreement No 780681. The present publication reflects only the authors' views. The European Commission is not liable for any use that might be made of the information contained therein.

References

[1] "Managing large state in apache flink: An intro to incremental checkpointing," https://flink.apache.org/features/2018/01/30/incremental-checkpointing.html, accessed: 2018-04-23.

[2] N. Abu-Ghazaleh and M. J. Lewis, "Differential checkpointing for reducing memory requirements in optimized soap deserialization," in *The 6th IEEE/ACM International Workshop on Grid Computing, 2005.*, Nov 2005, pp. 6 pp.–.

[3] ——, "Lightweight checkpointing for faster soap deserialization," in *2006 IEEE International Conference on Web Services (ICWS'06)*, Sept 2006, pp. 11–18.

[4] N. Abu-Ghazaleh, M. J. Lewis, and M. Govindaraju, "Differential serialization for optimized soap performance," in *Proceedings. 13th IEEE International Symposium on High performance Distributed Computing, 2004.*, June 2004, pp. 55–64.

[5] K. Chiu, M. Govindaraju, and R. Bramley, "Investigating the limits of soap performance for scientific computing," in *Proceedings 11th IEEE International Symposium on High Performance Distributed Computing*, 2002, pp. 246–254.

[6] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Hybrid checkpointing for mpi jobs in hpc environments," in *2010 IEEE 16th International Conference on Parallel and Distributed Systems*, Dec 2010, pp. 524–533.

[7] R. Gioiosa, J. C. Sancho, S. Jiang, and F. Petrini, "Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, Nov 2005, pp. 9–9.

[8] G. Bronevetsky, D. Marques, K. Pingali, S. McKee, and R. Rugina, "Compiler-enhanced incremental checkpointing for openmp applications," in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–12.

[9] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "**Libckpt**: Transparent checkpointing under Unix," in *Usenix Winter Technical Conference*, January 1995, pp. 213–223.

[10] K. B. Ferreira, R. Riesen, R. Brighwell, P. Bridges, and D. Arnold, "libhashckpt: Hash-based incremental checkpointing using gpu's," in *Recent Advances in the Message Passing Interface*, Y. Cotronis, A. Danalis, D. S. Nikolopoulos, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 272–281.

[11] K. B. Ferreira, "Keeping checkpointing viable for exascale systems," Ph.D. dissertation, The University of New Mexico, 2011.

[12] "Libckpt home page," http://web.eecs.utk.edu/~plank/plank/www/libckpt.html, accessed: 2018-04-22.

[13] 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*.

[14] N. Devillard, "Iniparser 4," https://github.com/ndevilla/iniparser, 2017.

[15] L. Bautista-Gomez, "Fti - fault tolerance interface," https://github.com/leobago/fti, 2018.

[16] "Fti file format documentation," http://leobago.github.io/fti/ftiff.html.

[17] M. Harran, W. Farrelly, and K. Curran, "A method for verifying integrity and authenticating digital media," *Applied Computing and Informatics*, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2210832717300753

[18] M. Adler and J.-l. Gailly, "Zlib data compression library," https://github.com/madler/zlib, 1995-2017.

[19] A. Nakassis, "Fletcher's error detection algorithm: How to implement it efficiently and how toavoid the most common pitfalls," *SIGCOMM Comput. Commun. Rev.*, vol. 18, no. 5, pp. 63–88, Oct. 1988. [Online]. Available: http://doi.acm.org/10.1145/53644.53648

[20] "OpenSSL 1.0.2 manpages - md5," https://www.openssl.org/docs/man1.0.2/crypto/md5.html, accessed: 2018-04-17.

[21] S. Ramanujam and M. Karuppiah, "Designing an algorithm with high avalanche effect," *IJCSNS International Journal of Computer Science and Network Security*, vol. 11, no. 1, pp. 106–111, 2011.

[22] S. Gueron, "Speeding up crc32c computations with intel crc32 instruction," *Information Processing Letters*, vol. 112, no. 5, pp. 179 – 185, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S002001901100319X

[23] "zlib home page," https://zlib.net, accessed: 2018-05-3.

[24] "bsc.es, marenostrum4 user's guide," https://www.bsc.es/user-support/mn4.php#systemoverview, accessed: 2018-04-27.

[25] "bsc.es, marenostrum iv (2017) system architecture," https://www.bsc.es/marenostrum/marenostrum/technical-information, accessed: 2018-04-27.

[26] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Tech. Rep. LLNL-TR-641973, August 2013.

[27] "Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory," Tech. Rep. LLNL-TR-490254.

[28] S. Markidis, G. Lapenta, and Rizwan-uddin, "Multi-scale simulations of plasma with ipic3d," *Mathematics and Computers in Simulation*, vol. 80, no. 7, pp. 1509 – 1519, 2010, multiscale modeling of moving interfaces in materials. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0378475409002444

[29] "Deep projects," http://www.deep-projects.eu/.

[30] "Boost - serialization," https://www.boost.org/doc/libs/1_67_0/libs/serialization/doc/index.html, accessed: 2018-05-17.

[31] W. Yu, J. S. Vetter, and H. S. Oral, "Performance characterization and optimization of parallel i/o on the cray xt," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, April 2008, pp. 1–11.

[32] H. Shan and J. Shalf, "Using ior to analyze the i / o performance for hpc platforms," 2007.

[33] L. B. Gomez, K. Keller, and O. Unsal, "Performance study of non-volatile memories on a high-end supercomputer," in *Workshop on Performance and Scalability of Storage Systems 2018 (WOPSSS'18), Frankfurt, Germany*, June 2018.