Improving Scalability of Application-Level Checkpoint-Recovery by Reducing Checkpoint Sizes

Iván Cores, Gabriel Rodríguez, María J. Martín, Patricia González and Roberto R. Osorio

Computer Architecture Group
University of A Coruña, Spain

ivan.coresg@udc.es

Abstract The execution times of large-scale parallel applications on nowadays multi/many-core systems are usually longer than the mean time between failures. Therefore, parallel applications must tolerate hardware failures to ensure that not all computation done is lost on machine failures. Checkpointing and rollback recovery is one of the most popular techniques to implement fault-tolerant applications. However, checkpointing parallel applications is expensive in terms of computing time, network utilization and storage resources. Thus, current checkpoint-recovery techniques should minimize these costs in order to be useful for large scale systems. In this paper three different and complementary techniques to reduce the size of the checkpoints generated by application-level checkpointing are proposed and implemented. Detailed experimental results obtained on a multicore cluster show the effectiveness of the proposed methods to reduce checkpointing cost.

§1 Introduction

High-performance computing (HPC) systems tend to increase their number of processors from year to year. Capello et al. [3] and Schroeder et al. [37] state that failure rates depend mostly on system size and are roughly proportional to the number of processors in a system. Thus, fault-tolerance techniques need to be applied to parallel applications running in HPC environments to guarantee computation progress.

Many methods for achieving fault tolerance in parallel applications exist in the literature, checkpoint-recovery [7] being the most popular. It periodically saves the computation state to stable storage, so that the application execution can be resumed by restoring such state. The overhead of saving checkpoints to disk is the main performance cost in checkpoint-recovery methods. This cost could become prohibitive for parallel applications running on large-scale facilities [3, 38], where the I/O bandwidths do not increase as quickly as their computational capability [17, 29] and the checkpoint frequency must be increased to manage the higher failure rate.

There are two fundamental approaches to checkpointing: system-level checkpointing (SLC), implemented at the operating system level, and application-level checkpointing (ALC), where the application program saves and restores its own state. In SLC the whole state of the processes (program counter, registers and memory) is saved to stable storage. The most important advantage of this approach is its transparency. However, it has two important drawbacks. First, storing the whole application state will have a higher associated cost than storing just necessary data. Second, it is inherently non-portable. We say a checkpointing technique is portable if it allows the use of state files to recover the state of a failed process on a different machine, potentially binary incompatible or using different operating systems or libraries. The basic condition that has to be fulfilled in order to achieve potential portability is not to store any low-level data along with the process state. Therefore, all SLC approaches are not portable. Application-level checkpointing, on the other hand, is able to obtain better performance by storing only necessary data. Additionally, it enables both data portability, by storing data using portable representation formats, and communication-layer independence, by implementing the solution at a higher level of abstraction. The drawback is the need for analyses of the application code in order to identify the state that needs to be stored.

This work proposes and evaluates different techniques to reduce checkpoint file sizes and, thus, the computational and I/O cost of checkpointing in ALC approaches. It's main contributions are:

- A hash-based implementation of incremental checkpointing for ALC approaches, which includes the elimination of regions of memory that contain only zeros with no additional computational cost.
- A simple compression algorithm specially designed for checkpoint files.
- An experimental comparison of SLC and ALC when using different check-

pointing optimization techniques.

We believe that ALC implementations will be needed to guarantee scalable solutions on large-scale architectures. Additionally, the advent of the Cloud makes potential portability of ALC methods a valuable feature. The rest of the paper is organized as follows. Section 2 describes related work. Section 3 proposes three different and complementary techniques to optimize the checkpoint sizes in ALC solutions: live variable analysis to avoid storing dead variables; incremental checkpointing and zero-blocks exclusion to store only modified data and to avoid storing null elements; and data compression to remove redundant information. Section 4 explains the implementation details of those techniques on an ALC tool. Section 5 evaluates the performance of the proposed methods. Finally, Section 6 concludes the paper.

§2 Related Work

Although checkpoint/restart is the most common solution to endow scientific applications with fault tolerance, its cost in terms of computing time, network utilization or storage resources can be a limitation for large scale systems. There exist in the literature a number of techniques to optimize the cost of checkpointing.

Checkpoint file size is the most important factor in determining checkpointing performance. As such, the reduction of the amount of stored state is one of the usual goals of checkpoint optimizations. However, most of the techniques described in the bibliography are applied to SLC approaches, since ALC solutions are less general, and they already achieve smaller checkpoint files. Nevertheless, in order to be useful for today large scale systems, ALC approaches will also need to minimize checkpoint file sizes. In this paper, different strategies to reduce checkpoint file sizes in ALC are proposed.

Incremental checkpointing is one of the most popular techniques to reduce checkpoint file sizes. This approach generates two types of checkpoints: full and incremental. Full checkpoints contain all the data that are to be stored. Incremental checkpoints store only the data that have changed since the previous one. The restart process starts from the most recent full checkpoint, and then orderly applies the changes reflected in the subsequent incremental ones. A number of approaches can be found in the bibliography to implement incremental checkpoint in SLC. One of them is to use the virtual memory page protection mechanism to track changed memory pages [34]. Another option is to use a

kernel-level memory management module that employs a page table dirty bit scheme [12]. Both solutions require memory protection support from the underlying hardware along with support from the OS to be able to handle page-fault exceptions. This feature, although very common, is not universally available. An alternative to page-based checkpoint is hash-based checkpoint [1], which uses a secure hash function to obtain a unique identifier for each variable-sized block of application memory to be written into state files. This value is stored and used to detect changes in memory blocks. In this paper this technique is adapted for ALC solutions. Using an application-level approach the number of memory blocks to be checked at runtime is reduced, which minimizes the size of the hash tables to be calculated and stored, improving the overhead. Additionally, in our approach the size of the generated checkpoint files is further reduced through the elimination of those memory blocks that contain only zeros.

Another means to reduce checkpoint file sizes is data compression. This technique has been implemented, for instance, in the ickp checkpointer [32], ErrMgr [15] and the CATCH compiler [20]. In ickp, a predictive algorithm is presented that offers very low overhead, but only performs well with some highly compressible sources, as it often produces data expansion. ErrMgr uses DEFLATE (gzip) [16] and shows results mainly for highly-compressible data. For less compressible data, the overhead offsets any compression benefit. In CATCH, the general purpose LZW [16] algorithm is used, which typically offers slightly worse performance than DEFLATE with similar overhead. CATCH also uses a heuristic algorithm to determine the optimal places, in terms of checkpoint size, to insert checkpoints. Based on particular features observed in checkpoint files, a new and faster compression algorithm is also proposed in this paper. This new algorithm addresses the trade-off between compression efficiency and overhead.

Memory exclusion is another powerful approach for reducing the size of checkpoint files in SLC approaches. Plank et al. [30] proposed a compiler-assisted solution to automate the memory exclusion process. The user is responsible for inserting `EXCLUDE_HERE` directives which are translated by the compiler into `include_bytes()` and `exclude_bytes()` function calls after analyzing the data flow of the program. Directive placement is critical to checkpoint size savings and performance. In this paper we propose a method based in live variable analysis to select only relevant variables in ALC. The inclusion approach avoids the complex memory patterns that appear in exclusion-based approaches, which may improve runtime performance. This solution does not require manual placement

of a directive to mark analysis points, since these are optimally detected by an interprocedural analysis.

All the techniques mentioned so far focus on reducing checkpoint file sizes. Another way to optimize the computational and I/O cost of checkpointing is to avoid the storage of checkpoint files in a parallel file system. Plank et al. proposed to replace stable storage with memory and processor redundancy [33]. Recent works [4, 5, 13, 43] have adapted the technique, known as *diskless checkpointing*, to contemporary architectures. The main drawback of diskless checkpointing are its large memory requirements. As such, this scheme is only adequate for applications with a relatively small memory footprint at checkpoint. Other recent solutions focus on the use of non-volatile memory technology, like solid-state disks (SSDs) to keep checkpoint data [21]. SSDs offer excellent read/write throughput when compared to secondary storage and thus they can help to reduce disk I/O load. Moody et al. propose a multi-level checkpoint system that writes checkpoints to RAM, Flash, or disk on the compute nodes in addition to the parallel file system [23].

Other works focus on minimizing the network and file system contention caused by the parallel checkpointing by reducing the number of simultaneous checkpoints. Norman et al. identify at compile-time recovery lines formed by staggered checkpoint calls so that the concurrent writing of checkpoint files is minimized at run-time [28]. In [18] the data layout of the checkpoint files are rearranged to reduce the number of files serviced by each I/O server. Additionally, the write operations of concurrent checkpoints are serialized on each computer node to further improve the checkpointing performance.

Accelerators have been also considered for reducing checkpointing overhead [9, 11]. Mainly, these works focus on computing hash functions using GPUs. Whereas significant speed-ups are obtained, hash calculation is not a bottleneck in the checkpointing process. Data compression is an interesting target for hardware accelerators that we intend to explore as future work. Up to the present moment, we have not found any implementation in the literature.

As shown above, there are multiple and non-exclusive alternatives to reduce the overhead associated to checkpoint-recovery. We think that parallel jobs will need to combine several of these techniques in order to scale in future HPC platforms.

Finally, some researches are currently evaluating the applicability of other fault-tolerance mechanisms such as process replication [10, 6], proactive mi-

gration [42] or algorithmic-based fault-tolerance [2].

§3 Checkpoint Size Optimization on Application-Level Checkpointing

The basic difference between SLC and ALC, in terms of state file size optimizations, surges from the fact that SLC sees the application memory as a single continuum, while ALC distinguishes a disperse set of contiguous memory blocks. Each block contains memory allocated to one or more variables, depending on the aliasing relationships of the application data. The following sections deal with the utilization of different checkpoint size optimization solutions into an application-level approach.

3.1 Live variable analysis

The knowledge of application code and memory in ALC can be used to select those variables that are live during the creation of state files, avoiding storage of dead variables. Depending on the considered application, applying this technique can significantly reduce checkpoint file sizes.

The identification of these variables can be performed at compile time through a standard live variable analysis. A variable x is said to be *live* at a given statement s in a program if there is a control flow path from s to a use of x that contains no definition of x prior to its use. The set LV_{in} of live variables at a statement s can be calculated using the following expression:

$$LV_{in}(s) = (LV_{out}(s) - DEF(s)) \cup USE(s) \quad (1)$$

where $LV_{out}(s)$ is the set of live variables after executing statement s, and $USE(s)$ and $DEF(s)$ are the sets of variables used and defined by s, respectively. The live variable analysis should take into account interprocedural data flow.

Checkpoints in application-level approaches are usually triggered by an explicit call to a checkpoint function in the application code. This guarantees that checkpoints are not performed during a library or system call, which may have internal state unknown to the checkpointer, but rather inside user-level code. In this way, checkpoint callsites are limited and known at compile time, which allows for the live variable analysis to be bounded and not span the whole application code. For each checkpoint callsite c_i, it is only necessary to store the set of variables which are live when the control flow enters the callsite, $LV_{in}(c_i)$.

3.2 Incremental checkpointing and zero-blocks exclusion

The most popular technique for checkpoint file size reduction in SLC approaches is incremental checkpointing. This technique involves creating two different types of checkpoints: full and incremental. Full checkpoints contain all the application data. Incremental checkpoints only contain data that has changed since the last checkpoint. Usually, a fixed number of incremental checkpoints is created in between two full ones. During a restart, the state is restored by using the most recent full checkpoint file, and applying, in an ordered manner, all the differences before resuming the execution.

As mentioned in Section 2, there exist in the literature different solutions to implement incremental checkpointing in SLC approaches. They can be mainly classified into hash-based [1, 25] or page-based [12, 34, 8, 31, 41]. In ALC it is not recommendable to track changes to memory blocks using a page-based method, as array variables do not necessarily start at page boundaries. Evaluating memory changes for each array as a whole is also inadvisable, following the locality principle. The best compromise is to divide array variables into chunks of memory of a previously specified size and control changes into these chunks using a secure hash function. The calculated hash value for each chunk is stored in memory and used for comparison when creating incremental checkpoints.

When working with real scientific applications it is well known that quite often many elements of the arrays are null, resulting in memory blocks that contain only zeros. Therefore, a possible optimization to further reduce the checkpoint file size is to avoid storage of those *zero-blocks*. In addition to control the changes into memory blocks, the hash function may also be used to detect zero-blocks. When a zero-block is detected, a small marker is saved into the checkpoint file to indicate that the block is null, instead of dumping its contents. During restart this marker is identified and the target memory is filled with zeros, which recovers the original state at a negligible cost in terms of both performance and disk usage.

The idea of not storing zero-blocks has a certain similarity to the technique used in the SLC tool Berkeley Lab's Checkpoint/Restart (BLCR) Library [19] to exclude *zero pages*, that is, those that have never been touched and logically contain all zeros.

3.3 Data compression

Checkpoint files may contain redundant information that can be removed

by means of data compression. Efficient compression algorithms such a LZMA and DEFLATE [16] can find hidden patterns in data and thus reduce the total size of the files. However, highly-efficient compression requires large computational resources. Whereas DEFLATE (zlib), uses less than 1MB of RAM, the more advanced LZMA (7-zip) would require up to 4 GB and be 1 or 2 orders of magnitude slower.

Hence, a fast compression algorithm is proposed in this work that addresses the trade-off between compression efficiency and overhead. We use the well-known technique of substituting repeated chains of bytes by special codes that mark the position of the chain and its length.

A string of bytes is processed sequentially. For each incoming byte, a match within the last 16 processed bytes is sought. The aim is finding the longest possible chain of matches, avoiding encoding each byte individually. Those bytes for which a match cannot be found, are known as literals. The compressed stream consists of a description of the literals, and the position and size of the matched chains. Additionally, entropy coding is applied, using shorter codes for the most common descriptors.

Figure 1 shows an example of the encoding process, where alphabet letters are used instead of numeric 8-bit values. A 16-byte buffer keeps the last processed bytes. A new incoming byte is compared with the content of the buffer, producing a 16-bit mask. The buffer is then updated by shifting-in the new byte. In Figure 1 we can see that 2 literals (*'k'* and *'l'*) are found first. Next, there are 3 candidate positions that match *'a'* and *'b'*. The length of the match keeps growing, but only 1 candidate remains. A logic AND between the current mask and the previous one is a simple way of detecting the end of the matching string. In the example, a new match starts, but it could also be a literal. Note that the length of the matches is not limited to the size of the buffer.

The number of literals (2) and their values are encoded, together with the size of the matching string (5) and its position (12 bytes from the starting point). The way in which those values are encoded was guided by the analysis of many gigabytes of data.

Essentially, an 8-bit token is built by combining the number of literals (up to 15 in a row) and the size of the match (from 1 to 16), as these values show strong correlation. Escape codes are used for longer chains of literals or matches. The tokens are then compressed using static Huffman codes [14]. The literals are not compressed, as they exhibit high entropy. And, finally, the positions are

16-byte buffer	new	comparison mask	match evolution
efab cabc defg abhj	← **k**	0000 0000 0000 0000	no match , 1 literal
fabc abcd efga bhjk	← **l**	0000 0000 0000 0000	no match , 2 literals
abca bcde fgab hjkl	← **a**	**1**00**1** 0000 00**1**0 0000	new match, 2 literals
bcab cdef gabh jkla	← **b**	1001 0000 0010 0000	2 matches, 2 literals
cabc defg abhj klab	← **c**	1001 0000 00**0**0 0000	3 matches, 2 literals
abcd efga bhjk labc	← **d**	**0**001 0000 0000 0000	4 matches, 2 literals
bcde fgab hjkl abcd	← **e**	0001 0000 0000 0000	5 matches, 2 literals
cdef gabh jkla bcde	← **h**	000**0** 000**1** 0000 0000	new match, 0 literals
defg abhj klab cdeh	← **j**	0000 0001 0000 0000	2 matches, 0 literals

Fig. 1 Example of pattern matching for data compression

compressed using a semi-adaptive scheme.

In the example in Figure 1, the inputs from *'k'* to *'e'* would be encoded as: *(2,5) + k + l + 12*. The resulting bit pattern could be: *111111110100000 KKKKKKKK LLLLLLLL 111100*. Hence, 7 bytes would be encoded using 37 bits instead of 56, a 34% gain.

Compared to general purpose algorithms, this proposal allows fast parallel search instead of using iterative search guided by hash keys. Focusing on just the nearest 16 values performs well as matches separated by large distances are not as common in checkpoints as they are in text files. In general, the most common patterns are: runs of values, and repeated exponents in floating point arrays.

Also, we use static Huffman codes combined with simple adaptability. Static means that the codes are the same for all the files, assuming that they all have the same statistical distribution of positions and lengths. We have found that this is a reasonable assumption for checkpoint files, contrarily to the general case. Using fixed, static codes is significantly faster than using dynamic ones and enables building optimized decoders.

§4 Implementation

The three techniques described in Section 3 have been implemented on CPPC [36], an open-source checkpointing tool available under GPL license from `http://cppc.des.udc.es`.

4.1 CPPC overview

CPPC is an application-level checkpointing tool focused on the insertion of fault tolerance into long-running message-passing applications. It is designed

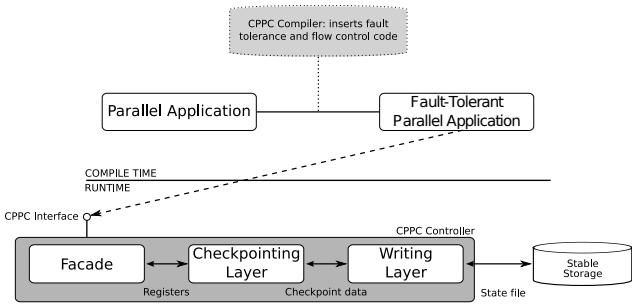


Fig. 2 Integration of a parallel application with the CPPC framework

with a special focus on portability: it uses portable code and protocols, and generates portable checkpoint files, allowing for execution restart on different architectures and/or operating systems.

CPPC appears to the user as a compiler tool and a runtime library. The integration between the application and the CPPC framework is automatically performed by the CPPC compiler, a source-to-source tool that converts an application code into an equivalent version with added checkpointing capabilities. The global process is depicted in Figure 2. At compile time, the CPPC compiler instruments the code by inserting calls to the CPPC library. At runtime, the application will send petitions to the CPPC controller. From the structural point of view, the controller consists of three basic layers: a facade, that keeps track of the state to be stored when the next checkpoint is reached; the checkpointing layer, which gathers, manages and puts together all data to be stored into the state files; and a writing layer which decouples the other two layers from the specific file format used for state storage. Currently CPPC writes checkpoint files using the 5th version of the Hierarchical Data Format (HDF5) [39], a data format and associated library for the portable transfer of graphical and numerical data between computers.

4.2 Live variable analysis

The live variable analysis explained in Section 3.1 is one of the code transformations performed by the CPPC compiler.

When dealing with calls to precompiled procedures located in external libraries, the default behavior is to assume all parameters to be of input type. As such, all function parameters will be included in the set $LV_{in}(s_p)$, being s_p the analyzed procedure call.

Since the proposed procedure is entirely performed at compile time, it adds no overhead during checkpoint operation. In this way, this technique can always improves the efficiency of checkpointing, regardless of the actual reduction obtained in checkpoint file sizes.

4.3 Incremental checkpointing and zero-blocks exclusion

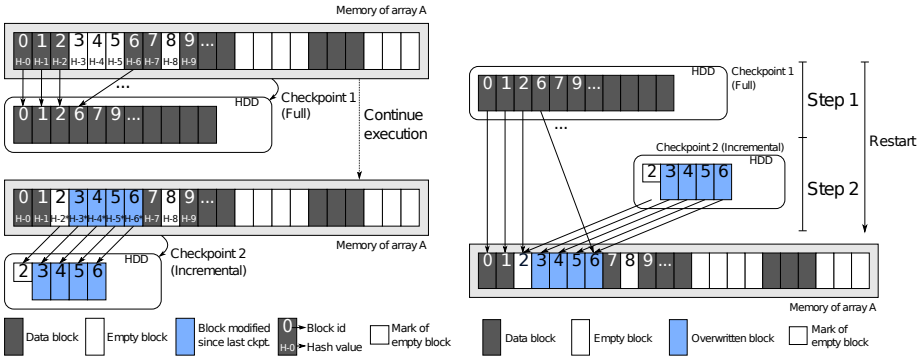
Hash functions are used to detect both changes in memory blocks from previous checkpoints and zero-blocks that can be excluded in the next checkpoint.

For the implementation, CPPC divides array variables into blocks of memory. The size of these memory blocks may have a great impact on the performance of both techniques. CPPC allows the user to choose the size to be used for each particular application. A block size of 8K elements is selected by default when the user does not specify any size. It experimentally proved to be a good compromise value.

CPPC also calculates the hash value of each memory block. The choice of the hash function impacts the correctness, since many hash functions present a significant probability of *collisions*, that is, situations where two different memory blocks are assigned the same hash value. In order to achieve reliable operation, secure hash functions should be used [26]. The implementation in CPPC allows the user to choose between different secure hash functions, such as MD5 or SHA. The MD5 function is selected by default.

In order to detect zero-blocks the calculated hash values are compared to the known hash value of a zero-block. To detect changes in the memory blocks, the hash values calculated in previous checkpoints have to be stored to be compared with the new ones. In our implementation, the hash codes are stored into main memory rather than in disk to improve the performance of the technique.

Only the modified blocks with non-zero elements will be stored in the checkpoint file. The construction of an incremental checkpoint is depicted in Figure 3(a). In order to enable full data recovery during restart, some meta-information needs to be stored together with the checkpoint data. Specifically, an identifier is stored in the checkpoint file for each modified memory block, including modified zero-blocks. This identifier indicates the original position of the block in memory relative to the start of the array. The high-order bit of the identifier is used to mark the zero-blocks that are not included in the checkpoint



(a) Construction of an incremental checkpoint (b) Restart from an incremental checkpoint

Fig. 3 Depiction of the incremental checkpointing technique

file but should be restored during recovery. CPPC uses an integer array called $Block\_ID$ to store the meta-information. The size overhead of storing this array can be calculated as:

$$Overhead = HDF5\_labels + sizeof(Block\_ID) \tag{2}$$

where $HDF5\_labels$ is the number of bytes used by HDF5 to store information about the $Block\_Id$ array (148 if the number of elements of $Block\_ID$ is zero and 892 in any other case). The size of the array of identifiers can be calculated as:

$$sizeof(Block\_ID) = 4\ bytes \times (\#MBlocks) \tag{3}$$

where $\#MBlocks$ is the number of modified blocks. Thus, the overhead varies between 148 and $892 + (4 \times \#TBlocks)$ bytes, being $\#TBlocks$ the total number of memory blocks of the application userspace.

In addition to the checkpointing mechanism, the restart mechanism when using incremental checkpointing also varies. The process of restarting from incremental checkpoints is shown in Figure 3(b). The last available full checkpoint is restored first, and the updates contained in each incremental checkpoint are then applied in an ordered manner.

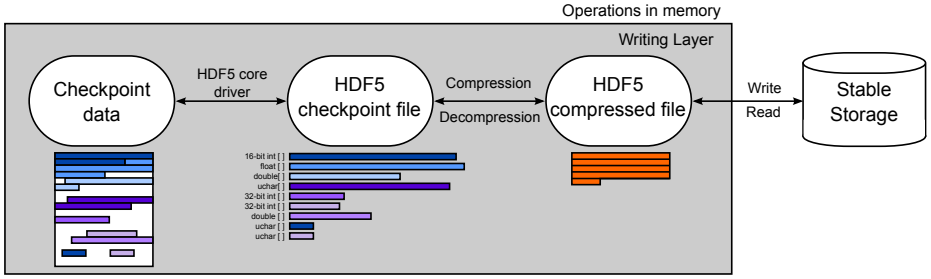


Fig. 4 Integration of the compression process in the CPPC writing layer

4.4 Data compression

To compress the checkpoint files, the CPPC writing layer seen in Section 4.1 must be extended. The HDF5 library provides users with different file drivers which map the logical HDF5 address space to different types of storage. In the current CPPC version the default file driver (`SEC2 driver`) is used to dump the HDF5 data directly to stable storage. This driver was substituted by the HDF5 `core driver` which constructs the HDF5 file in memory.

The checkpoint and recovery processes using compressed files are shown in Figure 4. In order to perform the compression step without storing temporary data to the process local disk first, the `HDF5 File Image Operations` available since HDF5 v1.8.9 are used. These are a set of functions that allow to work with HDF5 files directly in memory. Disk I/O is not required when files are opened, created, read from, or written to. Once the state file is committed to memory, the compression routine is invoked. Afterwards, the compressed data are stored into stable storage. The decompression process is the reverse: the compressed file is read from stable storage to local memory, data are decompressed, and finally the HDF5 is read in place. Note that if compression is disabled, the HDF5 checkpoint file in memory is directly stored into stable storage without compression.

Compression speed is crucial in order to minimize the introduced overhead. In this sense, platform-specific optimizations, such as SIMD instructions, play an important role. Then, the following optimizations are possible: fitting the buffer into one 128-bit register or two 64-bit ones; and performing 16 comparisons with 1 or 2 instructions that produce a 16-bit mask. SIMD instructions are primarily intended to accelerate multimedia processing, and they are commonplace in modern architectures. However, as they are not standardized, the optimized code is not portable. Hence, plain ANSI C code was developed to-

gether with optimized code for x86 and Itanium platforms. At compile time, directives will select which code will be used.

Note that CPPC creates a checkpoint file per process, each one containing a subset of the total data to be stored. The different checkpoint files are simultaneously compressed in the different processes. Thus, compression is implicitly performed in parallel, and its overhead is expected to decrease when increasing the number of processes.

§5 Experimental Results

This section assesses the impact of the described optimization techniques in the size of the checkpoint files and in the execution time overheads. A multicore cluster, Pluton, was used to evaluate our proposal. It consists of 16 nodes, each one of them powered by two Intel Xeon E5620 quad-core CPUs with 16 GB of RAM. The cluster nodes are connected through an Infiniband network. The front-end is powered by one Intel Xeon E5502 quad-core CPU with 4 GB of RAM. The connection between the front-end and the execution nodes is an Infiniband network too. The working directory used for storing checkpoints files is connected to the cluster by a Gigabit Ethernet network and it consists of disks of 2 TB configured in RAID 6.

The application testbed was comprised of the eight applications in the MPI version of the NAS Parallel Benchmarks v3.1 [27] (NPB from now on). These are well-known and widespread applications that provide a de-facto test suite. Out of the NPB suite, the biggest problem size that would fit the available memory was selected for each application. As such, the BT, LU and SP benchmarks were run using class B; the rest were run using class C. All the experiments were executed using 16 and 32-36 processes (32 processes for all the applications except for BT and SP as they require a square number of processes).

The experiments can be divided into two blocks. The first block analyzes the checkpoint size reductions obtained through the use of the proposed techniques. The second block evaluates the execution overhead caused by the computation of the hash functions and data compression, and the restart overhead caused by the restart mechanism in the incremental technique and data decompression.

5.1 Checkpoint file sizes

The reduction in checkpoint file size is the main goal of the techniques

Table 1 Baseline checkpoint sizes (in MB) per process

	16 processes		32-36 processes	
NPB	*SLC*	*ALC Base*	*SLC*	*ALC Base*
BT	97.45	31.36	83.61	17.50
CG	153.05	85.85	114.50	43.30
EP	67.42	1.18	67.42	1.18
FT	514.93	256.14	290.93	128.14
IS	210.50	144.13	138.57	72.13
LU	81.03	14.78	74.86	8.54
MG	288.56	222.32	181.00	114.70
SP	99.00	32.81	86.08	19.87

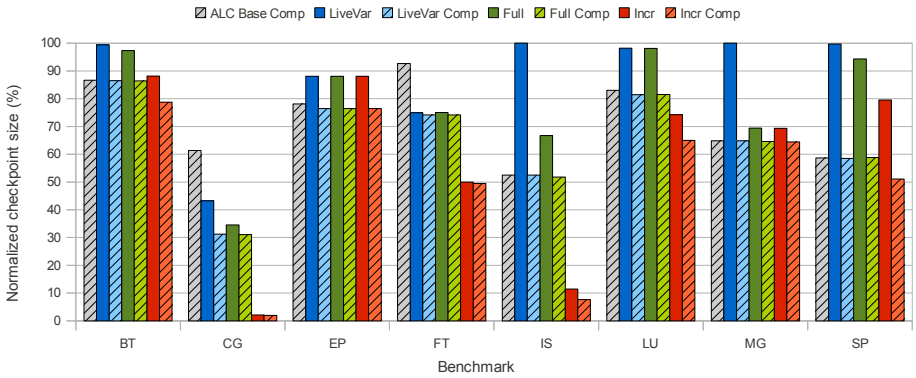


Fig. 5 Checkpoint sizes per process for 16 processes normalized with respect to the ALC base case (see Table 1)

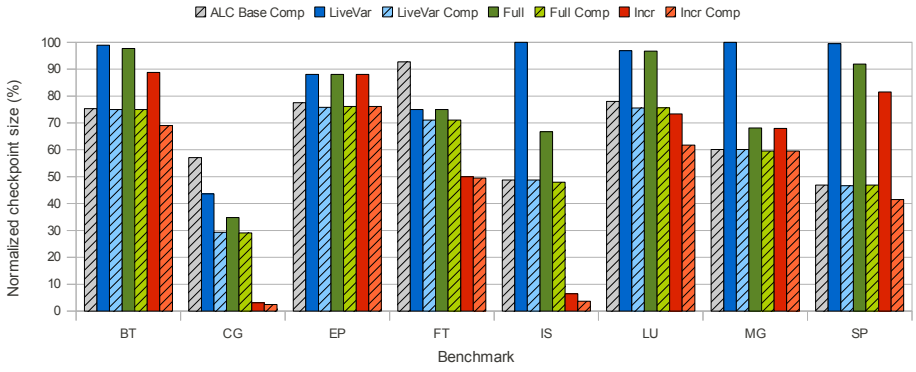


Fig. 6 Checkpoint sizes per process for 32-36 processes normalized with respect to the ALC base case (see Table 1)

described in this work. Table 1 allows to compare the baseline checkpoint sizes per process in the Pluton cluster. The first column (`SLC`) shows results for an SLC approach, the CKPT [40] checkpoint library was used. The second column (`ALC Base`) shows results for an ALC approach without applying any optimization technique, that is, all user variables are stored in the checkpoint file. As can be seen, ALC obtains better results than the SLC approach and its checkpoint files can be further reduced using the optimization techniques proposed in this work. Additionally, the size of the checkpoint files per process decreases more significantly for ALC approaches, which helps obtain scalable fault tolerance. Figures 5 and 6 show normalized checkpoint file sizes with respect to the ALC base case when using the live variable analysis (`LiveVar`) and the incremental checkpointing and zero-blocks exclusion techniques proposed in this paper. Several incremental checkpoints (`Incr`) are created after a full checkpoint (`Full`). However, since their sizes are similar, only the first one is shown in the figures.

The live variable analysis significantly reduces checkpoint file sizes for CG (56% reduction) and FT (25%). It can be concluded that this technique may have great influence on reducing file sizes for certain applications and, as it introduces overhead only at compile time, no application can be adversely affected by its use.

The incremental checkpointing and zero-blocks exclusion technique achieves important file size reductions for almost all the applications. Note that this technique was applied in addition to the live variable analysis. Thus, reductions achieved in the full checkpoint relative to the live variable technique are only due to the elimination of zero-blocks. These reductions vary with the size of the memory block. Figures 5 and 6 show results for the default value of 8K elements per block. Reductions with respect to the ALC base case range from 3% (BT) to 65% (CG) for the full checkpoint and from 12% (BT) to 98% (CG) for the incremental checkpoints.

The results of compressing the checkpoints are also shown in Figures 5 and 6 (`ALC Base Comp, Live Var Comp, Full Comp, Inc Comp`). On average, size reductions of 20% and 25% are achieved for 16 and 32-36 processes, respectively. For some benchmarks, like IS (Integer Sort), it is easy to discern why compression performs better for 32 processes: sorting removes entropy, helping the compressor to find repeated patterns. For other benchmarks the underlying reason may not be so obvious. There are also important differences among benchmarks, as some of them, like FT, are hardly compressible. Also,

Table 2 Baseline checkpoint latency (in s)

	16 proc.	32-36 proc.
NPB	*ALC Base*	*ALC Base*
BT	5.29	6.65
CG	14.38	15.33
EP	0.25	0.30
FT	37.67	38.56
IS	21.90	21.93
LU	2.56	2.69
MG	34.92	35.64
SP	5.71	7.08

Table 3 Baseline restart times (in s)

	16 proc.	32-36 proc.
NPB	*ALC Base*	*ALC Base*
BT	4.52	5.63
CG	12.23	12.88
EP	0.19	0.36
FT	36.54	36.39
IS	20.53	20.49
LU	2.15	2.46
MG	31.69	32.56
SP	4.73	6.38

incremental checkpoints are generally less compressible than the others, as much of the redundant data have been removed from the first checkpoint to the incremental ones. Comparatively, DEFLATE and LZMA would offer an additional 7-10% gain, but with large overhead, as will be discussed in Section 5.2.

5.2 Checkpoint latency

The checkpoint latency is defined as the ellapsed time between the call to the checkpointing function and the return of control to the application. Table 2 shows the baseline checkpoint latency obtained for the different NPB applications for 16 and 32-36 processes. Note that the increase in the number of processes does not have a great influence in the latency times, since a shared filesystem is used and the total amount of data to be dumped remains almost constant. All tables and graphs in this section and the next one display the average data of at least 10 executions.

As regards the incremental checkpointing, some extra time is spent in the computation of the hash functions and the inspections needed. The hash function selected for these experiments was MD5. From the results shown in Figures 7 and 8, it can be observed that the overhead introduced by the incremental checkpointing technique is hidden by the gain obtained from the reduction in checkpoint size. Results for the creation of the full checkpoint in the incremental technique also allow to assess the obtained gain when solely applying the zero-blocks exclusion.

Data compression also allows reducing checkpointing latency in virtually all the tests. The main exception is FT, which contains poorly-compressible data. This gain is possible by the combination of two factors. Firstly, compression allows a significant size reduction, as seen in Section 5.1. Consequently, storage overhead is proportionally reduced. Secondly, compression speed is 85-

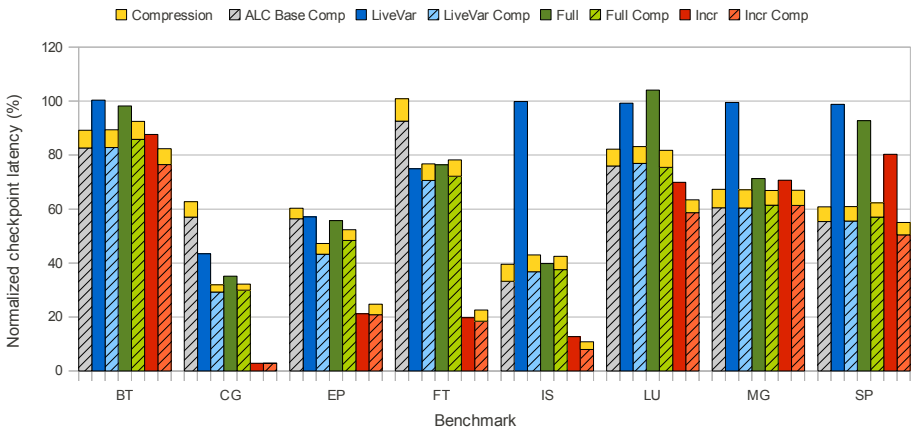


Fig. 7 Checkpoint latency for 16 processes normalized with respect to the ALC base case (see Table 2)

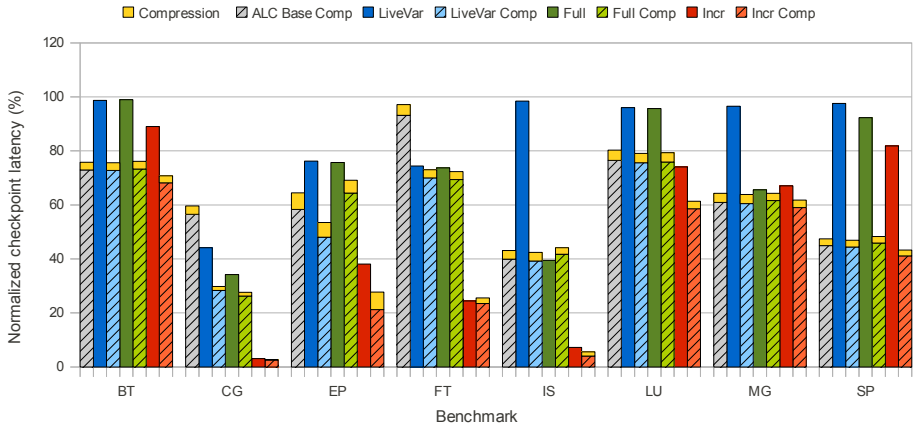


Fig. 8 Checkpoint latency for 32-36 processes normalized with respect to the ALC base case (see Table 2)

90 MB/s on average. That is close to the maximum bandwidth of the Gigabit network on which the testbed storage system is based upon. Therefore, our compression system adds very little overhead to checkpointing, 7% on average (the compression overhead is labeled as `Compression` in the Figures). In comparison, DEFLATE and LZMA are, respectively, 3 and 12 times slower, which makes them impractical alternatives (the gain does not compensate the overhead introduced by the compression).

Although the compression algorithm proposed in this work has been applied to an ALC approach, it could equally be applied to SLC. We have experimentally tested that it is also viable for compressing SLC checkpoint files as, unlike the DFLATE and LZMA algorithms, it is fast enough to provide a performance benefit. Nevertheless, compressing SLC checkpoint files will be always computationally more expensive than compressing ALC ones due to the significantly larger sizes. Additionally, the resulting compressed files will be also larger than their ALC counterparts. Thus, starting from ALC checkpointing files will be always a better solution.

As can be seen by comparing Figures 7 and 8, compression overhead drops as the number of processes is increased. This is due to the fact that the total compression workload is shared by more processors. Hence, checkpoint compression in large scale supercomputers will allow to reduce the volume of stored data with almost negligible overhead.

In general, all the proposed techniques perform better than the ALC base approach. In some cases the reduction in latency can be as high as $92 - 97\%$ (IS or CG).

CPPC can be configured so that the checkpoint file is created in parallel with the execution of the application by creating new threads [35]. Thus, the application execution does not need to be stalled until the checkpoints are created, and the latencies may be hidden.

5.3 Restart overhead

Baseline restart times are shown in Table 3. Measured restart times include the read of the checkpoint files and the restart of the application up to the point where the checkpoint was dumped. Write buffers were flushed before each execution to avoid the effect of page cache and to guarantee that checkpoint files are read from disk.

Columns labeled `Full` in Figures 9 and 10 show the restart overhead

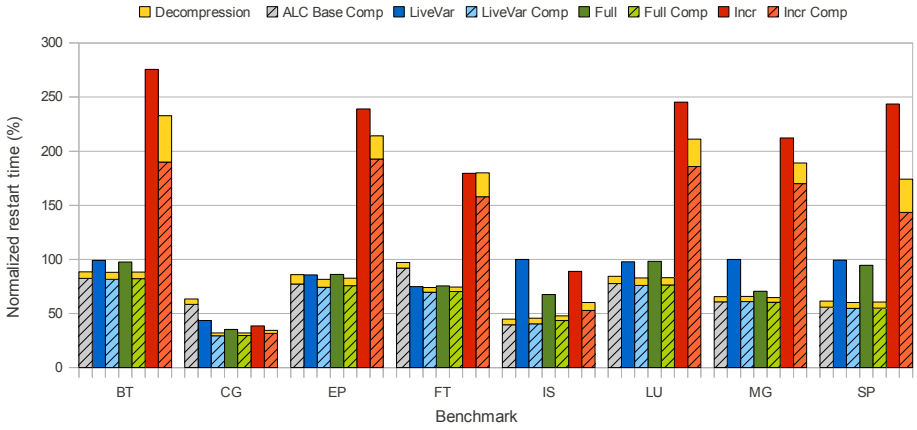


Fig. 9 Restart times for 16 processes normalized with respect to the ALC base case (see Table 3)

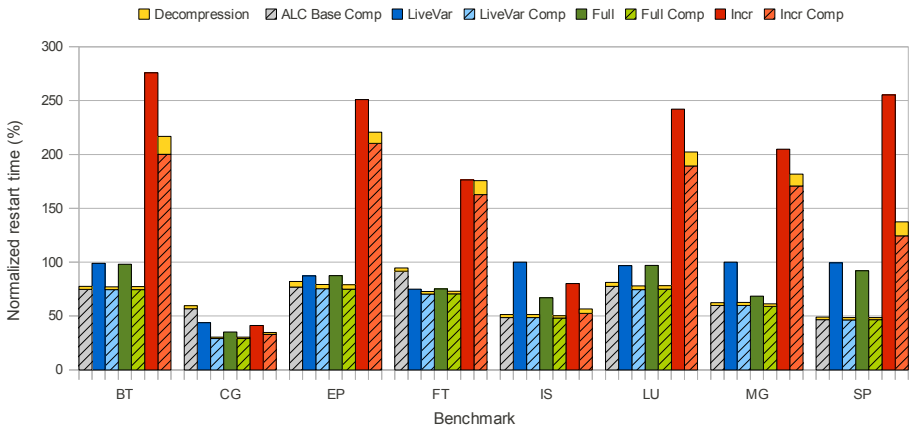


Fig. 10 Restart times for 32-36 processes normalized with respect to the ALC base case (see Table 3)

when there are no incremental checkpoint files, but just the full one. These correspond to the overhead when applying only the zero-blocks exclusion, which is always less than the overhead of the base approach.

The incremental checkpointing technique presents a higher restart overhead compared to the others. This is due to a larger volume of data being moved and read, which can be calculated as the sum of the incremental and full checkpoint file sizes. In these experiments two incremental checkpoints were created after a full one.

Compression has also a positive impact in restart overhead. As data decompression is very fast, the restart process benefits of data reduction with a minimal decompression overhead (see `Decompression` in the Figures). On average, a 20-25% time saving is achieved. When restarting from incremental checkpoints, the overhead is large, and the benefits of compression are more noticeable.

Due to the high influence that the read of the checkpoint files can have on the performance of the restart operation, recent studies are focused on reducing this impact. A post-checkpointing tracking mechanism is presented in [22] to reduce restart latency by overlapping application recovery with the retrieval of checkpoint files. In the case of incremental approaches, the number of incremental checkpoints has great influence in the restart overhead. There exist studies [24] that provide a model to determine the optimal number of incremental checkpoints between two consecutive full checkpoints. A possible approach to reduce the restart overhead would be to merge the full checkpoint file and the incremental ones into a single file at the checkpoint server before a restart is required [1]. Nevertheless, It must be considered that the main object of this work is accelerating checkpoints storage, which is performed several times per execution. Contrarily, restart is a secondary target, as it may never be necessary.

§6 Concluding Remarks

This work has analyzed different alternatives to reduce the size of the checkpoint files generated by ALC approaches: live variable analysis, zero-blocks elimination, incremental checkpointing and data compression. These techniques have been implemented in an ALC tool, CPPC, obtaining important file size and checkpoint latency reductions.

The results have shown that incremental checkpointing is very effective in terms of checkpoint size reduction. However, global storage requirements

increase for this technique, as it is necessary to keep stored at least one full checkpoint and all the associated incremental ones. Additionally, it complicates the restart, introducing an overhead that may become important depending on the number of incremental checkpoints and the characteristics of the network. The results indicate that merging the checkpoint files before transferring them to the computation nodes could significantly reduce restart times.

Data compression also obtains important reductions in checkpoint sizes. Besides, although this technique introduces some overhead due to the compression and decompression step, the fast compression algorithm proposed has proved to be effective in reducing both checkpoint latency and restart overhead.

As regards live variable analysis and zero-block elimination techniques, the checkpoint size reductions obtained are not as significant. However, they decrease globally the storage demand and, as data compression, are able to reduce the overhead of both the checkpoint file writing and the restart phase. At present, our implementation of the live variable analysis does not perform optimal bounds checks for pointer and array variables. This means that they are entirely stored if they are used at any point in the re-executed code. Thus, there is still room for future optimizations in this compilation analysis.

The reduction of the checkpoint sizes will be particularly useful for parallel applications with a large number of parallel processes, where the transference of a large amount of checkpoint data to stable storage can saturate the network and cause a drop in application performance.

Finally, the implementation in the application-level is a key aspect of the proposal. On one hand, it allows a more efficient implementation of the proposed techniques. On the other hand, it does not make any assumptions about the underlying system hardware/software characteristics, thus enabling portable operation.

Acknowledgment This research was supported by the Ministry of Science and Innovation of Spain (Project TIN2010-16735) and by the Galician Goverment (Project 10PXIB105180PR).

References

1) Agarwal, S., Garg, R., and Gupta, M. S. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th Annual International Conference on Supercomputing (ICS'04)* (Saint Malo, France, 26 June–01 July 2004), ACM, New York, pp. 277–286.

2) Bosilca, G., Delmas, R., Dongarra, J., and Langou, J. Algorithm-based fault tolerance applied to high performance computing. *J. Parallel Distrib. Comput. 69*, 4 (2009), 410–416.

3) Cappello, F. Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. *International Journal of High Performance Computing Applications, IJHPCA 23*, 3 (2009), 212–226.

4) Chen, Z., Fagg, G. E., Gabriel, E., Langou, J., Angskun, T., Bosilca, G., and Dongarra, J. Fault tolerant high performance computing by a coding approach. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2005), PPoPP '05, ACM, pp. 213–223.

5) Chiu, G.-M., and Chiu, J.-F. A new diskless checkpointing approach for multiple processor failures. *IEEE Transactions on Dependable and Secure Computing 8*, 4 (2011), 481–493.

6) Elliott, J., Kharbas, K., Fiala, D., Mueller, F., Ferreira, K. B., and Engelmann, C. Combining partial redundancy and checkpointing for hpc. In *2012 IEEE 32nd International Conference on Distributed Computing Systems, Macau, China, June 18-21, 2012* (2012), pp. 615–626.

7) Elnozahy, E., Alvisi, L., Wang, Y.-M., and Johnson, D. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys 34*, 3 (2002), 375–408.

8) Elnozahy, E., Johnson, D., and Zwaenepoel, W. The performance of consistent checkpointing. In *Proceedings of the 11th Symposium on Reliable Distributed Systems, 1992.* (Oct. 1992), pp. 39 –47.

9) Ferreira, K. B., Riesen, R., Brightwell, R., Bridges, P. G., and Arnold, D. libhashckpt: Hash-based incremental checkpointing using gpu's. In *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings* (2011), pp. 272–281.

10) Ferreira, K. B., Stearley, J., Laros, J. H., Oldfield, R., Pedretti, K. T., Brightwell, R., Riesen, R., Bridges, P. G., and Arnold, D. Evaluating the viability of process replication reliability for exascale systems. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011* (2011), pp. 1–44.

11) Gharaibeh, A., Al-Kiswany, S., Gopalakrishnan, S., and Ripeanu, M. A gpu accelerated storage system. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC 2010, Chicago, Illinois, USA, June 21-25, 2010* (2010), pp. 167–178.

12) Gioiosa, R., Sancho, J. C., Jiang, S., and Petrini, F. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing, November 12-18, 2005, Seattle, WA, USA* (2005), p. 9.

13) Gomez, L. A. B., Maruyama, N., Cappello, F., and Matsuoka, S. Distributed diskless checkpoint for large scale systems. In *Proceedings of the 2010*

10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (Washington, DC, USA, 2010), CCGRID '10, IEEE Computer Society, pp. 63–72.

14) Huffman, D. A. A method for the construction of minimum-redundancy codes. In *Proceedings of the Institute of Radio Engineers* (September 1952), vol. 40, pp. 1098–1101.

15) Hursey, J., and Lumsdaine, A. A composable runtime recovery policy framework supporting resilient HPC applications. Tech. Rep. TR686, Indiana University, Bloomington, Indiana, USA, August 2010.

16) IEEE Global History Network. History of lossless data compression algorithms. `http://www.ieeeghn.org/wiki/index.php/History_of_Lossless_Data_Compression_Algorithms`. Last accessed October 2012.

17) Iskra, K., Romein, J. W., Yoshii, K., and Beckman, P. Zoid: I/o-forwarding infrastructure for petascale architectures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (2008), PPoPP '08, ACM, pp. 153–162.

18) Jin, H., Ke, T., Chen, Y., and Sun, X.-H. Checkpointing orchestration: Toward a scalable hpc fault-tolerant environment. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012, Ottawa, Canada, May 13-16, 2012* (2012), pp. 276–283.

19) Laurence Berkeley National Laboratory. Berkeley Lab Checkpoint/Restart. https://ftg.lbl.gov/CheckpointRestart/. Last accessed October 2012.

20) Li, C.-C., and Fuchs, W. Catch-compiler-assisted techniques for checkpointing. In *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium* (June 1990), pp. 74 –81.

21) Li, M., Vazhkudai, S. S., Butt, A. R., Meng, F., Ma, X., Kim, Y., Engelmann, C., and Shipman, G. M. Functional partitioning to optimize end-to-end performance on many-core architectures. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA, November 13-19, 2010* (2010), pp. 1–12.

22) Li, Y., and Lan, Z. FREM: A fast restart mechanism for general checkpoint/restart. *IEEE Transactions on Computers 60*, 5 (2011), 639–652.

23) Moody, A., Bronevetsky, G., Mohror, K., and de Supinski, B. R. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA, November 13-19, 2010* (2010), pp. 1–11.

24) Naksinehaboon, N., Liu, Y., Leangsuksun, C. B., Nassar, R., Paun, M., and Scott, S. L. Reliability-aware approach: An incremental checkpoint/restart model in hpc environments. In *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid* (2008), pp. 783–788.

25) Nam, H.-C., Kim, J., Hong, S., and Lee, S. Probabilistic checkpointing. In *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on* (June 1997), pp. 48 –57.

26) Nam, H.-C., Kim, J., Hong, S. J., and Lee, S. Secure checkpointing. *Journal of Systems Architecture 48*, 8-10 (2003), 237–254.

27) National Aeronautics and Space Administration. The NAS Parallel Benchmarks. `http://www.nas.nasa.gov/Software/NPB`. Last accessed October 2012.

28) Norman, A., and Lin, C. A scalable algorithm for compiler-placed staggered checkpointing. In *Proceedings of the 23rd International Conference on Parallel and Distributed Computing and Systems (PDCS 2011)* (2012), Acta Press.

29) Oldfield, R., Arunagiri, S., Teller, P. J., Seelam, S. R., Varela, M. R., Riesen, R., and Roth, P. C. Modeling the impact of checkpoints on next-generation systems. In *24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007), 24-27 September 2007, San Diego, California, USA* (2007), pp. 30–46.

30) Plank, J., Beck, M., and Kingsley, G. Compiler-assisted memory exclusion for fast checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments 7*, 4 (1995), 10–14.

31) Plank, J. S., Beck, M., Kingsley, G., and Li, K. Libckpt: Transparent Checkpointing under Unix. In *Usenix Winter Technical Conference* (January 1995), pp. 213–223.

32) Plank, J. S., and Li, K. ickp: A consistent checkpointer for multicomputers. *IEEE Parallel Distrib. Technol. 2* (June 1994), 62–67.

33) Plank, J. S., Li, K., and Puening, M. A. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems 9*, 10 (October 1998), 972–986.

34) Plank, J. S., Xu, J., and Netzer, R. H. B. Compressed differences: an algorithm for fast incremental checkpointing. Tech. Rep. CS-95-302, University of Tennessee, Department of Computer Science, Aug. 1995.

35) Rodríguez, G., Martín, M. J., González, P., and Touriño, J. Analysis of performance-impacting factors on checkpointing frameworks: the CPPC case study. *The Computer Journal 54*, 11 (2011), 1821–1837.

36) Rodríguez, G., Martín, M. J., González, P., Touriño, J., and Doallo, R. CPPC: A compiler-assisted tool for portable checkpointing of message-passing applications. *Concurrency and Computation: Practice and Experience 22*, 6 (2010), 749–766.

37) Schroeder, B., and Gibson, G. A large-scale study of failures in high-performance computing systems. *IEEE Trans. Dependable Secur. Comput. 7*, 4 (Oct. 2010), 337–351.

38) Schroeder, B., and Gibson, G. A. Understanding failures in petascale computers. *Journal of Physics: Conference Series 78*, 1 (2007), 012–022.

39) The HDF5 Group. HDF-5: Hierarchical Data Format. `http://www.hdfgroup.org/HDF5/`. Last accessed October 2012.

40) Victor C. Zandy. CKPT process checkpoint library. `http://pages.cs.wisc.edu/~zandy/ckpt/`. Last accessed October 2012.

41) Wang, C., Mueller, F., Engelmann, C., and Scott, S. L. Hybrid checkpointing for mpi jobs in hpc environments. In *IEEE 16th International Conference on Parallel and Distributed Systems, ICPADS 2010, 8-10 Dec. 2010, Shanghai, China* (2010), pp. 524–533.

42) Wang, C., Mueller, F., Engelmann, C., and Scott, S. L. Proactive process-level live migration and back migration in hpc environments. *J. Parallel Distrib. Comput. 72*, 2 (Feb. 2012), 254–267.

43) Zheng, G., Ni, X., and Kalé, L. V. A scalable double in-memory checkpoint and restart scheme towards exascale. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, DSN 2012, Boston, MA, USA, June 25-28, 2012* (2012), pp. 1–6.