The Computer Journal Advance Access published March 3, 2011

doi:10.1093/comjnl/bxr018

Analysis of Performance-impacting Factors on Checkpointing Frameworks: The CPPC Case Study

Gabriel Rodríguez*, María J. Martín, Patricia González and Juan Touriño

Computer Architecture Group, Department of Electronics and Systems, University of A Coruña, Spain
**Corresponding author: gabriel.rodriguez@udc.es*

This paper focuses on the performance evaluation of Compiler for Portable Checkpointing (CPPC), a tool for the checkpointing of parallel message-passing applications. Its performance and the factors that impact it are transparently and rigorously identified and assessed. The tests were performed on a public supercomputing infrastructure, using a large number of very different applications and showing excellent results in terms of performance and effort required for integration into user codes. Statistical analysis techniques have been used to better approximate the performance of the tool. Quantitative and qualitative comparisons with other rollback-recovery approaches to fault tolerance are also included. All these data and comparisons are then discussed in an effort to extract meaningful conclusions about the state-of-the-art and future research trends in the rollback-recovery field.

Keywords: fault tolerance; checkpointing; parallel programming; message-passing; MPI

Received 22 October 2010; revised 30 December 2010
Handling editor: Erol Gelenbe

1. INTRODUCTION

Experimentation conducted on different high-performance computing systems reveals the mean time to interrupt of these platforms to be relatively low [1]. Furthermore, increasing the number of cores in the system causes a proportional growth in failure rates. In this situation, it becomes increasingly difficult for long-running application executions to progress in the absence of fault tolerance mechanisms.

Checkpointing has become a widely used technique to obtain fault tolerance. It periodically saves the computation state to stable storage, so that the application execution can be resumed by restoring such a state. A number of solutions and techniques have been proposed [2], each having its own pros and cons.

Current trends toward new computing infrastructures, such as large heterogeneous clusters and Grid systems, present new constraints for checkpointing techniques. Heterogeneity makes it impossible to apply traditional state-saving techniques, which use non-portable strategies for recovering opaque structures such as application stack, heap or communication state. In this context, modern checkpointing techniques need to provide strategies for portable state recovery, where the computation can be resumed on a wide range of machines, from binary incompatible architectures to incompatible versions of software facilities, such as different implementations for communication interfaces.

Some mathematical models to predict the principal performance measures of fault-tolerant systems have been developed [3–6]. However, the complexity and stochastic behavior of current supercomputers running multitasking operating systems makes it difficult to devise mathematical equations to model the behavior of such systems as a whole. This paper presents a thorough performance evaluation of ComPiler for Portable Checkpointing (CPPC) [7], a checkpointing framework for message-passing applications with an emphasis on portability. The fundamental factors impacting the performance of fault-tolerant jobs are identified, and their impact in application runtimes is qualitatively and quantitatively addressed.

The structure of the paper is as follows. Section 2 describes CPPC's design and implementation, focusing on the distinguishing characteristics of its checkpointing approach. Section 3 presents the experimental results for both the automatic instrumentation of parallel applications and the runtime behavior, in terms of size and creation time of state files, checkpoint overhead and restart overhead in case of failure. Section 4 covers related work, drawing qualitative and quantitative comparisons between the proposed approach and the literature. Finally, Section 5 concludes the paper.

2. THE CPPC FRAMEWORK

CPPC is a checkpointing tool focused on the insertion of fault tolerance into long-running message-passing applications. It is an open-source tool, available at http://cppc.des.udc.es under GNU general public license (GPL) license. It consists of a runtime library containing checkpoint-support routines together with a compiler that automates the use of the library. The remainder of this section summarizes various fundamental design aspects of the CPPC framework. For an in-depth description of the design and implementation of CPPC, the reader is referred to [7].

2.1. Portability

The vast majority of checkpointing research has focused on systems implemented either inside the operating system (OS) kernel or immediately above the OS interface. This kind of solutions generally become locked into the platform for which they were originally developed. For instance, when checkpointing parallel communication APIs, such as message passing interface (MPI), the typical approach has been to modify the existing implementations of such APIs. This is, for instance, the case of MPICH-IG [8], Open MPI [9, 10] or MPICH-V [11, 12], among others. The problem arises when, for this approach to be practical, it becomes necessary to adopt the modified libraries in real systems, which use highly tuned and optimized communication libraries. CPPC, instead, is implemented at the application level [13], and thus it is independent of the underlying communications application programming interface.

CPPC aims to achieve portable restart of high-performance applications in heterogeneous environments. A state file is said to be portable if it can be used to restart the computation on an architecture (or OS) different from the one that generated the file. To achieve portability, state files should not contain an architecture-dependent state. Rather, this state should be recovered at the restart time using special protocols. The solution used in CPPC is to recover the non-portable state by means of the re-execution of the code responsible for creating such an opaque state in the original execution. Moreover, CPPC uses a portable checkpoint format based on the Hierarchical Data Format 5 (HDF-5) [14], a data format and associated library for the portable transfer of graphical and numerical data between computers. This enables the restart on different architectures. Portable offsets [15] are used to store pointers in a way that enables aliasing relationships to be preserved throughout application restarts.

2.2. State file sizes

The solution of large real scientific problems requires the use of large computational resources, both in terms of CPU effort and memory. Thus, many scientific applications are developed to be run on a large number of processors. The *full checkpointing* of this kind of applications, which consists in the storage of the entire application state, including structures such as the application stack or heap, leads to a large amount of stored state, the cost being so high as to become impractical [1]. Besides, the size of the state files is one of the most significant performance-impacting factors in checkpointing. CPPC reduces the amount of data to be saved by working at the *variable level* (i.e. storing user variables only) and performing a live variable analysis that identifies those variable values that are needed for the correct restart of the execution. The process of marking a variable to be included in subsequent state files is called variable registration. Besides, compression of the checkpoint data can be enabled. This not only does help save disk space and network transfers (if needed), but also can improve checkpointing performance when working with large data sets with high compression rates. A multithreaded dumping option is also provided to overlap the application execution and the generation of the checkpoint file. If a failure occurs in the checkpointing thread, inconsistent checkpoint files may be created. CPPC generates a CRC-32 code for the checkpoint file. This redundancy code is checked upon restart to ensure file correctness.

2.3. Coordination protocol

When checkpointing parallel message-passing applications, the dependencies created by interprocess communications have to be preserved during recovery. If a checkpoint is placed in the code between two matching communication statements, an inconsistency will occur when restarting the application, since the first one will not be executed. If it is a send statement, the message will not be resent and becomes an in-transit message. If it is a receive statement, the message will not be received, becoming an inconsistent message.

Checkpoint consistency has been well studied in the last decade [2]. Approaches to consistent recovery can be categorized into two main protocols: uncoordinated and coordinated. In uncoordinated checkpoint protocols, the state of each process is saved independently of the others, leading to the so-called domino effect (processes may be forced to roll back up to the beginning of the execution). Because of this, these protocols are not used in practice. As an alternative, uncoordinated checkpointing may be combined with message logging to avoid the domino effect at the expense of a high overhead on communication latencies [16]. Coordinated checkpointing synchronizes the individual checkpoints of all processes to ensure that the set of all process checkpoints is a globally consistent state of the system. Coordinated approaches are the most common practical choice [10, 17–19] due to the simplicity of recovery. However, an important drawback of coordinated protocols is their scalability [16].

CPPC avoids the runtime overhead of classical consistency protocols by focusing on simple program multiple data (SPMD) parallel applications and using a non-blocking spatially coordinated approach [20]. Checkpoints are taken at the same relative code locations by all processes, but not forcibly at the same time. By statically ensuring that checkpoints occur at the selected places, no interprocess communications or runtime synchronizations are necessary. In order to avoid synchronization problems caused by messages between processes, checkpoints must be taken at points where it is guaranteed that there are no in-transit, nor inconsistent messages. These points will be called *safe points*. This coordination protocol achieves to improve both efficiency and scalability by transferring consistency concerns from runtime to compile time and restart time. The restart is divided into three phases: negotiation, file read and state recovery. During the negotiation step, application processes identify the most recent valid recovery line, formed by the newest checkpoint available simultaneously to all processes. Once each process has selected its appropriate local checkpoint file, in the second phase the file contents are read into memory. Finally, in the third step, the application state is effectively recovered.

2.4. CPPC compiler

In early stages of the work, the user was responsible for inserting compiler directives to guide the operation of the runtime library [21]. Currently, all analyses and code transformations are transparently applied by a compiler that translates the application source files into derived code with added checkpointing capabilities. The global process is depicted in Fig. 1.

The compiler performs code analyses and transformations that insert checkpoint instrumentation into an application. The most relevant of these transformations are: the detection of variable values that are necessary during a restart, the automatic safe point discovery and the insertion of checkpoint operations at automatically selected safe points.

In order to identify the variables needed upon application restart, the compiler performs an interprocedural live variable

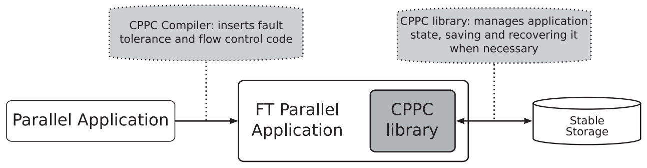

CPPC Compiler: inserts fault tolerance and flow control code
CPPC library: manages application state, saving and recovering it when necessary
Parallel Application
FT Parallel Application
CPPC library
Stable Storage

FIGURE 1. CPPC framework design.

analysis. This is a complementary approach to memory exclusion techniques used in sequential checkpointers to reduce the amount of memory stored, such as the one proposed in [22]. The compiler does not perform optimal bounds checks for pointer and array variables. This means that some arrays and pointers are conservatively marked as necessary during restart, and thus stored in state files.

To automatically discover regions in the code where neither in-transit nor inconsistent communications exist, the compiler analyzes communication statements and matches message sends to their respective receives. The approach used is similar to a static partial simulation of the execution, requiring constant propagation and aggressive symbolic analysis. Only statements that affect the outcome of communications are analyzed, thus avoiding analysis of irrelevant code. A statement in the application code is considered a safe point if, and only if, the analysis does not detect any pending communications upon reaching the location of the said statement.

Some parallel applications present irregular communication patterns (those that depend on runtime input data) or non-deterministic communications (which use wildcard receives). In these situations, the information available at compile time may not be enough to completely determine how the communications will play out during runtime. To ensure the correctness of the results, a conservative solution is used. This approach is based on considering any of the potential matches and deferring the match to the latest possible one in the code. The correctness of the results is guaranteed, although some actual safe points may not be considered as such for the sake of consistency.

For automatic checkpoint insertion, the compiler locates sections of the code that take a long time to execute, where checkpoints would be needed in order to guarantee execution progress in the presence of failures. Since the time to compute a code section cannot be accurately predicted at compile time without the knowledge of the computing platform and input data, heuristics are used. The compiler discards any code location that is not inside a loop, and ranks all loop nests in the code using computational metrics. Once the loop nests in which checkpoints are to be inserted are identified, the compiler uses the results of the previous communication analysis and inserts a checkpoint at the first available safe point in each selected nest.

The CPPC compiler is built on the Cetus compiler infrastructure [23], which is written in Java and thus inherently portable. Although Cetus was originally designed to support C codes, we have extended it for parsing Fortran 77 codes. More details about the CPPC compilation analyses and code transformations are provided in [7, 20].

3. PERFORMANCE EVALUATION

This section is divided into two main parts: the evaluation of the compiler, which performs the transformations to instrument the checkpointing of parallel codes, and the experimental results obtained by the runtime library containing the routines for checkpoint and restart. Twelve applications were selected for testing, separated into three different categories. The first category is formed by the eight applications in the NPB-MPI v3.1 benchmarks [24]. They are well known and widespread, which makes them a good option for comparison purposes. Each kernel has several workloads to scale from small systems to supercomputers. The second category contains two scientific applications in use in the Supercomputing Center of Galicia (CESGA), called *CalcuNetw* [25] and *Fekete* [26]. While these applications do not have particularly long runs, they are good choices for evaluating the performance of the framework when checkpointing typical applications used in supercomputing centers. Finally, two large-scale applications called DBEM [27] and STEM-II [28] were added to test the tool in long-running programs. Table 1 gives more details about the test applications used.

3.1. Compiler performance

A summary of the static characteristics of the test codes is shown in Table 2, including number of files, lines of code (LOCs), the time it takes the CPPC compiler to instrument them, number of

TABLE 1. Summary of test applications.

Source	Application	Description
NAS NPB-MPI v3.1	BT (class = B)	A simulated computational fluid dynamics (CFD) application that uses an implicit algorithm to solve 3D compressible Navier–Stokes equations. The resulting systems are Block-Tridiagonal of 5×5 blocks and are solved sequentially along each dimension.
	CG (class = C)	Uses a Conjugate Gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix.
	EP (class = C)	Embarrassingly Parallel benchmark. It generates pairs of Gaussian random deviates according to a specific scheme.
	FT (class = B)	Contains the computational kernel of a Fast Fourier Transform (FFT)-based spectral method. It performs three 1D FFTs, one for each dimension.
	IS (class = C)	Integer Sort: It works with a list of small integer values, not really sorting them but assigning every list member a number indicating the position in the sorted list.
	LU (class = B)	A simulated CFD application that uses a symmetric successive over-relaxation (SSOR) method to solve a seven-block-diagonal system by splitting it into block Lower and Upper triangular systems.
	MG (class = B)	Uses a V-cycle MultiGrid to compute the solution of the 3D scalar Poisson equation. The algorithm works continuously on a set of grids that are made between coarse and fine.
	SP (class = B)	A simulated CFD application that has a similar structure to BT. The finite differences solution to the problem is based on a Beam-Warming approximate factorization that decouples the x, y and z dimensions. The resulting system has Scalar Pentadiagonal bands of linear equations that are solved sequentially along each dimension.
CESGA	CalcuNetw [25]	Calculates some characterization measurements in a given network, consisting of a set of nodes or vertices joined together in pairs by links or edges, and compares it with a number of random networks specified by the user. The program calculates the subgraph centrality, bipartivity and network communicability.
	Fekete [26]	Determines the position of a certain number of points on a 2D sphere such that the potential energy produced by the interaction of these points is minimum. This is the seventh of the Smale's problems [45].
Others	DBEM [27]	Crack growth analysis using the Dual Boundary Element Method. The analysis leads to a large number of discretized equations that grow at every step when the crack growth is evaluated. It solves the resulting dense linear system using the generalized minimal residual (GMRES) iterative method, regarded as the most robust of the Krylov subspace iterative methods.
	STEM-II [28]	Used to know in advance how the meteorological conditions, obtained from a meteorological prediction model, would affect the emissions of pollutants by the power plant of As Pontes (A Coruña, Spain) in order to fulfill EU regulations. The underlying equation is a time-dependent, 3D partial differential atmospheric-diffusion equation.

TABLE 2. Characteristics and compilation times for test applications.

Application	Files	LOCs	Compile time (s)	Loop nests	# Checkpoints	# Registers
BT	18	3650	14.19	25	1	121
CG	1	1044	2.61	13	2	38
EP	1	180	0.90	4	1	14
FT	1	1269	4.82	20	1	31
IS	1	672	3.88	6	3	34
LU	25	3086	7.78	35	1	74
MG	1	1618	13.08	12	1	34
SP	24	3148	13.69	25	4	143
CalcuNetw	5	810	24.55	14	1	28
Fekete	1	182	0.86	6	1	17
DBEM	42	12533	77.14	92	4	279
STEM-II	110	6506	15.14	24	1	94

TABLE 3. Breakdown of compilation times for test applications (s).

Application	Total time	Data flow	Communications analysis	Checkpoint insertion	Rest
BT	14.19	3.17	2.45	0.85	7.72
CG	2.61	0.52	0.64	0.08	1.37
EP	0.90	0.12	0.08	0.01	0.69
FT	4.82	1.18	0.84	0.27	2.53
IS	3.88	1.04	1.68	0.05	1.11
LU	7.78	1.92	1.30	0.39	4.17
MG	13.08	1.56	7.71	0.32	3.49
SP	13.69	2.21	1.76	0.78	8.94
CalcuNetw	24.55	21.96	0.00[a]	0.20	2.39
Fekete	0.86	0.15	0.07	0.02	0.62
DBEM	77.14	17.20	15.23	20.33	24.38
STEM-II	15.14	5.95	3.55	1.45	4.19

[a]CalcuNetw is a sequential application.

loop nests in the code and number of checkpoints and variable registrations inserted by the CPPC compiler. Compilation times were measured in a desktop machine, an Intel Core2 Duo CPU at 3.00 GHz with 1 GB of RAM. Although the CPPC compiler is not yet optimized for production use, it can be seen that compile times are acceptable for all test applications, and mostly dependent on the number of source LOCs. The highest compile time, 77.14 s, is obtained for the DBEM application, which contains 12 533 LOCs.

A more in-depth analysis of compilation times is given in Table 3. Times have been broken down into the times consumed by each of the foremost analyses performed by the compiler: data flow analysis, communication analysis and checkpoint insertion. The column labeled *Rest* includes the combined times for other compilation tasks such as parsing, instrumentation and code generation. Although the number of applications is not nearly enough as to develop a complete mathematical model of execution times, some tendencies can be inferred from these times. The data flow analysis is $O(\#\text{LOC})$. However, it is very dependent on the programming language of the application. If a linear regression model were fitted to the times for the data flow analysis, two different models would be needed for the analysis of C and Fortran 77 applications. Regarding the communications analysis, it also tends to $O(\#\text{LOC})$, but depends on more complex factors like the number of communication-related variables. The checkpoint insertion analysis does not depend on the LOCs of the application, but rather on the number of loop nests ($\#L$), being $O(\#L^2)$. As for the times labeled as *Rest*, they include many compilation passes, such as insertion of the actual registration calls for restart-relevant variables, control flow code, etc. Each of these passes is $O(\#\text{LOC})$.

TABLE 4. Number of nodes and cores used for runtime tests for each application.

Application	Nodes	Cores per node	Total cores
BT	3	12	36
CG	2	16	32
EP	2	16	32
FT	2	16	32
IS	2	16	32
LU	2	16	32
MG	2	16	32
SP	3	12	36
CalcuNetw	1	1	1
Fekete	2	16	32
DBEM	2	16	32
STEM-II	2	16	32

3.2. Runtime performance

Runtime tests of the CPPC library were performed on the Finis Terrae supercomputer hosted by CESGA, which consists of 142 HP Integrity rx7640 nodes with 16 Itanium Montvale cores at 1.6 GHz and 128 GB of RAM per node. The nodes are connected through an Infiniband 4xDDR network with a bandwidth of 20 Gbps. It features a high-performance storage system, the HP scalable file share (SFS), made up of 20 HP Proliant DL380 G5 servers and 72 SFS 20 disk arrays. The SFS storage system is accessed through the Infiniband network. The Finis Terrae software configuration consists of a SuSE Linux Enterprise Server 10 IA64 OS, the Intel C and Fortran compilers (icc and ifort) v10.1.012 and the MPI library HP-MPI v2.2.5.2. Table 4 details the number of nodes used for each application in runtime tests. Whenever possible, two nodes with 16 cores each were used for the execution. However, the NPB-MPI BT and SP applications require the number of parallel processes to be a perfect square, which caused the number of processes to be raised to 36, allocating 3 nodes and using 12 cores of each of them. The remaining 4 cores in each node were allocated to prevent applications belonging to other users from being executed on them, thus disrupting the results. CESGA CalcuNetw is a sequential application. A whole node was allocated for its execution, while only one of its cores was used for the tests.

Measurements taken include: generated state file sizes, times for state file generation, checkpointing overhead and restart times. For proving portability, the applications were cross-restarted on the Finis Terrae using state files generated on the NM cluster, a local cluster hosted by the Computer Architecture Group of the University of A Coruña. It consists of eight execution nodes, each powered by two Intel Xeon dual-core CPUs with 4 GB of RAM. The cluster nodes are connected through an Infiniband network, and the IntelMPI library v4.0.0.0.017 is used for communication. With regard to software configuration, the cluster runs on a Linux Rocks 5.2 v2.6.18-128 OS using the GNU C and Fortran compilers v4.1.2. Note that the OS, compilers and MPI implementations are different from those available in the Finis Terrae.

The NPB-MPI CG, IS and SP and the DBEM application generate two or more different checkpoint files during their execution, due to the compiler detecting more than one loop nest with intensive computation (see column #Checkpoints in Table 2). In order to provide normalized results for the state file sizes, state file creation times and restart times, these parameters were measured for the largest checkpoint file created by each code. In this way, worst case results are reported. This makes it easier to provide graphical comparisons of the obtained results for different applications.

3.2.1. State file sizes

When using CPPC's spatially coordinated technique, the incurred overhead will mainly depend on the overhead introduced by the checkpoint file creation. This overhead heavily depends on the size of the data to be dumped. In this context, it is important to determine how checkpointing at the variable level affects checkpoint file sizes. In order to analyze this effect, state file sizes have been measured for different checkpointing configurations, and compared with sizes obtained using full checkpointing.

For most applications, file sizes vary depending on the number of processes involved in the execution, because the sizes of array data are defined to match the problem size. The exceptions to this rule are NPB-MPI EP, CESGA Fekete, DBEM and STEM-II, which are Fortran 77 codes that statically allocate a fixed array size that determines the maximum allowed problem size. An example of state file size variations is shown in Fig. 2 for the NPB-MPI IS application. The values labeled as 'Automatic' are file sizes obtained by the automatic variable registration included in the compiler. 'Compressed' corresponds to file sizes obtained by enabling the compression included in the HDF-5 library. Compression is only applied to both static and dynamic array variables larger than a certain user-specified threshold, which in this case was set to 2000 elements. For comparison purposes, 'Optimal' shows the optimal file sizes obtained by a manual analysis of live variables, and 'Full checkpointing' presents the sizes obtained for a checkpointer that stores the whole application state. The CKPT library [29] was used for this test. Figure 3 shows the summary of sizes obtained for all applications using the number of cores (processes) shown in Table 4. The high compression rates obtained for DBEM and STEM-II (97.86 and 96.10%, respectively) are due to the already stated fact that these applications statically allocate arrays that are oversized to fit a maximum problem size. As a result, an important amount of empty memory is allocated in our tests, resulting in high compression rates.

Sizes obtained using automatic analyses are close to the optimal ones for most applications. The existing difference is due to the registration of unnecessary array sections because

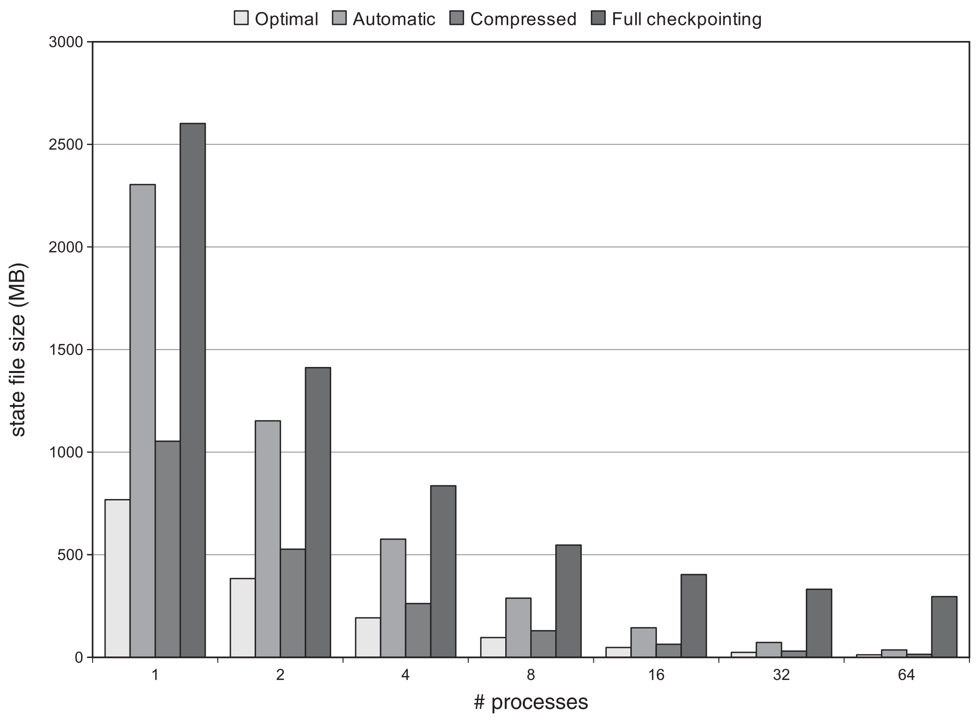


FIGURE 2. Evolution of file sizes with the number of processes for the NPB-MPI IS application.

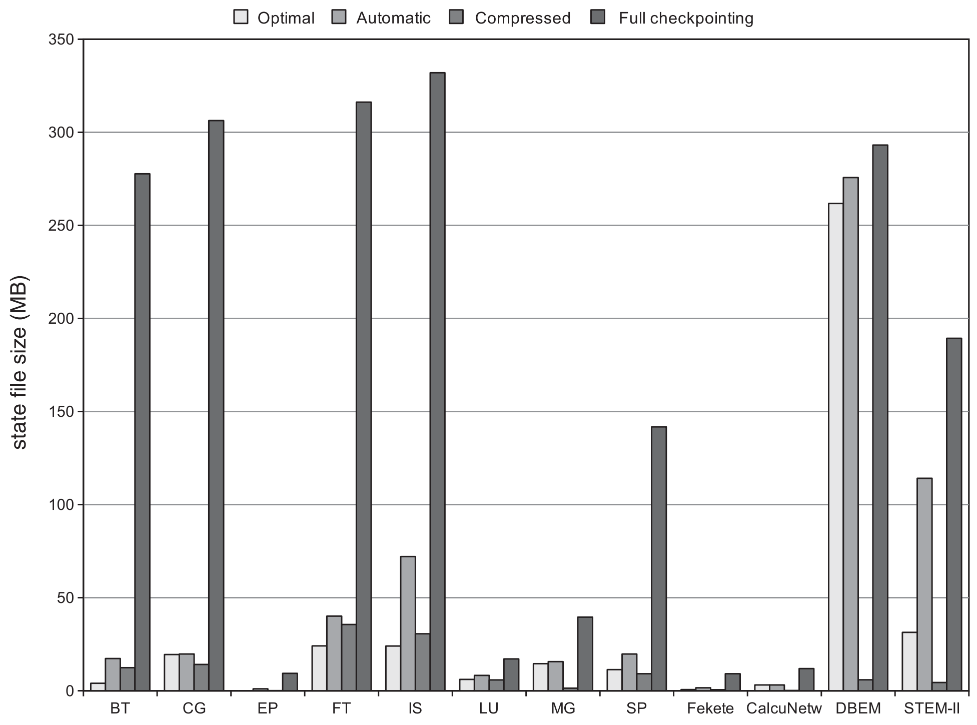


FIGURE 3. Summary of state file sizes.

of the conservative approach employed by the compiler, as explained in Section 2.4. As can be seen, variable level checkpointing achieves very important file size reductions compared with full checkpointing. Table 5 gives the details and reduction percentages for the file sizes depicted in Fig. 3. The table includes the size of the files created by the automatic live variable analysis, the reduction obtained with respect to the size of the files created by full checkpointing and the optimal reduction (bracketed) obtained by a manual analysis.

Throughout the rest of this section, all measurements are taken for codes that use the automatic version of the variable registration.

3.2.2. *State file creation times.*

The Finis Terrae machine exhibits a high variability between the observed runtimes of different executions of the same experiment. Allocating entire nodes reduces this variability, but does not completely solve the problem. Because of this, performing a large number of experiments and statistically analyzing the results is a better approach than just giving the minimum, maximum or mean times for each experiment. The experimental approach consisted in measuring the creation time for state files a minimum of 500 times for each application, followed by the elimination of outliers and the calculation of the 99% confidence interval for the mean. The approach used for outlier identification was to discard observations higher than a certain threshold. This threshold is defined for each application as the third quartile of the data series plus 1.5 times the interquartile range (IQR). This is a classical approach to outlier identification [30]. Table 6 shows, for all the applications, the 99% confidence interval for the mean creation times ($\bar{x}_{\min}$, $\bar{x}_{\max}$) of: (1) a standard HDF-5 state file, (2) the same HDF-5 file including a CRC-32 error detection scheme and (3) the compressed HDF-5 file. The minimum times obtained are also shown for comparison purposes. Note that these times correspond to the raw dumping time of a single checkpoint, not the real contribution of file creation times to the checkpoint overhead, which is reduced by using multithreaded dumping. Also, it should be noted that the width of the confidence interval obtained for the mean creation times of compressed files for the NPB-MPI MG application is relatively large ($\sim$ 0.25 s). This is due to the entropy of the stored data being variable through the execution, causing compression times to decrease as the execution progresses.

Stored data are tagged with their corresponding datatypes by the HDF-5 library to allow for conversions to be performed, if necessary, during the restart of the execution. This improves checkpoint performance by moving the conversion overhead to restart, which should be a much less frequent operation. Figure 4 graphically compares the results shown in Table 6, using the worst case estimated value (i.e. the upper limit of

TABLE 5. Performance of the automatic variable registration algorithm.

Application	File size (MB)	Reduction (Optimal)
BT	17.30	93.77% (98.55%)
CG	19.77	93.55% (93.64%)
EP	1.04	88.87% (99.71%)
FT	40.10	87.32% (92.38%)
IS	72.10	78.28% (92.76%)
LU	8.26	51.81% (64.26%)
MG	15.64	60.49% (63.15%)
SP	19.77	86.05% (91.99%)
Fekete	1.60	82.59% (92.81%)
CalcuNetw	3.12	73.78% (73.80%)
DBEM	275.67	5.94% (10.70%)
STEM-II	114.13	39.72% (83.42%)

TABLE 6. Checkpoint file creation times (s).

	Standard			CRC-32			Compressed		
Application	$\bar{x}_{\min}$	$\bar{x}_{\max}$	Min	$\bar{x}_{\min}$	$\bar{x}_{\max}$	Min	$\bar{x}_{\min}$	$\bar{x}_{\max}$	Min
BT	0.1374	0.1436	0.1162	0.2221	0.2286	0.2016	1.3248	1.3299	1.3095
CG	0.0904	0.0991	0.0896	0.1914	0.1988	0.1728	1.7216	1.7256	1.7093
EP	0.0065	0.0066	0.0061	0.0112	0.0113	0.0108	0.1195	0.1198	0.1187
FT	0.1612	0.1717	0.1259	0.3601	0.3685	0.3217	3.4236	3.4339	3.3957
IS	0.2875	0.2996	0.2457	0.6500	0.6599	0.6008	6.2744	6.3449	6.1869
LU	0.0776	0.0810	0.0661	0.1145	0.1172	0.1040	0.6191	0.6279	0.6006
MG	0.1035	0.1077	0.0890	0.1796	0.1840	0.1666	0.6795	0.9111	0.5113
SP	0.1633	0.1692	0.1471	0.2657	0.2706	0.2509	1.2629	1.2699	1.2387
Fekete	0.0251	0.0275	0.0187	0.0309	0.0322	0.0267	0.0818	0.0824	0.0802
CalcuNetw	0.0207	0.0214	0.0189	0.0353	0.0363	0.0313	0.0964	0.0968	0.0951
DBEM	2.2384	2.2725	2.1339	3.6321	3.6769	3.4345	7.0284	7.0646	6.8588
STEM-II	0.9898	1.0147	0.8724	1.5406	1.5617	1.4581	4.3067	4.3527	4.0857

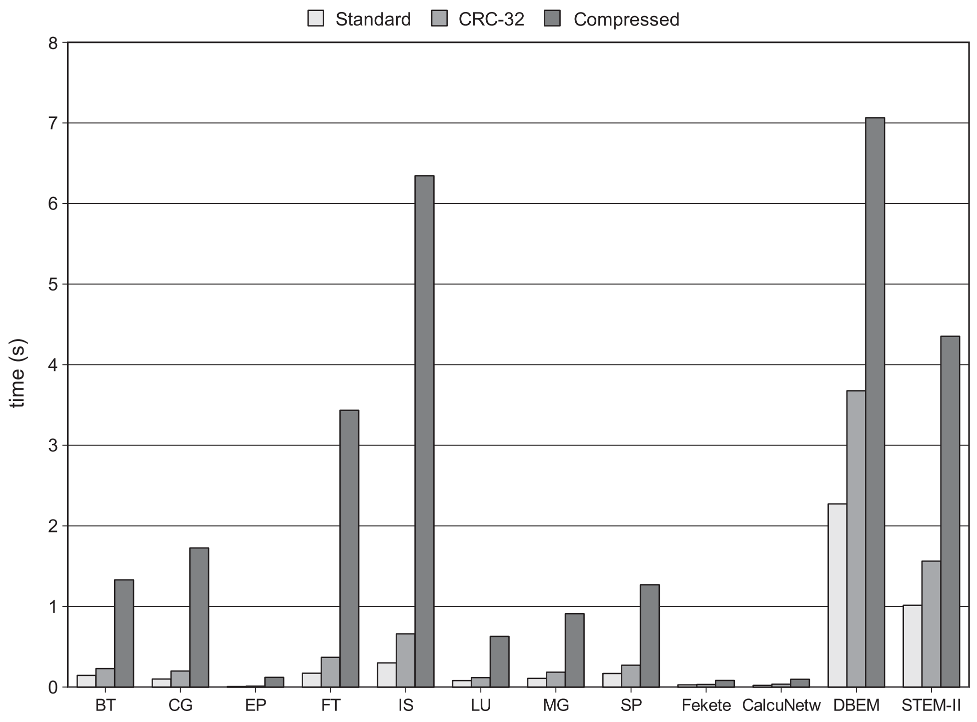


FIGURE 4. Maximum mean dumping times ($\bar{x}_{\max}$) for test applications.

the confidence intervals) for each application. The experimental data also show that using compression notably increases overall dumping times. Therefore, it should be enabled only when the physical size of state files is critical: for instance, if there are problems with disk quotas or when the files are going to be transferred using a slow network.

A regression analysis on the dumping times with respect to the checkpoint file sizes yields a linear relationship between both factors in both the standard and the CRC-32 file creations. Figure 5 shows the obtained function for the standard file creations. The coefficient of determination (R^2) indicates that the state file size alone accounts for a 98.81% of the variability of these times. The regression line grows ∼1 s for each 120 MB of generated data. For the CRC-32 file creation, this percentage increases slightly, up to 99.10%, while the regression line is steeper than in the standard case, growing ∼1 s for each 75 MB of data due to the necessary data hash calculations. Using compression, however, the relationship is not linear, since the dumping time depends on the number of variables to be compressed and the entropy of the data. Hence, R^2 decreases, and the percentage of variability explained by a linear model is only 71.59%.

3.2.3. *Checkpoint overhead*

To reduce the overhead introduced by file generation, multithreaded state dumping has been implemented into the CPPC library. This absorbs a big amount of the state file creation and disk writing times. When performing the overhead tests, the problem of the experimental variability in Finis Terrae has to be carefully dealt with. The load of the allocated nodes dramatically impacts execution times. To handle this situation, full nodes were reserved for the execution of the experimental applications as explained before. Moreover, it is important that all the repetitions of a given experiment are executed on the same nodes and under the same load conditions. To achieve this, once the nodes were allocated, a sequence of N experiments was run on them. Before the start of each experiment a random number was generated, and depending on it being odd or even the original version of the application or the CPPC-instrumented one was run. For the shorter applications, N was limited to 500, a number that should provide statistically representative results. For the larger ones this number was determined by the maximum allowed node allocation time (10 h). Performing more than one allocation is not a valid approach, since the obtained time series may not be assumed to come from the same statistical distribution. This experimental setup is designed to ensure that the variability exhibited by the machine affects all types of experiments to the same degree. Outliers are identified, as in the previous experiment, by removing observations higher than the third quartile plus 1.5 times IQR in each of the series. Table 7 shows the number of regular and CPPC runs performed for each application (excluding outliers), a 99% confidence interval for the mean checkpointing overhead, the minimum execution time for both the regular and CPPC versions of the code and the worst case overhead percentage calculated as the upper limit of the overhead confidence interval divided by the minimum original

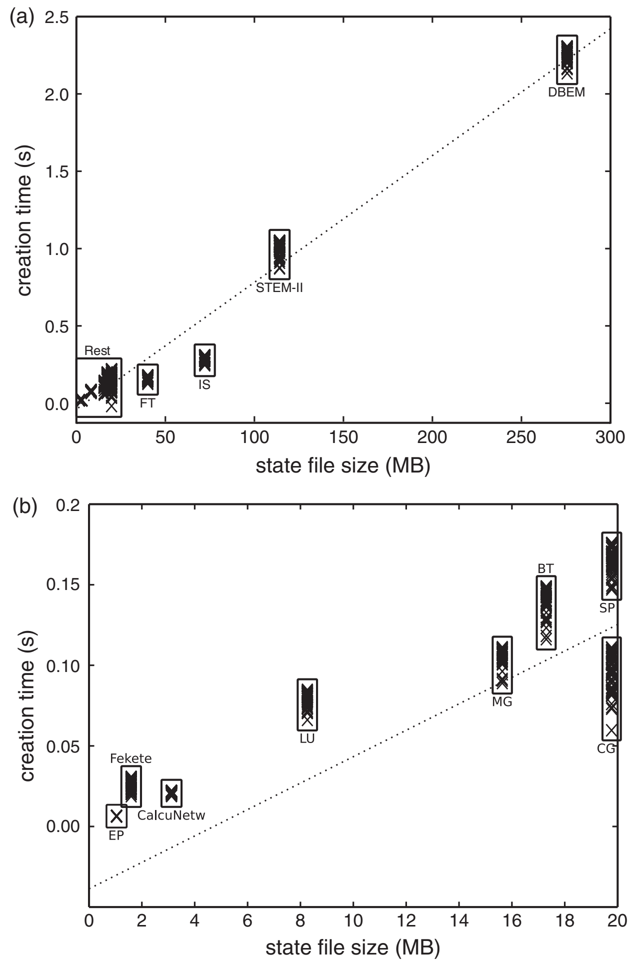


FIGURE 5. Relationship between state file sizes and creation times. (a) Linear regression model for standard file creation time depending on file size ($R^2 = 0.9881$). (b) Zoom of the region labeled "Rest" in (a).

runtime. These overheads include, besides file generation, all CPPC instrumentation (e.g. variable registration). Files were generated with CRC-32 and without compression. For all the applications in the table, one state file per checkpoint introduced by the CPPC compiler was generated (see column #Checkpoints in Table 2).

As can be seen in Table 7, sometimes the minimum execution time for the CPPC version is below the minimum time for the original version of the application. This evidences the need for statistical analyses of execution times. Note that, even when considering the maximum possible overhead at 99% confidence and the minimum time obtained for the execution of the original code, the overhead percentages remain low, usually in the 1% range, except for the IS application. IS runs for ∼25 s and therefore the 9.37% overhead obtained does not account for more than 2.5 s.

The previously detailed experimental setup is not feasible for the long-running applications, DBEM and STEM-II, since they run for more than 22 and 6 h, respectively. Obtaining statistically significant measurements for these applications using the same approach as for the shorter ones would require an unreasonable amount of computation time. For this reason, these applications were run on the NM cluster. This environment exhibits neither the high loads nor the high variability experienced in the Finis Terrae supercomputer, therefore making it possible to reach plausible conclusions without executing a large number of experiments. As such, 10 executions of each version of the codes were run. The checkpointing frequency for these tests was adjusted to approximately 1 checkpoint per hour, meaning that 23 checkpoints were created during the DBEM executions and 7 checkpoints during the STEM-II ones. The selection of the optimum checkpoint interval is a well-studied research area [31–33].

Table 8 details the minimum execution times for the original and CPPC-instrumented versions of both DBEM and STEM-II executed on 4 cores, and the corresponding checkpoint overhead percentage. Generally speaking, the main difference between real-world applications and kernel benchmarks lies in their code sizes and total execution times. For instance, a STEM-II execution lasts for more than 100 times longer than the longest running NPB code. However, checkpoint file sizes do not increase proportionally. Since the time to create a checkpoint file is linearly dependent on its size, the overhead of checkpointing large-scale applications is proportionally much lower than when checkpointing short benchmarks. The table shows how the relative incurred overheads drop down to below 0.5% working with large-scale applications.

3.2.4. Restart times

If a failure occurs, the restart time overhead must be taken into account into the global execution time. Restart times have been measured and split into its three fundamental phases: negotiation, file read and effective data recovery, as described in Section 2.3. As when measuring checkpoint file creation times, 500 experiments were performed, outliers were discarded from each series and a 99% confidence interval was calculated for the mean times for each of the phases. The results for uncompressed files including CRC-32 codes are shown in Table 9. Figure 6a graphically compares the upper limit of the time intervals for each phase. As can be seen, the negotiation is the most costly phase. During the negotiation, application processes have to check the integrity of the checkpoint files available for restarting the execution. If this experiment is performed with standard files without CRC-32 codes, negotiation times are reduced to the range of milliseconds for all applications. If performed on compressed files with CRC-32 codes, negotiation times are reduced due to the files being smaller, thus requiring less time for the integrity checks to be performed.

The times obtained for the file read phase mainly depend on the size of the files themselves, the relationship between both being linear. Read times would be increased if using compressed

TABLE 7. Runtime overhead caused by checkpointing.

Application	Test runs		Overhead (s)		Min. time (s)		Max. % overhead
	Original	CPPC	$\bar{x}_{\min}$	$\bar{x}_{\max}$	Original	CPPC	
BT	157	138	−2.8233	3.6010	562.87	563.43	0.6398
CG	73	74	−1.1995	1.2385	211.07	210.25	0.5868
EP	213	238	0.5300	0.8049	58.71	58.32	1.3710
FT	132	148	2.0734	3.6943	256.81	259.14	1.4385
IS	232	227	1.9252	2.4136	25.76	27.66	9.3696
LU	106	89	−2.7466	5.0750	857.30	861.99	0.5920
MG	249	218	−1.3841	0.9145	66.07	66.87	1.3841
SP	47	63	1.8426	3.1817	774.12	775.48	0.4110
Fekete	247	227	−0.1340	0.1874	51.63	52.02	0.3630
CalcuNetw	92	87	−3.5301	3.9988	351.71	350.23	1.1370

TABLE 8. Runtime overhead of large-scale applications.

Application	Min. runtime (s)		% overhead
	Original	CPPC	
DBEM	80473.13	80729.15	0.31
STEM-II	21622.41	21723.55	0.47

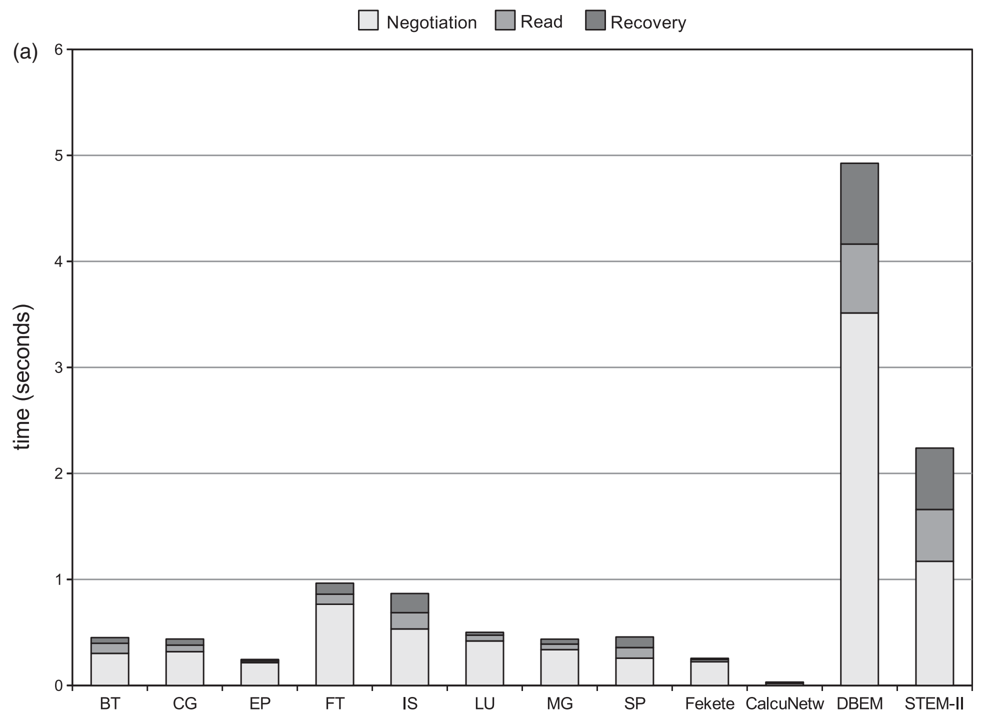
files because of the need for data decompression. Read times may also be increased when using state files generated on different architectures, due to the necessary data conversions. This effect is shown in Fig. 6b, which compares file read times in the Finis Terrae supercomputer for both Finis Terrae- and NM cluster-generated state files, labeled 'Native' and 'Cross-restart', respectively. This test also serves to demonstrate portability, by verifying that restarts take place correctly using externally generated files. Although the tests were performed using the same statistical approach as for the other experiments, only the upper limits of the intervals are shown for simplicity. The relationship between both times depends exclusively on the amount of data that needs to be converted, but the general trend is that the read time increases ∼50% when compared with the original one.

Finally, data recovery times depend on the amount of data being recovered and the amount of code that must be re-executed in order to achieve complete state recovery.

Note that the time for the failure detection is not being taken into account in these experiments. Failure detection techniques are well studied in the literature [34].

TABLE 9. Restart times (s).

Application	Negotiation		File read		Recovery	
	$\bar{x}_{\min}$	$\bar{x}_{\max}$	$\bar{x}_{\min}$	$\bar{x}_{\max}$	$\bar{x}_{\min}$	$\bar{x}_{\max}$
BT	0.2668	0.3027	0.0913	0.0947	0.0507	0.0534
CG	0.2845	0.3177	0.0608	0.0629	0.0551	0.0568
EP	0.1714	0.2145	0.0152	0.0162	0.0132	0.0148
FT	0.6850	0.7667	0.0911	0.0947	0.1010	0.1033
IS	0.5248	0.5330	0.1476	0.1533	0.1783	0.1813
LU	0.3675	0.4193	0.0535	0.0553	0.0252	0.0271
MG	0.3057	0.3382	0.0500	0.0524	0.0440	0.0461
SP	0.2454	0.2580	0.0937	0.0992	0.0903	0.1006
Fekete	0.1775	0.2233	0.0198	0.0210	0.0100	0.0115
CalcuNetw	0.0178	0.0180	0.0111	0.0113	0.0030	0.0030
DBEM	3.3010	3.5141	0.6193	0.6498	0.7348	0.7624
STEM-II	1.1269	1.1715	0.4711	0.4885	0.5468	0.5792



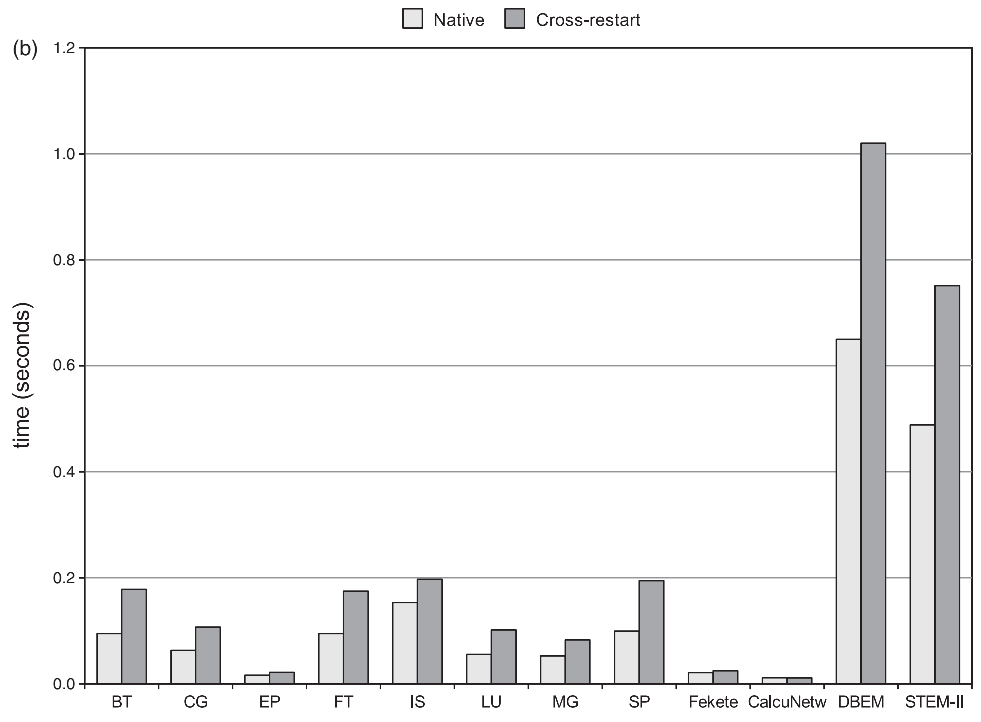


FIGURE 6. Restart times for test applications

In conclusion, these tests show restart times to be very low and fairly negligible in most situations. Restart times never exceed 1 s, except when working with large state files as is the case with the DBEM and STEM-II applications. Even in these cases, most of the restart overhead is introduced by the consistency checks on state files, while the restart protocol itself remains very light and negligible when compared with the total runtime of these applications.

4. RELATED WORK

Although several tools for checkpointing have been proposed in the literature, each one with its own approach, it is often very difficult to draw performance comparisons due to the experimental evaluation of these tools being very weak or to a complete mismatch in the corresponding focuses of the evaluations themselves. Nevertheless, this section tries to address both the qualitative and quantitative (when possible) differences between CPPC and other approaches.

CoCheck [19] is a full checkpointing library for parallel applications based on Condor [35]. It uses a coordinated checkpoint-based approach, with messages requesting checkpoints acting as initiators. Upon reception of a checkpoint request, each process stores the state of its communication buffers and sends another request via all its communication channels. The drawbacks of this approach are its coordinated nature, which hampers its scalability, and the use of full checkpointing, which makes it unable to work on heterogeneous environments and also results in lower efficiency. Besides, CoCheck depends on *tuMPI*, a specific MPI implementation. Regarding its performance, the only concern it addresses is the time it takes for the system to migrate a checkpoint file from one machine to another. It serves no purpose to draw comparisons regarding this parameter, since in the general case it will only depend on the network connecting the source and destination machines and the size of the state files to be transferred.

CLIP [17] is a tool focused on checkpointing parallel applications written for Intel Paragon architectures. CLIP implements efficient and simple solutions. Portability is not a design goal, and its architecture is completely Paragon-centric. It implements a blocking coordinated approach and performs full checkpointing, but allows for dead memory regions to be excluded from the checkpoint file, which improves efficiency. It requires the user to specify checkpoint locations in the application code. Some interesting details are given in the experimental evaluation of CLIP. Tests are executed on an Intel Paragon machine and use 128 execution nodes. Quantitative comparisons can be drawn from its use of the NPB-MPI LU application. Although the actual version used when evaluating CLIP was NPB-MPI v2.1, the benchmarks did not include significative changes between the two versions of the code. For this application, the memory exclusion techniques used by CLIP achieve no visible reduction, while CPPC's automatic live variable analysis achieves a 51.81% reduction (Table 5). The main difference between CPPC and CLIP is that CPPC only stores live variables into checkpoint files, while CLIP performs full checkpointing excluding dead variables. In this way, significant amounts of extra non-portable information, such as buffered messages or stack data, are included into CLIP-generated checkpoint files. However, not having access to the codes used for these experimental tests, no conclusive reasons for this difference in file sizes can be given. Regarding the overhead per checkpoint taken, CLIP presents an overhead of 27.5 s per checkpoint, approximately a 0.69% of the total execution time. CPPC creates a CRC-32 checkpoint for this application in ~ 0.12 s (Table 6), or a 0.01% of the execution time of the application. The total overhead introduced by CPPC in the whole execution is 0.59% (Table 7), which is less than the overhead for only one checkpoint file creation in CLIP (not taking into account the instrumentation overhead). When comparing the overheads of both CLIP and CPPC, it has to be taken into account that there is a 10-year technological gap between both.

Porch [36] is a source-to-source compiler that translates sequential C programs into semantically equivalent C programs that are capable of saving to and recovering from portable checkpoints. The user inserts a call to a checkpoint routine and specifies the desired checkpointing frequency. A compiler then makes source-to-source transformations to instrument its operation. Although this is a tool for sequential applications, it is of fundamental importance in the context of this paper, since it introduces basic ideas and techniques for checkpoint portability. The performance of Porch was evaluated in [37]. Its instrumentation overhead is generally low, although it can get as high as 301% when checkpointing small kernels with many recursive calls. The overheads for checkpointing itself are of $\sim$1% as long as the checkpoint file sizes are kept low, but get significantly higher for medium- and large-scale applications.

In Starfish [38], each MPI node runs a separate Starfish daemon. Its modular system enables the use of different checkpoint-based protocols, namely uncoordinated and coordinated forms of checkpointing. Since the checkpointing mechanisms are implemented on top of the OCaml virtual machine, and given that OCaml is platform-independent, it is able to operate in heterogeneous clusters. However, the use of OCaml is also the biggest drawback of the approach performance-wise. This is aggravated by the creation of full checkpoints.

MPICH-V2 [39] is a communications driver for MPICH. It provides transparent fault tolerance by using the Condor checkpoint and restart capabilities. As such, it uses full checkpointing with no support for portability or heterogeneous environments. It uses a sender-based pessimistic logging approach. Although checkpointing using message logging is an efficient operation, since it involves no process coordination, the scalability is hindered by the log-based approach, which affects the latency of communications causing severe performance degradation for applications with frequent, short-message communications. Performing full checkpointing helps neither efficiency nor portability. The fact that it is built as an MPICH driver forces all machines to implement it in order to obtain fault tolerance. The advantage of message logging is that only failed processes are required to roll back, which improves the recovery performance when compared with coordinated approaches.

In the C^3 system [40], the user must insert checkpoint locations and a compiler is in charge of orchestrating fault tolerance at the variable level, but storing all the variables in the code (i.e. it does not perform liveness analysis). It is implemented on top of MPI and therefore, like CPPC, does not need any specific implementation to be used in order to benefit from fault tolerance. It uses a non-blocking coordination protocol piggybacking information into sent messages: a heavier, modified version of the Chandy–Lamport algorithm which is capable of dealing with out-of-order messages. Reported evaluation results (the tool is not available) show performance degradation for applications generating large state files or with intensive communications.

More recently, checkpoint/restart support has been added to the Open MPI project [9, 10] through a coordinated approach supported primarily by the Berkeley Lab's Checkpoint/Restart (BLCR) Library [41]. Although this is a system-level approach, and therefore the portability is not among their characteristics, a comparison with CPPC in terms of overhead and checkpoint sizes is presented in the following section.

4.1. CPPC vs Open MPI fault tolerance

The current version of BLCR does not support Itanium architectures, which is another limiting factor for the fault tolerance support in Open MPI. For this reason, the checkpoint/restart functionality of Open MPI could not be tested on the Finis Terrae supercomputer. The NM cluster described in Section 3.2 was used instead. The underlying storage system is mounted via NFS through the Gigabit Ethernet administration network, and is composed of four 7.200 RPM hard drives. The NPB-MPI v3.1 applications were used for these comparison tests using the same problem sizes as for the performance analysis in the Finis Terrae supercomputer.

The overhead introduced by both tools was studied for each of the NPB-MPI applications. The results were obtained working with 32 processes, except for BT and SP that run on 36. Tables 10 and 11 show the execution times and overheads in seconds for CPPC and Open MPI, respectively. The NFT columns represent an execution without fault tolerance support. The FT column shows the results with fault tolerance support for a failure-free execution; one checkpoint file is created during the execution of all the applications. If a failure occurs, the restart time overhead must be taken into account in the global execution time. The FT + *R* column presents the results for an execution with one failure; one checkpoint file is generated before crash and the applications are restarted from this checkpoint file. The restart overhead was not measured independently, unlike in Section 3.2.4, due to the different checkpointing approaches in CPPC and BLCR. For this reason, to ensure accurate comparisons, it is preferable to measure the faulty execution and its corresponding retry as a whole. This cluster does not exhibit the high loads nor the high variability experienced in the Finis Terrae, and thus the minimum of 10 executions of each version of the codes are shown in the tables.

TABLE 10. CPPC runtime overheads (s) for the NPB-MPI applications on the NM cluster.

	NFT	FT		FT + R	
NPB	Time	Time	Overhead	Time	Overhead
BT	59.11	59.83	0.72	79.21	20.10
CG	27.38	27.80	0.42	48.63	21.25
EP	19.34	19.33	0.01	23.30	3.96
FT	13.81	33.29	19.48	44.56	30.75
IS	3.77	53.81	50.04	63.40	59.63
LU	61.68	61.56	0.12	69.87	8.19
MG	4.33	12.58	8.25	18.85	14.52
SP	62.22	62.39	0.17	80.90	18.68

TABLE 11. Open MPI runtime overheads (s) for the NPB-MPI applications on the NM cluster.

	NFT	FT		FT + R	
NPB	Time	Time	Overhead	Time	Overhead
BT	57.20	64.26	7.06	90.47	33.27
CG	27.37	53.99	26.62	77.51	50.14
EP	18.83	18.36	0.47	24.92	6.09
FT	14.00	74.98	60.98	101.21	87.21
IS	5.09	64.26	59.17	78.96	73.87
LU	60.06	89.32	29.26	104.26	44.20
MG	3.94	21.02	17.08	23.11	19.17
SP	60.85	75.18	14.33	102.38	41.53

The times shown in Table 10 are quantitatively and qualitatively different from the ones previously shown for the Finis Terrae supercomputer. Running times for all applications are drastically reduced due to the difference in raw performance by the nodes in each system (which is much better on NM). The checkpointing overhead experiments a large relative growth for the shorter applications, because of two different facts: first, the storage system in the NM cluster does not achieve the performance of the Finis Terrae high-performance storage system; and second, because of the shorter runs multithreaded checkpointing is not able to completely mask the file creation overheads. In this situation, the execution is not completed until the state files have been created, and the file creation overhead is transferred directly to the running time of the applications.

Regarding direct comparisons between both tools, the overhead of CPPC is always lesser than the one introduced by Open MPI (the improvement being in the 15–99% range). This is in part due to the coordination protocol used by CPPC and in part due to the smaller size of the checkpoint files. Table 12 shows the sizes of the checkpoint files generated for each application. This table aggregates the sizes for processes executing an application

TABLE 12. Aggregate sizes (in MB) of the checkpointing files generated by CPPC and Open MPI.

NPB	CPPC	Open MPI
BT	623.83	752.92
CG	633.38	1205.90
EP	33.51	147.41
FT	1284.76	2033.32
IS	2309.88	1774.64
LU	264.88	390.20
MG	500.89	461.57
SP	711.74	752.85

in order to easily display the total differences between the two checkpointing tools. It could be expected that CPPC checkpoint files were always smaller than the ones generated by Open MPI, given that CPPC follows a variable level approach with a live variable analysis included in the compiler. However, this is not the case for the MG and IS benchmarks. The reason is that BLCR optimizes state saving by omitting *zero pages*, i.e. those that contain only 0s. This optimization happens to be especially useful for some applications, and thus its implementation is an ongoing work to be incorporated into CPPC.

5. DISCUSSION AND CONCLUDING REMARKS

In the last decade, there has been a fair amount of research directed toward adapting sequential checkpointing techniques to develop tools able to work in parallel environments. More recently, some efforts have focused on checkpointing for heterogeneous environments. However, most of this research has produced theoretical approaches that were ultimately implemented as academic prototypes that were never made available to the general public. Moreover, the experimental validation of most of these approaches has shown scalability limitations imposed by the solutions adopted for interprocess consistency.

CPPC has been designed to improve on the traditional bottlenecks of checkpointing for message-passing applications. The first and foremost are the performance and scalability issues derived from the coordination of processes. The approaches in the literature solve this problem through process coordination and message logging, two well-established techniques that are applicable to any message-passing application. The approach taken in CPPC requires SPMD codes containing safe points at the critical computation locations. It then solves interprocess consistency by statically placing checkpoints at detected safe points. The experiments have shown that, when applicable, the spatially coordinated approach solves the scalability bottleneck created by the consistency requirements of the checkpointing of message-passing applications. Thus, the improvement over other approaches is significant when using an increasing number of processors. The experimental validation performed in this paper has also shown that CPPC's approach is valid for a wide range of applications.

Another factor impacting checkpointing performance is the size of the generated state files. As shown in the related work, the memory size of applications is a limiting factor in the efficiency of checkpointing. CPPC improves this situation by carefully selecting those parts of application runtime memory that are needed when recovering the application state. Moreover, instead of taking non-portable solutions for storing opaque state, these concerns are transferred to restart time and solved via code re-execution. In this way, state files contain only variables and recovery-related metadata, achieving an important reduction of state file sizes in relation to application memory footprints. Our current efforts focus on the reduction of checkpoint file sizes. As mentioned in the previous section, checkpoint files could be further reduced by not storing zero pages into state files in the same way as BLCR. Another technique to reduce file sizes is incremental checkpointing, proposed by Plank *et al.* [42] and recently addressed in different works [43, 44]. Incremental checkpointing involves using the operating system's page protection mechanisms to detect which pages have changed since the last checkpoint, saving only those. During a restart, the state is restored by using the first checkpoint file, and applying in an orderly manner all the differences before resuming the execution. Although incremental checkpointing techniques have been typically applied to system-level approaches, a similar idea could also be incorporated to the variable level approach used by CPPC.

Both the feasibility and the performance of the CPPC framework have been tested using a large number of very different applications. Experimental tests have shown adequate automatic processing by the compiler in all cases. The performance of the compilation process has been shown to be acceptable. We are currently undertaking the implementation of the compiler analyses on top of the plugin interface available since GCC v4.5. This will improve both their performance and usability.

Statistical techniques were employed for the accurate estimation of all the parameters measured. The majority of the experiments shown in this paper were executed on a supercomputer in operation. This represents a typical machine with a realistic load, and demonstrates that the framework can be used in a computing infrastructure on which the user has no superuser access or administration privileges.

Experiments have been conducted to establish the mathematical relationship between state file sizes and checkpoint times in CPPC. When not using compression, linear models explain up to 99% of the variability of the checkpoint file creation times. In order to reduce the overhead introduced by the file creation step, CPPC uses a multithreaded dumping algorithm. The global overhead introduced in the test applications using this approach was also determined. The results have shown very low checkpoint overheads, even when the presented numbers were the worst case ones and a better behavior is to be expected in the majority of the executions. The overhead of the restart process has been analyzed and shown to be negligible in reasonable execution scenarios (i.e. with standard failure rates).

Regarding checkpoint file portability, the CPPC application-level approach constitutes a major advantage over system-level checkpointing approaches, which are very sensitive to the architecture and operating systems. The platform independence feature of CPPC has been assessed by running it on different systems, regardless of the hardware/software configuration. This paper has shown cross-restart results obtained using the Finis Terrae supercomputer and a local cluster, demonstrating low overheads for data conversions and the portability of the generated state files.

Summarizing, the checkpointing techniques evaluated in this paper improve on the two most important overhead factors in other checkpointing approaches: state file sizes and process coordination. Sections 3 and 4 have shown that this approach achieves considerable performance gains, while maintaining the transparency of less efficient solutions and adding the portability of both the tool and the generated checkpoint files.

To our knowledge, CPPC is the only publicly available portable checkpointer for message-passing applications. CPPC is an open-source project, available at http://cppc.des.udc.es under GPL license.

ACKNOWLEDGEMENTS

We gratefully thank CESGA (Supercomputing Center of Galicia, Santiago de Compostela, Spain) for providing access to the Finis Terrae computer.

FUNDING

This research was supported by the Ministry of Science and Innovation of Spain and FEDER funds of the European Union (Project TIN2010-16735).

REFERENCES

[1] Gibson, G., Schroeder, B. and Digney, J. (2007) Failure tolerance in petascale computers. *CTWatch Quart.*, **3**, 4–10.

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

[3] Gelenbe, E. and Derochette, D. (1978) Performance of rollback recovery systems under intermittent failures. *Commun. ACM*, **21**, 493–499.

[4] Gelenbe, E., Tripathi, S. and Finkel, D. (1986) On the availability of a distributed computer system with failing components. *Acta Inform.*, **23**, 643–655.

[5] Wong, K.F. and Franklin, M. (1996) Checkpointing in distributed systems. *J. Parallel Distrib. Comput.*, **35**, 67–75.

[6] Plank, J.S. and Thomason, M.G. (2001) Processor allocation and checkpoint interval selection in cluster computing systems. *J. Parallel Distrib. Comput.*, **61**, 1570–1590.

[7] Rodríguez, G., Martín, M.J., González, P., Touriño, J. and Doallo, R. (2010) CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications. *Concurrency. Comput. Pract. Exp.*, **22**, 749–766.

[8] Cardoso, M.C. and Costa, F.M. (2010) MPI support on opportunistic grids based on the InteGrade middleware. *Concurr. Comput. Pract. Exp.*, **22**, 343–357.

[9] Hursey, J., Mattox, T.I. and Lumsdaine, A. (2009) Interconnect Agnostic Checkpoint/Restart in Open MPI. *Proc. HPDC 09*, Munich, Germany, June 11–13, pp. 49–58. ACM, New York.

[10] Hursey, J., Squyres, J.M., Mattox, T.I. and Lumsdaine, A. (2007) The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI. *Proc. IPDPS 07*, Long Beach, CA, USA, March 26–30, pp. 415–422. IEEE Computer Society Press, Los Alamitos.

[11] Bosilca, G. *et al.* (2002) MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. *Proc. SC 02*, Baltimore, MD, USA, November 16–22. IEEE Computer Society Press, Los Alamitos.

[12] Buntinas, D., Coti, C., Hérault, T., Lemarinier, P., Pilard, L., Rezmerita, A., Rodriguez, E. and Cappello, F. (2008) Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI protocols. *Futur. Gener. Comput. Syst.*, **24**, 73–84.

[13] Walters, J. and Chaudhary, V. (2006) Application-Level Checkpointing Techniques for Parallel Programs. *Proc. ICDCIT 2006*, Bhubaneswar, India, December 20–23, pp. 221–234. Springer, Berlin.

[14] The HDF Group. HDF-5: File format specification. http://www.hdfgroup.org/HDF5 (last accessed December 2010).

[15] Strumpen, V. (1998) Portable and fault-tolerant software systems. *IEEE Micro*, **18**, 22–32.

[16] Elnozahy, E.N. and Plank, J.S. (2004) Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery. *IEEE Trans. Dependable Secur. Comput.*, **1**, 97–108.

[17] Chen, Y., Plank, J.S. and Li, K. (1997) CLIP: A Checkpointing Tool for Message-Passing Parallel Programs. *Proc. SC 97*, San Jose, CA, USA, November 16–21. IEEE Computer Society Press, Los Alamitos.

[18] Schulz, M., Bronevetsky, G., Fernandes, R., Marques, D., Pingali, K. and Stodghill, P. (2004) Implementation and Evaluation of a Scalable Application-Level Checkpoint-Recovery Scheme for MPI Programs. *Proc. SC 04*, Pittsburgh, PA, USA, November 6–12. IEEE Computer Society Press, Los Alamitos.

[19] Stellner, G. (1996) CoCheck: Checkpointing and Process Migration for MPI. *Proc. IPPS 96*, Honolulu, HI, USA, April 15–19, pp. 526–531. IEEE Computer Society Press, Los Alamitos.

[20] Rodríguez, G., Martín, M.J., González, P. and Touriño, J. (2009) A heuristic approach for the automatic insertion of checkpoints in message-passing codes. *J. Univers. Comput. Sci.*, **15**, 2894–2911.

[21] Rodríguez, G., Martín, M.J., González, P. and Touriño, J. (2006) Controller/precompiler for portable checkpointing. *IEICE Trans. Inf. Syst.*, **E89-D**, 408–417.

[22] Plank, J.S., Beck, M. and Kingsley, G. (1995) Compiler-assisted memory exclusion for fast checkpointing. *IEEE Techn. Committee Oper. Syst. Appl. Environ.*, **7**, 10–14.

[23] Lee, S.I., Johnson, T.A. and Eigenmann, R. (2004) Cetus — An Extensible Compiler Infrastructure for Source-To-Source Transformation. *Proc. LCPC 03*, College Station, TX, USA, October 2–4, pp. 539–553. Springer, Berlin.

[24] National Aeronautics and Space Administration. *The NAS Parallel Benchmarks*. http://www.nas.nasa.gov/Software/NPB. (last accessed December 2010).

[25] CESGA-2005-003 (2005). CalcuNetw: Calculate Measurements in Complex Networks. Supercomputing Center of Galicia, Santiago de Compostela, Spain.

[26] Carmona, A., Encinas, A.M. and Gesto, J.M. (2007) Estimation of Fekete points. *J. Comput. Phys.*, **225**, 2354–2376.

[27] González, P., Pena, T.F. and Cabaleiro, J.C. (2000) Dual BEM for crack growth analysis on distributed-memory multiprocessors. *Adv. Eng. Softw.*, **31**, 921–927.

[28] Martín, M.J., Singh, D.E., Mouriño, J.C., Rivera, F.F., Doallo, R. and Bruguera, J.D. (2003) High performance air pollution modeling for a power plant environment. *Parallel Comput.*, **29**, 1763–1790.

[29] Zandy, V. *CKPT library home page*. http://www.cs.wisc.edu/∼zandy/ckpt (last accessed December 2010).

[30] Tukey, J.W. (1977) *Exploratory Data Analysis*. Addison-Wesley, Reading, MA.

[31] Gelenbe, E. (1979) On the optimum checkpoint interval. *J. ACM*, **26**, 259–270.

[32] Toueg, S. and Babaoğlu, Ö. (1984) On the optimum checkpoint selection problem. *SIAM J. Comput.*, **13**, 630–649.

[33] Vaidya, N.H. (1997) Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Trans. Comput.*, **46**, 942–947.

[34] Chabridon, S. and Gelenbe, E. (1995) Failure Detection Algorithms for a Reliable Execution of Parallel Programs. *Proc. SRDS 95*, Bad Neuenahr, Germany, September 13–15, pp. 229–238. IEEE Computer Society Press, Los Alamitos.

[35] Litzkow, M.J., Livny, M. and Mutka, M. (1988) Condor — A Hunter of Idle Workstations. *Proc. ICDCS 88*, San Jose, CA, USA, June 13–17, pp. 104–111. IEEE Computer Society Press, Los Alamitos.

[36] Ramkumar, B. and Strumpen, V. (1997) Portable Checkpointing for Heterogeneous Architectures. *Proc. FTCS 97*, Seattle, WA, USA, June 24–27, pp. 58–67. IEEE Computer Society Press, Los Alamitos.

[37] ECE-96-6-1 (1996). Portable Checkpointing and Recovery in Heterogeneous Environments. University of Iowa, Iowa City, IA, USA.

[38] Agbaria, A. and Friedman, R. (2003) Starfish: fault-tolerant dynamic MPI programs on clusters of workstations. *Cluster Comput.*, **6**, 227–236.

[39] Bouteiller, A., Capello, F., Hérault, T., Krawezik, G., Lemarinier, P. and Magniette, F. (2003) MPICH-V2: A Fault-Tolerant MPI for Volatile Nodes Based on Pessimistic Sender Based Message Logging. *Proc. SC 03*, Phoenix, AZ, USA, November 15–21. ACM, New York.

[40] Bronevetsky, G., Marques, D., Pingali, K. and Stodghill, P. (2003) Automated Application-Level Checkpointing of MPI Programs. *Proc. PPoPP 03*, San Diego, CA, USA, June 11–13, pp. 84–94. ACM, New York.

[41] LBNL-54941 (2003). The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Lawrence Berkeley National Laboratory, Berkeley, CA, USA.

[42] CS-95-302 (1995). Compressed Differences: An Algorithm for Fast Incremental Checkpointing. University of Tennessee, Knoxville, TN, USA.

[43] Agarwal, S., Garg, R. and Gupta, M.S. (2004) Adaptive Incremental Checkpointing for Massively Parallel Systems. *Proc. ICS 04*, Saint Malo, France, June 26–July 01, pp. 277–286. ACM, New York.

[44] Gioiosa, R., Sancho, J.C., Jiang, S. and Petrini, F. (2005) Transparent, Incremental Checkpointing at Kernel Level: A Foundation for Fault Tolerance for Parallel Computers. *Proc. SC 05*, Seattle, WA, USA, November 12–18. IEEE Computer Society Press, Los Alamitos.

[45] Smale, S. (1998) Mathematical problems for the next century. *Math. Intell.*, **20**, 7–15.