CPPC: A compiler–assisted tool for portable checkpointing of message-passing applications

Gabriel Rodríguez, María J. Martín, Patricia González, Juan Touriño, Ramón Doallo

Computer Architecture Group, Department of Electronics and Systems
University of A Coruña, Spain
{grodriguez,mariam,pglez,juan,doallo}@udc.es

Abstract

With the evolution of high-performance computing towards heterogeneous, massively parallel systems, parallel applications have developed new fault tolerance necessities. Checkpointing has become a widely used technique to obtain fault tolerance. Whether due to a failure in the execution or to a migration of the processes to different machines, checkpointing tools must be able to operate in heterogeneous environments. Portable checkpointers usually work around portability issues at the cost of transparency: the user must provide information as what data needs to be stored, where to store it, or where to checkpoint. CPPC (Controller/Precompiler for Portable Checkpointing) is a checkpointing tool designed to feature both portability and transparency. It is made up of a library containing checkpointing routines and a compiler which automates the use of the library. This paper gives an overview of the CPPC tool. Experimental results using benchmarks and large-scale real applications are included, demonstrating usability, efficiency and portability.

1 Introduction

Current trends towards 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 structures such as application stack, heap, or communication state. 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.

Most approaches [1, 2, 10, 15] perform data segment level checkpointing, that is, they store the entire application state. This leads to a lack of portability, as a number of non-portable structures will be saved along with application data (as application stack or heap).

This paper presents CPPC, a checkpointing framework focused on the automatic insertion of fault tolerance into long-running message-passing applications. It is designed to allow for execution restart on different architectures and/or operating systems, also supporting checkpointing over

heterogeneous systems, such as the Grid. It uses portable code and protocols, and generates portable checkpoint files while avoiding traditional solutions (such as process coordination [3, 17] or message-logging [2]) which add an unscalable overhead.

The structure of the paper is as follows. Section 2 describes CPPC's design and the CPPC Library. Section 3 is devoted to the description of the CPPC Compiler. Section 4 presents experimental results. Finally, Section 5 concludes the paper with the main features and contributions of the tool.

2 The CPPC Framework

As stated in the previous section, current fault tolerance trends require portable tools for message-passing applications, focusing on providing the following fundamental features:

1. OS-independence: checkpointing strategies must be compatible with any given operating system. This means having at least a basic modular structure to allow for substitution of certain critical sections of code (e.g. filesystem access) depending on the underlying OS.

2. Support for parallel applications with communication protocol independence: the checkpointing framework should not make any assumption as to the communication interface or implementation being used. Large-scale machines in computational Grids belong to independent entities which cannot be forced to provide a certain version of the MPI interface. Even recognizing the role of MPI as the message-passing de-facto standard, the checkpointing technique cannot be theoretically tied to MPI in order to provide a truly portable, reusable approach.

3. Reduced checkpoint file sizes: the tool should optimize the amount of data being saved, avoiding dumping state which will not be necessary upon application restart. This improves performance, which depends heavily on state file sizes. It also enhances the performance of application migration between massively parallel systems in case of failure.

4. Portable data recovery: the state of an application can be seen as a structure containing different types of data. The checkpointing tool must be able to recover all these data in a portable way. This includes the recovery of opaque state, such as MPI communicators, as well as of OS-dependent state, such as the file table or the execution stack.

The CPPC framework provides all these features which are key issues for fault-tolerance support on heterogeneous large-scale systems. It appears to the user as a runtime library containing checkpoint-support routines, together with a compiler which automates the use of the library. A preliminary version of the CPPC Library was presented in [16].

In order to flexibly support different host languages, library routines such as variable registration, state dumping, etc. are implemented through delegation into a controller. Interfaces masking programming language features, such as static/dynamic memory management or variable typing are provided to decouple both the application code and the checkpoint system. Currently supported languages are C and Fortran 77.

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; a 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 file storage.

CPPC has two operation modes: *checkpoint operation* and *restart operation*. Checkpoint operation mode is active when a normal execution is being run. It basically consists of marking relevant variables for dumping at next checkpoint (*variable registration*) and dumping state files at the specified locations in the code. Restart operation mode emerges when the application must be restarted from a previously saved checkpoint file. Both portable and non-portable state have to be recovered. Also, when working with parallel applications it must be ensured that all processes are restarted in a consistent global state: no coordination problems may arise at restart time.

The following subsections address the techniques used by CPPC for solving the aforementioned issues.

2.1 Checkpoint file dumping

From the point of view of the data stored in state files, checkpointers can work at segment level, storing the whole application state, or at variable level, storing user variables only. CPPC works at variable level, that is, it stores only those variables which are needed upon application restart. When the execution flow reaches a checkpoint, data to be stored are structured and passed to the writing layer. This includes both static and dynamic data. The actual data writing will be performed by the selected writing plugin. This enables the restart on different architectures, as long as a portable format is used to store the data.

The CPPC framework includes an HDF-5-based [13] writing plugin. HDF-5 is a general purpose library and file format for portably storing scientific data. It supports recovery on binary incompatible machines and/or different operating systems. The CPPC Library design allows for the implementation of new writing plugins that can be attached to the Library without recompiling it.

The state file format used keeps track of function calls performed by the application, allowing for its state to be rebuilt by recreating the sequence of procedure calls made by the original execution. Any pattern in procedure calling may be represented, including recursive calls. The use of portable offsets instead of memory addresses [18] for data representation enables pointer portability, preserving aliasing relationships.

CPPC provides checkpointing options such as multithreaded dumping. If multithreading is active, a checkpoint will first calculate aliasing relationships between variables to be stored, and offsets for each register into a memory block. Then, memory blocks to be stored are copied over to ensure that the checkpointing will work over a clean, unmodified copy. After this process is complete, a new thread is created to handle checkpointing using copied blocks, while the application resumes its execution using the original data. This avoids the overhead caused by waiting for checkpoint files to be written to disk.

If a failure occurred in the checkpointing thread, inconsistent checkpoint files would be created. CPPC generates a CRC-32 for the checkpoint file. This CRC-32 is checked upon restart to ensure file correctness.

CPPC also supports file compression. A scheme based on the ZLib library [6] has been developed, but other algorithms can be included. File compression does not only help save disk space and network transfers (if needed), but can also improve performance when working with large datasets with high compression rates.

2.2 Global consistency

When checkpointing parallel applications, special considerations regarding message-passing have to be taken to ensure that the coordination implicitly established by the communication flow between processes is not lost when restarting the application. If a checkpoint is placed in the code between two matching communication statements, an inconsistency would 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 a *ghost message*.

A global checkpoint is said to be *transitless* if there are no in-transit messages when it is created [4]. It is called *consistent* if there are no ghost messages [8]. Traditional approaches use some kind of process coordination to ensure consistency, and message-logging techniques to solve issues generated by in-transit messages. As stated in [5], the main drawback of coordinated checkpointing is potential lack of scalability, since it may force all processes to take a checkpoint concurrently; the drawback of message-logging is lack of efficiency derived from the overhead of log requirements. CPPC avoids these issues by focusing on SPMD parallel applications and using a spatially coordinated approach. Checkpoints are taken at the same relative code locations by all processes, without performing interprocess communications or runtime synchronization. To avoid problems caused by messages between processes, checkpoints must be inserted at points where it is guaranteed that there are no in-transit, nor ghost messages. These points will be called *safe points*. Safe point identification and checkpoint insertion is automatically performed by the compiler. Generated checkpoints are transitless and consistent, both being conditions for a checkpoint to be called *strongly consistent* [8].

This protocol achieves to improve efficiency and scalability by transferring consistency concerns from runtime to both compile and restart time: at compile time, safe points are detected, and at restart time a negotiation is performed to achieve an agreement about the checkpoint files to be used for application restart. Both compiling and restarting an application are expected to be far less frequent operations than checkpoint file generation.

2.3 Restart protocol

When restarting an application, not only user variables must be recovered, but also non-portable state created in the original execution, such as MPI communicators, virtual topologies, or derived data types. However, CPPC only stores portable data into state files. This introduces the need for a restart protocol to regenerate the original state that has not been stored. CPPC uses code re-execution to achieve complete application state recovery. Through code re-execution, non-portable state is recovered by the same means used originally to create it. Therefore, a CPPC application is just as portable as the original one: variables are saved in a portable manner, non-portable state is

recreated using the original code.

A piece of application code is defined as *Required-Execution Code* (REC) if it must be re-executed at restart time to ensure correct state recreation. The recovery process consists of the ordered re-execution of such blocks of code. Examples of RECs are the call to the CPPC initialization function; the execution of variable registration calls, responsible for recovering variable values when restarting; or the execution of procedures with non-portable outcome. REC detection is completely automated by the CPPC compiler.

CPPC controls execution flow when restarting an application, making it jump from the end of one REC to the beginning of the next one, and skipping non-relevant code. The result is an ordered execution of state-recovering statements which, eventually, creates a replica of the original application state. Once all state has been recovered and the checkpoint statement where the state file used for recovery was created is reached, the execution changes to checkpoint mode and proceeds normally.

3 CPPC Compiler

The CPPC Compiler is built on the Cetus compiler infrastructure [9]. It is written in Java, which makes its code inherently portable. Although Cetus was originally designed to support C codes, we have extended it to allow for parsing Fortran 77 codes. It uses the same basic intermediate representation language (IRL) for both C and Fortran codes, hence allowing the same transformation code to be applied to applications written in both languages.

Some transformations performed by the compiler require knowledge about which specific procedures implement certain semantics (e.g. which procedure initializes the parallel system). We call these transformations *semantic-directed*. Descriptive files included in the CPPC distribution are used for supplying such knowledge. Each of these files is called a *semantic module*, and contains information about a set of related procedures (e.g. MPI functions). It marks functions as implementors of certain roles (e.g. a sender function) and also contains information on how the role is implemented (e.g. which parameter of the send function specifies the recipient's rank, and which one specifies the message tag). This information will be used by the compiler when applying semantic-directed transformations (e.g. communication matching).

Besides semantic information, these modules also contain data flow information. Function parameters are categorized as input, output, or input-output, which gives details about the outcome, in terms of data flow, of a call to a certain procedure. This prevents data flow analyses from having to use conservative approaches when analyzing calls to procedures in external libraries, which usually involve executing code not accessible at compile time.

The use of semantic modules enables transformations to work with different programming interfaces for a given subsystem: extensions to other communication or file I/O interfaces would only require the creation of the corresponding descriptive semantic module. This makes the CPPC Compiler a portable, versatile and easily extensible tool. Note that semantic modules are not intended to be written by CPPC users, but provided along with the tool.

The most important analyses and transformations performed by the compiler are covered in the following subsections.

3.1 Detection of procedures with non-portable outcome

Information about procedures which need to be re-executed when restarting an application is included in the semantic modules supplied to the compiler. Upon discovery of a non-portable call, the CPPC Compiler adds instrumentation code which ensures that the call will be made with exactly the same parameter values as the original one. Moreover, the application stack of the original execution is also reconstructed, guaranteeing that all generated non-portable state belongs to its proper scope.

3.2 Checkpoint insertion

In order to discover good locations for checkpointing, the compiler performs two analyses. The first one finds safe points in the code by matching communications. It performs constant propagation and symbolic expression analysis to identify literal values for variables used as source, destination, and tags in communication statements. These are called *communication relevant variables*. Then it uses this information to match communications in an interprocedural way. In order to optimize the process, constant propagation and symbolic expression analysis are only performed for statements which modify communication relevant variables.

The second analysis identifies the most computationally expensive loops and chooses safe points inside them for inserting checkpoints. Also, checkpoints are placed at strategic locations in which the amount of live variables is at a local minimum, minimizing the size of the state file as well.

3.3 Registration of restart-relevant variables

In order to identify the variables needed upon application restart, the compiler performs a live variable analysis. This is a somehow complementary approach to memory exclusion techniques used in sequential checkpointers to reduce the amount of memory stored, such as the one proposed in [14]. Before each checkpoint statement c_i, the compiler inserts register functions to mark the variables that must be stored in the checkpoint file, which are those contained in the set of live variables before c_i, $LV_{in}(c_i)$. The data type for the register is determined by checking the variable definition. Variables registered or defined at previous checkpoints are not registered again. Also, before each checkpoint c_i, the compiler inserts unregistration calls for the variables in the set $LV_{in}(c_{i-1}) - LV_{in}(c_i)$, which contains variables that are no longer relevant.

The compiler does not currently perform optimal bounds checks for pointer and array variables. This means that some arrays and pointers are registered in a conservative way: they are entirely stored if they are used at any point in the re-executed code.

When dealing with calls to precompiled procedures located in external libraries, the default behavior is to assume all parameters to be of input type, therefore registering them all. To avoid this default behavior, the CPPC Compiler uses the data flow information available in semantic modules.

Table 1: Test applications

		Files	LOCs	Description	Compile time (s)	Registers
NPB	BT (class=B)	23	4066	Block Tridiagonal	34.58	193
	LU (class=B)	29	3509	LU Symmetric Gauss-Seidel	21.48	89
	CG (class=C)	1	1044	Conjugate Gradient	6.92	37
	MG (class=B)	2	1618	MultiGrid	15.50	34
	IS (class=B)	1	671	Integer Sort	7.36	28
Real	DBEM	45	13130	Crack Growth Simulation	51.28	180
	STEM-II	141	7524	Air Quality Simulation	31.46	156

4 Experimental results

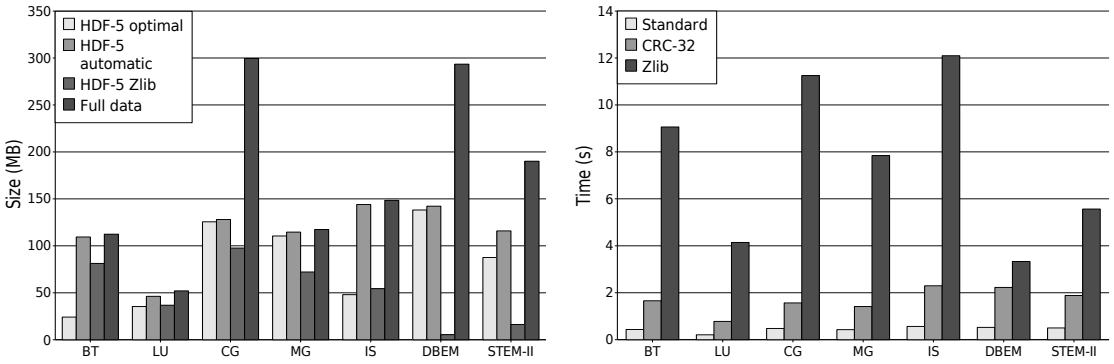
For testing purposes, a twofold approach has been selected. First, five applications contained in the NAS Parallel Benchmarks (NPB) [12] have been used. These applications have short execution times (below 11 minutes), so they are not appealing options for applying checkpointing techniques in practice. For this reason, a crack growth simulation code (named DBEM) [7], as well as an air quality simulation application (named STEM-II) [11], have also been tested. A summary of these applications, along with their sizes in terms of number of files and lines of code (LOCs), the time needed by the CPPC Compiler to instrument them, and the number of variables automatically registered, can be seen in Table 1. Note that there are more than a hundred relevant variables for the large-scale applications, making the automatization of the variable detection and register insertion critical for the usability of the tool.

Tests were performed on a cluster of Intel Xeon 1.8 Ghz nodes, 1 GB of RAM, connected through an SCI network. Size of generated state files, time for state file generation, checkpoint overhead and restart times were measured. For demonstrating portability, these applications were also run on an HP Superdome (Intel Itanium 2 at 1.5 Ghz, 3 GB of RAM, connected through Infiniband) with proprietary C/Fortran compilers and also proprietary MPI implementation. Checkpoint files created in the Superdome were used to restart the applications on the cluster, which allowed for comparing restart times using both native and imported files.

In order to perform a complete evaluation of the tool, it would be desirable to compare these results against others obtained using similar tools. This is not feasible, since there are no such tools in the public domain that we can use in our execution environment. However, some qualitative comparisons are outlined according to experimental results in the literature.

4.1 State file sizes

When using spatially coordinated, non-logging checkpointing techniques, the incurred overhead will only depend on the overhead introduced by the checkpoint file dumping. This overhead heavily depends on the size of the data to be dumped. Thus, the first parameter to be measured is how the variable level approach affects checkpoint file sizes. Results for the test applications are shown in Fig. 1(a). The values tagged as “HDF-5 automatic” are file sizes obtained by the automatic analyses included in the compiler. “HDF-5 optimal” shows the optimal sizes, obtained



(a) Generated file sizes per process

(b) File generation overhead

Figure 1: File sizes and generation overhead

by a manual analysis. The HDF-5 writer can, optionally, compress data using a ZLib plugin. The "HDF-5 Zlib" results have been generated using the automatic variable registration. Sizes obtained for a checkpointer that stores all the application data are included for comparison purposes, and tagged as "Full data". Most checkpointers in the literature belong to this last category. Others use incremental checkpointing, which achieves results roughly similar to the optimal ones shown in the figure. In the future, we plan to include incremental checkpointing on top of the live variable analysis in order to further reduce file sizes.

Sizes obtained using automatic analyses are very close to the optimal ones, except for BT and IS applications. This is due to the fact that unnecessary array sections are registered because of the conservative approach of the compiler, as explained in Section 3. As can be seen, variable level checkpointing achieves very important size reductions for some applications when compared to full data sizes, like in CG where this reduction reaches a 57.26%. For the large-scale applications the reduction is of 51.54% for DBEM, and 39.03% for STEM-II.

The high compression rates obtained for DBEM and STEM-II (96.2% and 85.92%, respectively) are due to the fact that these applications statically allocate arrays which are oversized to fit a maximum problem size. As a result, an important amount of empty memory is allocated, which results in high compression rates.

4.2 State file creation time

The performance obtained by CPPC is tightly tied to the size of the generated files. The other factor that plays a key role in dumping time is the writing strategy used, as shown in Fig. 1(b). Times were taken for a standard HDF-5 file creation, for the same HDF-5 file including an error detection scheme (CRC-32), and for the HDF-5 file using ZLib compression. Note that these times correspond to the raw dumping of a single checkpoint, not the real contribution of dumping times to the checkpoint overhead, which can be reduced by using multithreaded dumping.

Table 2: Multithreaded checkpoint overheads

		Original runtime	Checkpoint overhead (#checkpoints)	Overhead percentage
NPB	BT	638.37 s	2.70 s (1)	0.42%
	LU	467.08 s	4.70 s (1)	1.01%
	CG	617.28 s	7.27 s (1)	1.18%
	MG	21.47 s	2.05 s (1)	9.55%
	IS	5.72 s	1.50 s (1)	26.22%
Real	DBEM	80473.13 s	256.02 s (23)	0.31%
	STEM-II	21622.41 s	101.14 s (7)	0.47%

Written data are tagged by the HDF-5 library to allow for conversions, if needed, when restarting the application. This improves checkpoint mode performance, moving conversion overhead to the restart mode, which is a much less frequent operation.

Using compression heavily increases overall dumping time. Therefore, it should be enabled only when the physical size of the state files is critical; e.g. if there are problems with disk quotas or when the files are going to be transferred using a slow network.

4.3 Checkpoint overhead

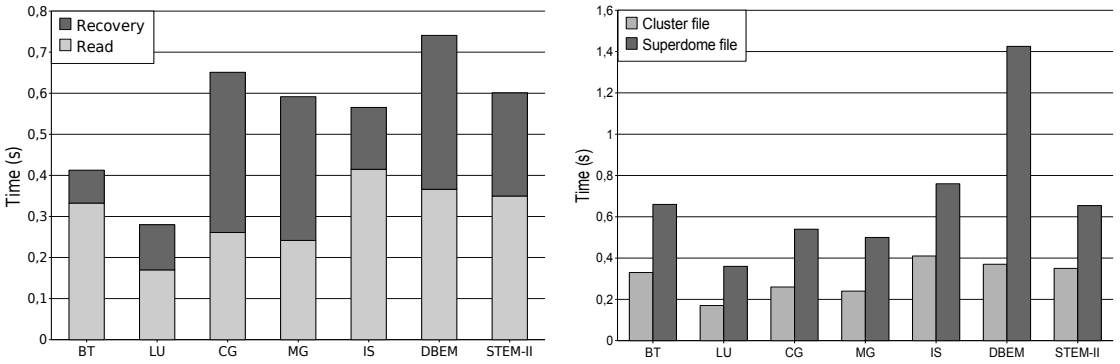
To reduce the overhead introduced by file generation, multithreaded state dumping has been implemented. Table 2 details the original execution times and the overhead introduced by checkpointing. One state file was generated for the NPB benchmarks, while checkpointing frequency was manually adjusted to create approximately one per hour for DBEM and STEM-II. Files were generated using the HDF-5 writer, with the automatic registration process and the standard writing strategy. Note that checkpoint overhead includes, besides file generation, the remaining CPPC operations: initialization, register management, context tracing and finalization. Increasing the checkpointing frequency up to one checkpoint each ten minutes for the large-scale applications did not noticeably vary total execution times, being additional overheads obtained less than 0.01%. Once the instrumentation overhead is introduced, the multithreaded technique is able to absorb the overhead of the data dumping step.

As can be seen, MG and IS have such a short execution time that checkpoint overhead is relatively high (although this overhead is in the range of only seconds). However, those applications with higher runtimes present very low overheads: only 0.31% and 0.47% for DBEM and STEM-II, with runtimes of more than 22 and 6 hours, respectively.

Approaches that solve consistency issues by means of message-logging or process coordination usually present an instrumentation overhead higher than the total overhead incurred by CPPC.

4.4 Restart overhead

If a failure occurs, restart time overhead must be taken into account in the global execution time. Restart times for the applications have been measured and split into its two fundamental phases:



(a) Restart times using HDF-5 files

(b) Comparison of file read times

Figure 2: Restart times for test applications

file read and effective data recovery. Results are depicted in Fig. 2(a). As can be seen, restart times are low (less than one second).

File read time measurement comprises all steps taken until the checkpoint data are loaded into memory and made available for the application to recover them. This process includes identification of the writing plugin to be used, file opening and data reading. Times obtained depend on both file size and whether or not data transformations are needed. This effect is shown in Fig. 2(b), which details file read times for both cluster- and Superdome-generated files when restarting the test applications on the cluster. This test also serves to demonstrate portability.

Recovery time begins when the file read ends, and stops when CPPC determines that the restart process has ended, switching to checkpoint operation mode. This happens when all state is recovered and the execution flow reaches the checkpoint statement where the file was originally generated. This time depends on the amount of state saved and the amount of state recovered using code re-execution.

5 Concluding remarks

CPPC is a portable checkpointing infrastructure for parallel applications. It uses a variable level, non-logging, modular approach to achieve scalability, efficiency and portability. The analyses and transformations performed by the compiler completely automate the instrumentation process. The live variable analysis, which would be a hard task for the user, shows potential for a significant reduction of checkpoint file sizes. The two most remarkable contributions of this framework address consistency issues and portable state recovery.

Consistency issues are not solved at runtime, but rather at both compile and restart time. At compile time, checkpoints are placed in safe points. At restart time, a negotiation between processes decides the safe point from which to restart the application. Process synchronization required by traditional coordinated checkpointing approaches is transferred to the restart operation. This ap-

proach greatly increases scalability, since communications are not necessary during a normal run, and all processes are allowed to operate in a completely independent way.

Portable state recovery is achieved by means of both the recovery of stored data and the re-execution of procedures with non-portable outcome. This re-execution also provides scope for the checkpointing of applications linked to external libraries.

CPPC has been thoroughly tested to demonstrate the proper behavior of its features. It correctly performed application restart for all the test cases, even using the same set of checkpoint files to restart on binary incompatible machines, and different C/Fortran compilers and MPI implementations.

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`. It can be downloaded under GPL license.

Acknowledgment

This research was supported by the Ministry of Education and Science of Spain and FEDER funds of the European Union (Project TIN-2004-07797-C02 and FPU grant AP-2004-2685) and by the Galician Government (Projects PGIDIT04TIC105004PR and PGIDIT05PXIC10504PN). We gratefully thank CESGA (Galician Supercomputing Center) for providing access to the HP Superdome computer.

References

[1] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, pages 167–176, 1999.

[2] A. Bouteiller, F. Capello, T. Hérault, G. Krawezik, P. Lemarinier, and F. Magniette. MPICH-V2: A fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (SC'03)*, 2003.

[3] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. C^3: A system for automating application-level checkpointing of MPI programs. In *Proceedings of LCPC'03*, pages 357–373, 2003.

[4] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[5] E. N. Elnozahy and J. S. Plank. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing*, 1(2):97–108, 2004.

[6] J. Gailly and M. Adler. ZLib Home Page. http://www.gzip.org/zlib/.

[7] P. González, T. F. Pena, and J. C. Cabaleiro. Dual BEM for crack growth analysis on distributed-memory multiprocessors. *Advances in Engineering Software*, 31(12):921–927, 2000.

[8] J.-M. Hélary, R. H. B. Netzer, and M. Raynal. Consistency issues in distributed checkpoints. *IEEE Transactions on Software Engineering*, 25(2):274–281, 1999.

[9] S.-I. Lee, T. A. Johnson, and R. Eigenmann. Cetus – an extensible compiler infrastructure for source-to-source transformation. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, pages 539–553, 2003.

[10] S. Louca, N. Neophytou, A. Lachanas, and P. Evripidou. MPI-FT: Portable fault tolerance scheme for MPI. *Parallel Processing Letters*, 10(4):371–382, 2000.

[11] M. J. Martín, D. E. Singh, J. C. Mouriño, F. F. Rivera, R. Doallo, and J. D. Bruguera. High performance air pollution modeling for a power plant environment. *Parallel Computing*, 29(11–12):1763–1790, 2003.

[12] National Aeronautics and Space Administration. The NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/.

[13] National Center for Supercomputing Applications. HDF-5: File Format Specification. http://hdf.ncsa.uiuc.edu/HDF5/doc/.

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

[15] S. Rao, L. Alvisi, and H. Vin. Egida: An extensible toolkit for low-overhead fault tolerance. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing (FTCS'99)*, pages 48–55, 1999.

[16] G. Rodríguez, M. J. Martín, P. González, and J. Touriño. Controller/Precompiler for Portable Checkpointing. *IEICE Transactions on Information and Systems*, E89-D(2):408–417, 2006.

[17] G. Stellner. Cocheck: Checkpointing and process migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS'96)*, pages 526–531, 1996.

[18] V. Strumpen. Portable and fault-tolerant software systems. *IEEE Micro*, 18(5):22–32, 1998.