CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2000; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

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

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

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

SUMMARY

With the evolution of high-performance computing towards heterogeneous, massively parallel systems, parallel applications have developed new checkpoint and restart necessities. Whether due to a failure in the execution or to a migration of the application processes to different machines, checkpointing tools must be able to operate in heterogeneous environments. However, some of the data manipulated by a parallel application are not truly portable. Examples of these include opaque state (e.g. data structures for communications support) or diversity of interfaces for a single feature (e.g. communications, I/O). Directly manipulating the underlying ad-hoc representations renders checkpointing tools unable to work on different environments. Portable checkpointers usually work around portability issues at the cost of transparency: the user must provide information such as what data needs to be stored, where to store it, or where to checkpoint. CPPC (ComPiler for Portable Checkpointing) is a checkpointing tool designed to feature both portability and transparency. It is made up of a library and a compiler. The CPPC library contains routines for variable level checkpointing, using portable code and protocols. The CPPC compiler helps to achieve transparency by relieving the user from time-consuming tasks, such as data flow and communications analyses and adding instrumentation code. This paper covers both the operation of the CPPC library and its compiler support. Experimental results using benchmarks and large-scale real applications are included, demonstrating usability, efficiency and portability.

KEY WORDS: Fault tolerance; checkpointing; parallel programming; message-passing; MPI; compiler support

This research was supported by the Ministry of Education and Science of Spain and FEDER funds of the European Union (Project TIN2007-67537-C03 and FPU grant AP-2004-2685) and by the Galician Government (Project PGIDIT05PXIC10504PN). We gratefully thank CESGA (Galician Supercomputing Center) for providing access to the HP Superdome computer.

1. INTRODUCTION

As parallel machines increase their number of processors, so does the failure rate of the global system. This is not a problem while the mean time to complete an application's execution remains well under the mean time to failure (MTTF) of the underlying hardware, but that is not always true for applications with large execution times. Under these circumstances, users and programmers need a way to ensure that not all computation done is lost on machine failures.

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 state. A number of solutions and techniques have been proposed [3], each having its own pros and cons.

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.

Therefore, 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.

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 or message logging) 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 the experimental results. Section 5 covers related work, and finally Section 6 concludes the paper.

2. THE CPPC FRAMEWORK

Modern computing trends require portable checkpointing tools for message-passing applications, focusing on providing the following fundamental features:

1. OS-independence: checkpointing strategies must work with any given operating system.
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. Computational Grids include machines belonging 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.

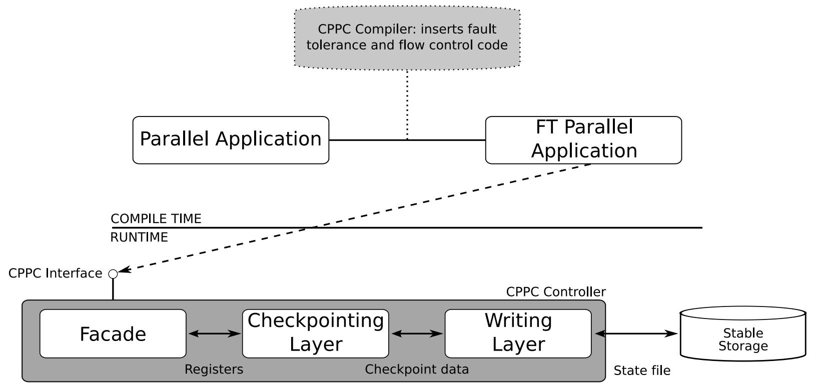


Figure 1. CPPC framework design

3. Reduced checkpoint file sizes: the tool should optimize the amount of data being saved. This improves checkpointing performance, which depends heavily on state file sizes. It also enhances performance of the application migration between massively parallel systems.
4. Portable data recovery: the checkpointing tool must be able to recover data in a portable way. This includes opaque state, such as MPI communicators, as well as OS-dependent state.

The CPPC framework provides all these key features. It consists of a runtime library containing checkpoint-support routines, together with a compiler which automates the use of the library. In early stages of the work, the user was responsible for inserting compiler directives to guide the operation of the runtime library [16]. Currently, all analyses and code transformations are transparently applied by the CPPC compiler, further described in Section 3. The global process is depicted in Figure 1. Library routines are implemented through delegation into a controller, which is written in C++ and does not depend on the specific host language. Interfaces masking programming language features, such as static/dynamic memory management, decouple the application code from the checkpoint system. Current 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; the checkpointing layer, which gathers and manages all the data; 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 used when a normal execution is being run. It consists of marking relevant variables (*variable registration*) and saving them into state files at checkpoints 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

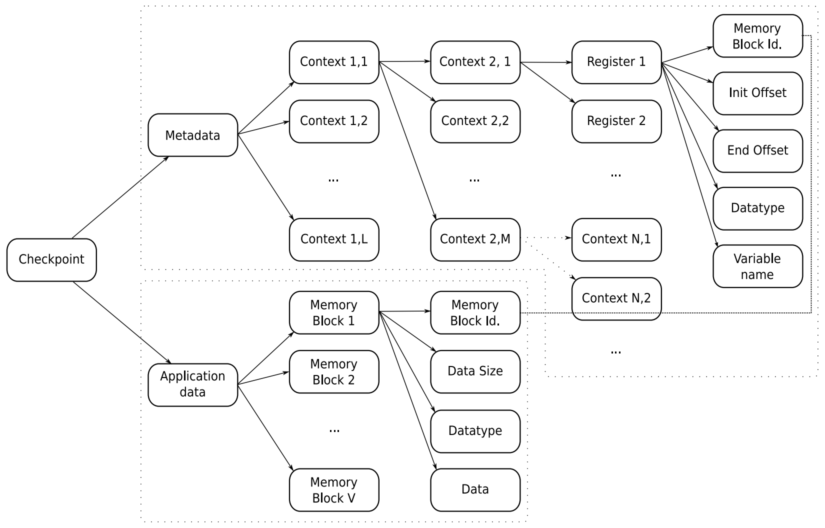


Figure 2. Data hierarchy format used by CPPC writing plugins

working with parallel applications it must be ensured that all processes are restarted in a consistent global state. The following subsections address the most relevant implementation issues in CPPC.

2.1. CHECKPOINT FILE DUMPING

From the point of view of the data stored in state files, tools can perform full checkpointing, storing the whole application state, or variable level checkpointing, saving user variables only. CPPC works at variable level, and stores only those variables which are needed upon application restart. When the execution flow reaches a checkpoint, data to be saved are structured and passed to the writing layer (see Figure 1). This includes both static and dynamic data. The actual data writing will be performed by the writing layer, using a user-selected plugin. The use of a portable format enables restart on different architectures. For this purpose, CPPC includes an HDF-5-based [12] writing plugin.

The CPPC library design allows for new writing plugins to be dynamically added. These must be able to accept the data format passed by the checkpointing layer. This format is hierarchically structured, as depicted in Figure 2. A checkpoint is divided in two different parts: a metadata section and an application data section. The application data section consists of memory blocks copied from the application memory, containing relevant data necessary for restart. The metadata section contains information for the correct recovery of the memory blocks, represented as a context hierarchy. Each of these contexts represents a call to a procedure, and contains information that enables the recovery of variables in that procedure scope. Contexts may also contain subcontexts, created by nested calls to the same or other procedures. This hierarchical representation allows for the application stack to be rebuilt upon restart, by recreating the sequence of procedure calls made by the original execution. The hierarchy generalizes any pattern in procedure calling, including recursive calls.

For each variable to be recovered, the context stores information about its type, an associated memory block in the application data section of the checkpoint file, and offsets to calculate the beginning and end of its data inside that memory block. The use of portable offsets inside a block instead of memory addresses enables pointer portability [20]. For each saved variable the CPPC library takes its beginning and end memory addresses, and compares them to the addresses of other registered variables. If any overlap exists, then the smallest memory block containing both variables is calculated. This memory block is then added to the checkpoint file under the application data section. In this way, a single memory block in this section may match multiple variables in the metadata section, depending on aliasing relationships. The beginning and end offsets in the metadata section are calculated taking into account the original address of the variable and the address of the corresponding block in the application data section. Upon recovery, the memory block will be loaded into memory, and memory addresses for the variables to be recovered will be back-calculated using the stored offsets. Thus, aliasing relationships are preserved through application restarts.

Besides, CPPC provides other checkpointing options such as multithreaded dumping. If multithreading is active, upon checkpointing CPPC first calculates aliasing relationships and memory blocks to be stored, which are copied over to ensure that the checkpointing will work over a clean, unmodified copy. Then, 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. CPPC also supports file compression, which may help save disk space and volume of network transfers, and includes a CRC-32-based algorithm for ensuring checkpoint file correctness upon restart.

2.2. GLOBAL CONSISTENCY

When checkpointing parallel applications, dependencies created by interprocess communication have to be preserved during recovery. If a checkpoint is placed in the code between two matching communication actions, an inconsistency exists when recovering, since the first one is not executed. If it is a send statement, it is called an *in-transit* message. If it is a receive, it is an *inconsistent* message.

Checkpoint-based rollback-recovery comes in three flavors: uncoordinated, coordinated and communication-induced. Uncoordinated protocols are subject to the domino effect, and do not guarantee progress. Communication-induced approaches are unpredictable, since checkpointing rates depend on the message-passing pattern of the application [4]. Coordinated approaches are the most common practical choice due to the simplicity of recovery. However, coordination protocols do not scale well. CPPC avoids these issues by focusing on SPMD parallel applications and using a non-blocking spatially coordinated approach. Checkpoints are taken at the same relative code locations by all processes (but not necessarily at the same time). By statically forcing checkpoints to occur at the selected places, no interprocess communications or runtime synchronization are necessary. To avoid 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. Thus, generated checkpoints are transitless and consistent. These points will be called *safe points*. To automatically identify safe points, a static communication analysis is performed by the CPPC compiler (see Section 3.4).

This coordination protocol achieves to improve both efficiency and scalability by transferring consistency concerns from runtime to compile time. Finding a valid recovery line is also very simple: it is formed by the most recent checkpoint file which is available to all processes.

2.3. RESTART PROTOCOL

The restart operation consists of three phases: negotiation, state file read and application state recovery. During the negotiation phase, application processes identify the most recent valid recovery line as previously described. Once each process has selected its appropriate local checkpoint file, the second phase reads its contents into memory, reconstructing the hierarchical format shown in Figure 2. Finally, in the third phase, the application state is effectively recovered.

Note that 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 that regenerates the original non-portable, non-stored state. CPPC uses code re-execution to recover this state. Non-portable state is recovered by the same means originally used to create it. Therefore, a CPPC application is as portable as the original one: variables are saved in a portable manner, non-portable state is recreated using the original code.

Each application is divided into blocks of *Required-Execution Code* (RECs) and blocks of code that can be safely skipped upon restart. The recovery process consists of the ordered execution of RECs, skipping non-relevant code. Examples of RECs are: the initialization of the CPPC library; the execution of variable registration procedures, which are responsible for recovering variable values when restarting; or the execution of procedures with non-portable outcome. REC detection is automated by the CPPC compiler, as described in the next section.

3. CPPC COMPILER

The CPPC compiler is built on the Cetus compiler infrastructure [8], 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.

The compiler performs code analyses and transformations in order to insert checkpoint instrumentation into an application. Some of these transformations require knowledge about procedure semantics (e.g. which procedure initializes the parallel system). We call these transformations *semantic-directed*. In order to provide semantic knowledge to the compiler, CPPC uses a catalog of procedures with their semantic behavior. An example of such catalog is shown in Figure 3(a), which exemplifies provided information for the `MPI_Comm_split` function. It implements the `CPPC/Nonportable` role. Upon detection of this semantic behavior, the CPPC compiler ensures that it is re-executed on restart, recovering the non-portable communicator it creates, as described in the next subsection.

```
<function name="MPI_Comm_split">
 <input parameters="1,2,3"/>
 <output parameters="4"/>
 <semantics>
  <semantic role="CPPC/Nonportable"/>
 </semantics>
</function>
```

(a) Semantic information for `MPI_Comm_split`

```
CPPC JUMP LABEL
REGISTER 'COLOR' PARAMETER
REGISTER 'KEY' PARAMETER
COMMIT REGISTERS
MPI_Comm_split( comm, color, key,
 comm_out );
CONDITIONAL JUMP TO NEXT REC
```

(b) Pseudo-code of a non-portable procedure call transformation

Figure 3. Semantic information and compiler transformation examples

The use of this semantic catalog enables transformations to work with different programming interfaces for a given subsystem. Extensions to other message-passing libraries would only require the addition of the corresponding semantic information. Note that the semantic catalog is not intended to be created by CPPC users, but provided together with the compiler distribution. In the current CPPC distribution the catalog includes information about Fortran 77 built-in functions, common UNIX functions (e.g. I/O functions to manage file streams), and the MPI standard (in both C and Fortran 77 versions).

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

The CPPC library recovers non-portable state by means of the re-execution of the code responsible for creating such state in the original run. The `CPPC/Nonportable` semantic role is used for identifying procedures that create or modify such state. Upon discovery of a non-portable call, the CPPC compiler performs the transformation depicted in Figure 3(b). It inserts a register for each input or input-output parameter passed to the call. The basic functional difference between a regular variable registration and a parameter registration is that in the former the variable address is saved and its contents stored when the checkpoint function is called. The value of a parameter, however, is stored in volatile memory when the `COMMIT REGISTERS` operation is invoked, and included in all subsequent checkpoint files. Therefore, the call will be re-executed using the same parameter values as in the original execution. The compiler also adds control code to ensure that the program flow enters the REC block and is directed towards the next one after executing it.

When restarting the application, control flow will be directed towards this REC, and values for `color` and `key` will be recovered. The `comm` variable (an MPI communicator) will have already been recovered through a previous re-execution of non-portable code. The non-portable call is performed in the same conditions as the original one, thus having the same semantic outcome: a new communicator. Note that the specific MPI implementation used in the application re-execution could be different from the one used in the original run, but the outcome will be semantically correct in the new execution environment.

3.2. 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 complementary approach to memory exclusion techniques used in sequential checkpointers to reduce the amount of memory stored, such as the one proposed in [13].

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

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

where $LV_{out}(s)$ is the set of live variables after executing statement s, and $USE(s)$ and $DEF(s)$ are the sets of used and defined variables by s, respectively. This analysis, which takes into account interprocedural data flow, is performed backwards, being $LV_{out}(s_{end}) = \emptyset$, and s_{end} the last statement of the code.

Before each checkpoint statement c_i, the compiler inserts registers to mark the variables that must be stored in the checkpoint file, which are those contained in the set $LV_{in}(c_i)$. The data type for the register is automatically 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 unregisters variables in the set $LV_{in}(c_{i-1}) - LV_{in}(c_i)$, for variables that are no longer relevant.

Checkpoints can be placed inside any given procedure. For a checkpoint statement c_i, let us define:

$$B_{c_i} = \{s_1 < s_2 < \ldots < s_{end}\}$$

as the ordered set of statements contained in all control flow paths from c_i (excluding c_i) and up to the last statement of the program code, where the $<$ operator indicates the precedence relationship between statements. Note that, if a checkpoint is placed inside a procedure f, not all statements in the set B_{c_i} will be inside f. Let us denote by $B^f_{c_i} = \{s_1 < \ldots < s_n\}$ the ordered set of statements contained inside f, and $L^f_{c_i} = B_{c_i} - B^f_{c_i}$ the ordered set of statements left to be analyzed outside f.

The interprocedural analysis and register insertion is performed according to the following algorithm:

- For a checkpoint statement c_i contained into a procedure f, the live variable analysis is performed for the set $B^f_{c_i}$, and registers for locally live variables are inserted before c_i.
- For the set $L^f_{c_i}$, containing the statements that are left unanalyzed in the previous step, let us consider g to be the procedure containing the statement s_{n+1}. The statement executed immediately before s_{n+1} must be a call to f. Note that $L^f_{c_i} = B^g_{c_i} \sqcup L^g_{c_i}$.
  The live variable analysis is performed for the set $B^g_{c_i}$, and registers for locally live variables are inserted before the call to f.
- The process is repeated for the statements contained in the ordered set $L^g_{c_i}$.

This algorithm ensures that, upon application restart, all variables will be defined before being used, and thus the portable state of the application will be correctly recovered. Proofs of correctness and termination are omitted due to space constraints.

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, registration calls will be inserted for the previously unrecovered variables contained in the set $LV_{in}(s_p)$, being s_p the analyzed procedure call. To avoid this default behavior, data flow information may be included in the semantic catalog.

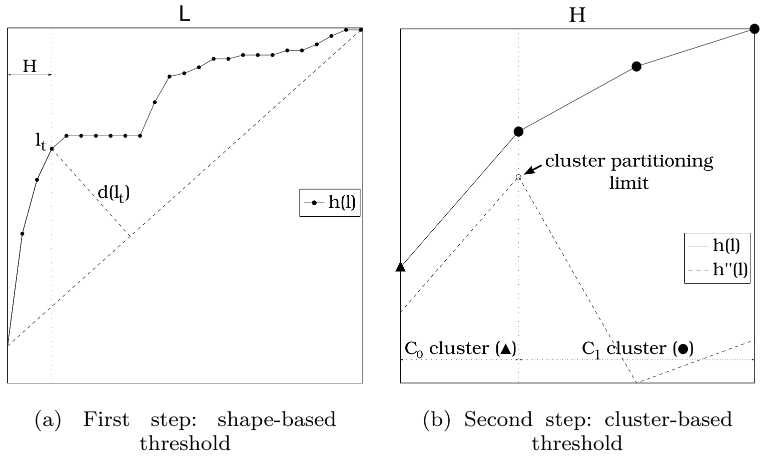
3.3. CHECKPOINT INSERTION

The compiler locates sections of the code that take a long time to execute, where checkpoints are needed to guarantee execution progress in the presence of failures. Since computation time cannot be accurately predicted at compile time without knowledge of the computing platforms 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. Currently, a metric derived from both the number of statements executed inside the loop and the number of variables accessed in them is used. A loop might have many statements but accessing mostly constants. These are usually not good candidates for checkpointing, since they involve initializing variables and their execution does not last long. In this case, considering the number of variable accesses in the loop reduces the cost associated to such loops, thus making them less prone to be checkpointed.

Let l be a loop in L, the loop population of the application P, and s and a two functions that give the number of statements and variable accesses in a given block of code, respectively. Let us define $S(l) = s(l)/s(P)$ and $A(l) = a(l)/a(P)$ the total proportion of statements and accesses, in that order, that exist inside a given loop l. The heuristic complexity value associated to each loop l is calculated as:

$$h(l) = -log(S(l) \cdot A(l)) \tag{1}$$

Eq. (1) multiplies $S(l)$ and $A(l)$ to ensure that the product is bigger for loops that are significant for both metrics. It applies a logarithm to make variations smoother. Finally, it takes the negative of the value to make $h(l)$ strictly positive. Thus, a lower value implies that more computing time is estimated for that loop. In order to select the best candidates for checkpoint insertion, the compiler ranks loops in L attending to $h(l)$ and applies thresholding methods [18]. After exploring several possibilities, a two-step method has been adopted. First, a “shape-based” approach is used to select the subset of the time-consuming loop nests in the application. The second step improves this selection by means of a “cluster-based” technique. Figure 4 shows the proposed method applied to the BT application of the NAS Parallel Benchmarks [11]. In the first step, using the so-called “triangle method”, loops in L are divided in two classes: time-consuming loops and negligible loops. Conceptually, the triangle method consists in: (a) drawing a line between the first and last values of $h(l)$; (b) calculating the perpendicular distance $d(l)$ to this line for all $l \in L$; and (c) calculating a threshold value l_t such that $d(l_t) = max\{d(l)\}$. This first step selects a subset of the loop population, referred to as H, as time-consuming loops candidates for checkpoint insertion, as shown in Figure 4(a).



(a) First step: shape-based threshold

(b) Second step: cluster-based threshold

Figure 4. Example of checkpoint insertion for the NAS BT application

The triangle method was originally developed in the context of image processing, and appears to be especially effective when there is a narrow peak in the histogram, which is often the case for the proposed $h(l)$ function in real applications.

However, this first threshold selects more loops than it would be desirable for checkpoint insertion. The second step of the proposed algorithm refines this selection using a cluster-based thresholding algorithm. Loops in H with similar associated costs are grouped into clusters, C_i, built by selecting the local maximums of the second derivative of $h(l)$ as partitioning limits. Since $h(l)$ is monotonically increasing, local maximums in $h''(l)$ represent inflection points at which $h(l)$ begins to change more smoothly. For an application with k clusters, the method calculates a threshold value t such that:

$$\sum_{i=0}^{t} h(l_{i+1}^0) - h(l_i^0) > \sum_{i=t+1}^{k-1} h(l_{i+1}^0) - h(l_i^0) \tag{2}$$

where l_i^0 denotes the first loop in C_i. This method selects for checkpoint insertion all the loops inside the clusters C_i such that $i \leq t$. In the example shown in Figure 4(b), loops in subset H are grouped in two clusters, delimited by the single maximum in $h''(l)$. Applying Eq. (2), the compiler selects cluster C_0 for checkpoint insertion, which contains only one loop (which corresponds to the main computational loop in BT).

Once the loops in which checkpoints are to be inserted are identified, the compiler performs the communication analysis described in the next subsection and inserts a checkpoint at the first available safe point in each selected loop nest. Experimental results are detailed in Section 4.

3.4. SAFE POINT IDENTIFICATION

In order to automatically find regions in the code where neither in-transit nor inconsistent communications exist, the compiler must analyze communication statements and match sends to their respective receives. The approach used by the compiler is similar to a static simulation of the execution. Two communications are considered to match if two conditions hold: (a) their tags are the same or the receive uses `MPI_ANY_TAG`; and (b) their sets of sources/destinations are the same: if a process i sends the message to a process j, then j must execute the corresponding receive statement using i as source.

In order to statically compare tags and source/destination pairs, the compiler performs symbolic analysis and constant propagation to determine their literal values. *Communication-relevant* variables are recursively found as: (a) variables directly used in tags or source/destination parameters; and (b) variables on which a communication-relevant variable depends. In order to optimize this process, only statements which modify communication-relevant variables are analyzed, since any other does not affect communications. Some communication-relevant variables are multivalued, that is, they potentially have a different value for each process. Thus, the compiler needs to know how many processes are involved in the execution of the code to perform the symbolic analysis. The constant propagation is performed together with the communication matching in the same compilation pass.

For keeping track of the communications status, the compiler uses a buffer object, which starts out empty. The analysis begins at the application's entry point. Statements that are neither control flow- nor communication-related are ignored. Each time the compiler finds a new communication, it first tries to match it with existing ones in the buffer. If a match is not found, the communication is added to the buffer and the analysis continues. If a match is found, both statements are considered linked and removed from the buffer, except when matching non-blocking sends and receives, in which case they remain in the buffer in an unwaited status until a matching wait is found.

When the compiler finds a procedure call, it stops the ongoing analysis and moves to analyze the code of the called procedure, using the same communications buffer. However, communications issued inside the procedure are also cached separately for optimization purposes. It can be proven that, if the procedure does not modify any communication-relevant variable, cached results can be reused when a new call to the same procedure is found, without analyzing the procedure again, but just symbolically analyzing their tags and source/destination parameters again. The compiler employs this optimization whenever possible. If the procedure modifies any communication-relevant variable, then its code must be analyzed each time a call is found.

A statement in the application code will be considered a safe point if, and only if, the buffer is completely empty when the analysis reaches that statement. An empty buffer implies that no pending communications have been issued, and therefore it is impossible for an in-transit or inconsistent message to exist at that point.

For collective communications spread across several conditional paths, the compiler detects them as linked and ensures that no checkpoints are inserted such that some processes take its local checkpoint before the collective while some others do so after it.

For applications that present non-deterministic or irregular communication patterns (those depending on runtime input data), the compiler must apply either a conservative approach or heuristics in order to detect safe points. A conservative approach involves matching communications to their latest possible match in the code, although this can lead to the compiler not finding safe points at loops selected by the checkpoint insertion analysis. The heuristic currently in use is to consider that two communications match if no matching incompatibility between their tags and source/destination pairs is found.

4. EXPERIMENTAL RESULTS

For testing purposes, a twofold approach has been followed. The NAS Parallel Benchmarks (NPB-MPI v3.1) [11] have been selected due to their widespread use. These applications have short execution times, so they are not appealing options for applying checkpointing techniques in practice. For this reason, a crack growth simulation code called DBEM [7], as well as an air quality simulation application named STEM [10], have also been tested. All codes are written in Fortran 77, except for one of the NPB applications, IS, which is written in C.

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 inserted checkpoints and variable registrations can be seen in Table I. The compiler automatically inserted checkpoints in the most computationally intensive loops for all applications, adequately placing them at safe points inside each loop. For instance, for the NAS BT application, the checkpoint was inserted as the first statement in the loop starting at line 190 in file `bt.f`, which is the main computational loop. One checkpoint was also inserted for the rest of the NPB in their respective main loops, with the exceptions of CG, which has two copies of the main loop that were checkpointed at lines 344 and 441 of file `cg.f`; and IS, where four loops were checkpointed (including the main loop) because the application has many loops with similar computational costs. For the other applications, four checkpoints were inserted in DBEM, corresponding to the two main loops and two smaller ones; and one checkpoint for the main computational loop in STEM.

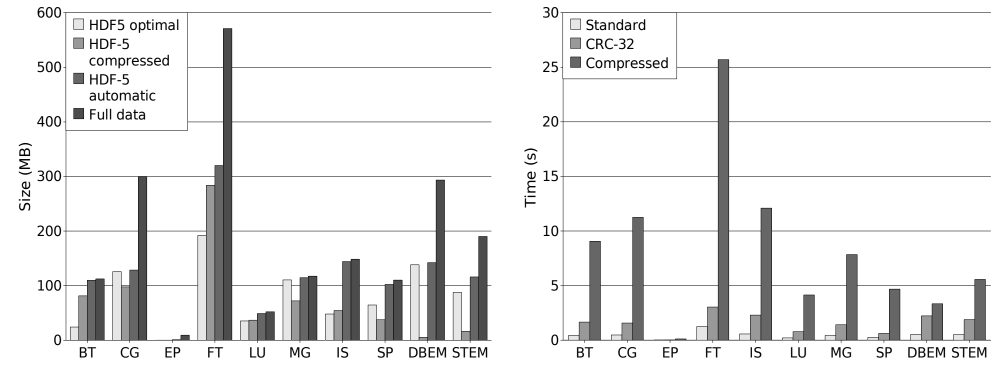
Tests were performed on a cluster of Intel Xeon 1.8 Ghz nodes, 1 GB of RAM, connected through an SCI network. Size of generated state files, time for state file generation, checkpoint overhead and restart times were measured. The runtimes presented in this section were obtained for executions on 4 processors. However, tests on increasingly large configurations (up to 128 processors) were executed, obtaining roughly the same checkpoint overhead percentages, which demonstrates the scalability of the tool. For proving portability, these applications were also run on an HP Superdome, with proprietary C/Fortran compilers and also a 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.

4.1. STATE FILE SIZES

When using CPPC's spatially coordinated technique, the incurred overhead will only depend on the overhead introduced by the checkpoint file dumping. This overhead heavily depends

Table I. Test applications

		Files	LOCs	Compile time (s)	# Chkpts.	# Registers
NPB	BT (class=B)	18	3650	13.84	1	193
	CG (class=C)	1	1044	2.87	2	37
	EP (class=C)	1	180	0.90	1	14
	FT (class=B)	1	1268	4.87	1	31
	IS (class=B)	1	671	3.70	4	28
	LU (class=B)	25	3086	7.81	1	89
	MG (class=B)	1	1618	12.95	1	34
	SP (class=B)	24	3148	9.96	1	129
	DBEM	42	12535	72.68	4	180
	STEM	141	7524	18.48	1	156



(a) Generated file sizes per process

(b) File generation overhead

Figure 5. File sizes and generation overhead

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 Figure 5(a). The values tagged as “HDF-5 automatic” are file sizes obtained by the automatic variable registration included in the compiler. “HDF-5 compressed” shows file sizes using the HDF-5 writer with compression enabled. For comparison purposes, “HDF-5 optimal” shows the optimal file sizes, obtained by a manual analysis, and “Full data” presents the sizes obtained for a checkpointer that stores all the application data.

Sizes obtained using automatic analyses are close to the optimal ones, except for BT, IS and SP. This is due to the fact that unnecessary array sections are registered because of the conservative approach of the compiler, as explained in Section 3.2. 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 58.06%. For the DBEM and STEM applications, the reduction is 52.92% and 53.91%, respectively.

The high compression rates obtained for DBEM and STEM (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 and the writing strategy used. Figure 5(b) shows the times obtained for a standard HDF-5 file creation, for the same HDF-5 file including a CRC-32 error detection scheme, and for compressed HDF-5 files. 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 is reduced by using multithreaded dumping.

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; for instance, 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 II details the original execution times (without checkpointing) and the overhead introduced by checkpointing. This overhead includes, besides file generation, all CPPC instrumentation (e.g. variable and parameter registration). Files were generated using the HDF-5 writer, with CRC-32 and without compression. One state file per checkpoint was generated for the NPB benchmarks, while checkpointing frequency was adjusted to create approximately one per hour for DBEM and STEM. Increasing the frequency up to one checkpoint each ten minutes 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.

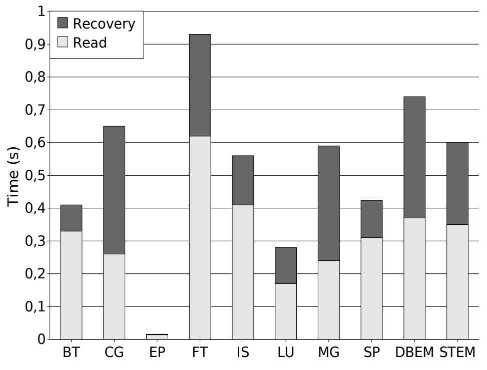
As can be seen, MG and IS have such a short execution time that checkpoint overhead is relatively high (although this overhead is in the range of only seconds). However, those applications with higher runtimes present very low overheads: only 0.31% and 0.47% for DBEM and STEM, with runtimes of more than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

4.4. RESTART OVERHEAD

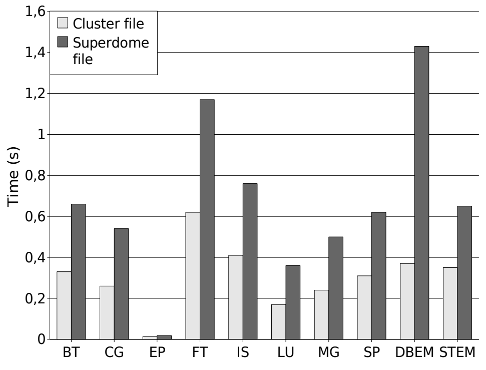
If a failure occurs, restart time overhead must be taken into account in the global execution time. Restart times have been measured and split into its two fundamental phases: file read and effective data recovery. Results are depicted in Figure 6(a). Negotiation times have been omitted because they are negligible (in the order of milliseconds for all applications). As can be seen, restart times are very low (less than one second).

Table II. Multithreaded checkpoint overheads

		Original runtime	Overhead (#chkpts.)	Overhead percentage
NPB	BT	638.37 s	2.70 s (1)	0.42%
	CG	617.28 s	7.27 s (2)	1.18%
	EP	304.32 s	1.45 s (1)	0.47%
	FT	186.07 s	2.22 s (1)	1.19%
	IS	5.72 s	1.50 s (4)	26.22%
	LU	467.08 s	4.70 s (1)	1.01%
	MG	21.47 s	2.05 s (1)	9.55%
	SP	952.29 s	5.23 s (1)	0.55%
	DBEM	80473.13 s	256.02 s (23)	0.31%
	STEM	21622.41 s	101.14 s (7)	0.47%



(a) Restart times



(b) Comparison of file read times

Figure 6. Restart times and file read times for test applications

File read time measurement begins when the negotiation ends, and 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 Figure 6(b), which details file read times for both cluster- and Superdome-generated files when restarting the test applications on the cluster. This test has also served 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 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 code re-execution.

5. RELATED WORK

Currently, research and development on checkpointing frameworks is focused on achieving operation over potentially large and heterogeneous systems. Issues concerning scalability, efficiency and portability are of particular interest.

Checkpoint file sizes are tightly related to network and file system scalability issues when checkpointing parallel applications. Thus, different approaches to reduce the amount of data stored in state files have been studied during the last decade. Plank et al. proposed in [13] a memory exclusion technique based on performing a dead variable analysis to identify memory regions which can be safely excluded from the checkpoint process. CPPC implements a similar technique, based on performing a live variable analysis at compile time to identify those variables that should be included in the checkpoint file. Another technique to reduce file sizes is incremental checkpointing, also proposed by Plank et al. [14] and recently implemented in different works [1, 6]. Incremental checkpointing techniques involve using the operating system's page protection mechanisms to detect, when a checkpoint is reached, which pages have changed since the last one, and to save only those. During a restart, the state is restored using the first checkpoint file, and then all the differences are applied in order before the execution is allowed to resume. Although incremental checkpointing techniques have been typically applied to system level approaches, a similar idea could be incorporated to CPPC's variable level approach.

Another strategy for avoiding network contention is staggered checkpointing [23]. Using this technique, processes checkpoint at different points in the code, hence staggering the data transfer over the network and avoiding the bottleneck it represents. An additional technique to reduce contention, used by Preaches [19], is to save checkpoints to local disk and then transfer them in a staggered way, possibly using an administration network if available, instead of the computing network. This approach could be easily used together with CPPC.

With regards to portability issues, Porch [15] is a source-to-source compiler that translates sequential C programs into semantically equivalent ones which are capable of saving 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 the operation. The data hierarchy format used by CPPC writing plugins to portably handle variables, stack structures and pointers was inspired by Porch.

Preaches is also a single-process checkpointer that addresses portability issues. Its portability model is based on checkpoint propagation. A primary process executes the computation core, and communicates with secondary lightweight processes executing on the target architectures in which a restart is potentially required. The primary process sends data to be stored in checkpoint files, while secondary processes create replicas using their local machine format.

The checkpointing tool most closely related to CPPC is PC^3 [5] (based on C^3 [2]). Both share the same goals: developing checkpointing techniques for parallel applications that enable a portable operation, while obtaining transparency and low overheads. The difference comes from the techniques used to achieve these objectives. PC^3 employs a coordinated protocol based on piggybacking information into sent messages, while CPPC uses a spatially coordinated approach which moves all necessary synchronizations to the restart phase. Both approaches present benefits and drawbacks. Coordinating processes at runtime allows for checkpoints to be taken based on a temporal frequency, rather than spatial frequency (i.e. each hour instead of each thousand iterations of a given loop), which is a more natural concept to users. However, CPPC's approach achieves high scalability, since it introduces a small overhead at runtime which does not depend on the number of processes running the parallel application. PC^3 has to catch and modify every single message being sent, in order to piggyback its state information. This becomes significant when dealing with applications with intensive collective communications, since each collective message is translated to several point-to-point ones.

Regarding the insertion of checkpoints into applications, none of the approaches in the literature, to the authors' best knowledge, perform complexity analyses of the code. Instead, they typically take the "potential checkpoint" approach [9], where many checkpoint calls are inserted at key locations in the code but only executed according to a checkpoint frequency timer. These typically perform simple structural analyses, such as inserting potential checkpoints at every loop and procedure call in the application. This approach cannot be followed by CPPC, where checkpointing frequencies cannot be defined in temporal terms, due to the need to statically coordinate all processes independently of how long they take to progress through the application's execution. Some theoretical approaches calculating the optimal mathematical solution to the checkpoint placement problem exist [21, 22]. However, these assume that all involved parameters are known. This is not the case under CPPC, where the execution environment is not assumed to be fixed due to its characteristic of portability.

6. CONCLUDING REMARKS

CPPC is a portable and transparent checkpointing infrastructure for parallel applications. It uses a variable level, modular approach to achieve scalability, efficiency and portability. The two most remarkable contributions of this framework address consistency issues and the portable recovery of the application's state.

Consistency issues are moved from runtime to 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, thus improving scalability. Moreover, this solution also enhances efficiency, since both compiling and restarting an application are less frequent operations than checkpoint generation.

State recovery is achieved in a portable way by means of both the hierarchical data format used for state dumping, 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.

Portability is a very interesting trait due to the inherent heterogeneity of current trends in HPC, such as Grid computing. CPPC-G [17] is an ongoing project developing an architecture based on web services to manage the execution of CPPC fault tolerant applications on the Grid.

CPPC has been experimentally tested, demonstrating usability, efficiency, and portability. 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` under GPL license.

REFERENCES

1. Agarwal S, Garg R, Gupta MS, Moreira JE. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th ACM International Conference on Supercomputing* 2004; 277–286.
2. Bronevetsky G, Marques D, Pingali K, Stodghill P. C^3: A system for automating application-level checkpointing of MPI programs. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing* 2003; 357–373.
3. Elnozahy EN, Alvisi L, Wang YM, Johnson DB. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys* 2002; **34**(3):375–408.
4. Elnozahy EN, Plank JS. Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing* 2004; **1**(2):97–108.
5. Fernandes R, Pingali K, Stodghill P. Mobile MPI programs in computational Grids. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming* 2006; 22–31.
6. Gioiosa R, Sancho JC, Jiang S, Petrini F. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Proceedings of the ACM/IEEE Supercomputing* 2005; 9–23.
7. González P, Pena TF, Cabaleiro JC. Dual BEM for crack growth analysis on distributed-memory multiprocessors. *Advances in Engineering Software* 2000; **31**(12):921–927.
8. Lee SI, Johnson TA, Eigenmann R. Cetus – an extensible compiler infrastructure for source-to-source transformation. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing* 2003; 539–553.
9. Li CCJ, Stewart EM, Fuchs WK. Compiler-assisted full checkpointing. *Software: Practice and Experience* 1994; **24**(10):871–886.
10. Martín MJ, Singh DE, Mouriño JC, Rivera FF, Doallo R, Bruguera JD. High performance air pollution modeling for a power plant environment. *Parallel Computing* 2003; **29**(11–12):1763–1790.
11. National Aeronautics and Space Administration. The NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/
12. National Center for Supercomputing Applications. HDF-5: File Format Specification. http://hdf.ncsa.uiuc.edu/HDF5/doc/
13. Plank JS, Beck M, Kingsley G. Compiler-assisted memory exclusion for fast checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments* 1995; **7**(4):10–14.
14. Plank JS, Xu J, Netzer RHB. Compressed differences: an algorithm for fast incremental checkpointing. Technical Report CS-95-302. University of Tennessee, Department of Computer Science. 1995.
15. Ramkumar B, Strumpen V. Portable checkpointing for heterogeneous architectures. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing* 1997; 58–67.
16. Rodríguez G, Martín MJ, González P, Touriño J. Controller/precompiler for portable checkpointing. *IEICE Transactions on Information and Systems* 2006; **E89-D**(2):408–417.
17. Rodríguez G, Pardo XC, Martín MJ, González P, Díaz D. A fault tolerance solution for sequential and MPI applications on the Grid. *Scalable Computing: Practice and Experience* 2008; **9**(2):101–109.
18. Sezgin M, Sankur B. Survey over image thresholding techniques and quantitative performance evaluation. *Journal of Electronic Imaging* 2004; **13**(1):146–165.
19. Ssu KF, Fuchs WK, Jiau HC. Process recovery in heterogeneous systems. *IEEE Transactions on Computers* 2003; **52**(2):126–138.
20. Strumpen V. Portable and fault-tolerant software systems. *IEEE Micro* 1998; **18**(5):22–32.
21. Toueg S, Babaoğlu Ö. On the optimum checkpoint selection problem. *SIAM Journal on Computing* 1984; **13**(3):630–649.
22. Vaidya NH. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Transactions on Computers* 1997; **46**(8):942–947.
23. Vaidya NH. Staggered consistent checkpointing. *IEEE Transactions on Parallel and Distributed Systems* 1999; **10**(7):694–702.