Cetus-assisted checkpointing of parallel codes

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

***Abstract*—Checkpointing tools may be typically implemented at two different abstraction levels: at the system level or at the application level. The latter has become an interesting alternative due to its flexibility and the possibility of operating in different environments. However, application-level checkpointing tools often require the user to manually insert checkpoints in order to ensure that certain requirements are met (e.g. forcing checkpoints to be taken at the user code and not inside kernel routines). This paper examines the Cetus transformations implemented into the CPPC checkpointing framework enabling automatic checkpointing of parallel applications.**

***Index Terms*—Fault tolerance, checkpointing, parallel programming, message-passing, compiler-support, Cetus**

I. Introduction: The CPPC Framework

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 [1], each having its own pros and cons.

The ComPiler for Portable Checkpointing (CPPC) is a checkpointing framework for message-passing applications with an emphasis on portability. It is an open-source tool, available at http://cppc.des.udc.es under the GNU General Public License (GPL). It consists of a runtime library containing checkpointing-support routines and a compiler that automates the use of the library. This work describes the implementation on Cetus [2] of the compilation techniques for the automatic insertion of checkpointing instrumentation. The remainder of this section summarizes various fundamental design aspects of the framework in order to introduce the necessity for such a compiler. For an in-depth description of CPPC the reader is referred to [3].

A. Portability

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 machine-dependent state. Rather, this state should be recovered at 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 opaque state in the original execution. This protocol, together with the use of portable storage formats, enables the restart on different architectures. The target application code must be instrumented in order to effectively implement the restart protocol, directing the control flow to the relevant code snippets. This restart protocol is further discussed in Sections II-D and II-E.

The authors are with the Computer Architecture Group, Department of Electronics and Systems, University of A Coruña, Spain.

E-mail: {grodriguez, mariam, pglez, juan, doallo}@udc.es

B. Relevant state selection

The solution of large real scientific problems requires the use of large computational resources, both in terms of CPU 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, leads to a large storage size, becoming impractical [4]. Besides, the size of the state files is one of the most significant performance-impacting factors in checkpointing. CPPC reduces the amount of data saved by storing only relevant user variables. The relevance of each variable is determined by a live variable analysis that identifies those values that are needed for the correct restart of an execution. The process of marking a variable to be included in subsequent state files is called *variable registration*.

C. Spatial coordination

When checkpointing 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. 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 [5]. Checkpoints are taken at the same relative code locations by all processes, but not forcibly at the same time. By statically ensuring that checkpoints are taken at points where no in-transit nor inconsistent messages may exist the necessity for interprocess communications or runtime synchronizations is removed. These points will be called *safe points*.

II. Compilation analyses

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

The most relevant transformations applied by the Cetus-based compiler are, in this order: (1) the communication analysis required in order to automatically detect safe points where

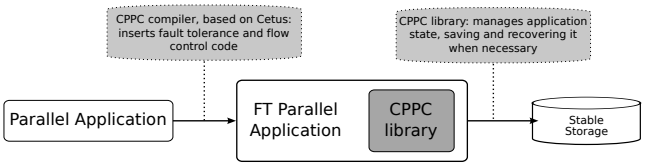

Fig. 1. CPPC framework design

to insert checkpoints, (2) a computational load estimation that selects code loops for checkpoint insertion, (3) the detection of the variables that are live at selected checkpoint locations, and are therefore necessary during application restart, and (4) the code instrumentation to coordinate all parts of the checkpointing runtime system.

The source distribution of CPPC includes a function catalog that contains information about different families of functions, such as functions in the MPI interface or POSIX functions available in most *NIX distributions. This information will be necessary in order to inform the compiler about the particular behavior of certain key functions in parallel applications, such as which functions are related to interprocess communication (see Section II-A); which ones generate non-portable state that must be recovered through code re-execution (as shown in Section II-D); and whether a function parameter is of input, output or input-output type (this information will be used for optimizing the amount of state to be saved during checkpoints, as further detailed in Section II-C).

The following subsections describe the fundamental transformations performed by the compiler, as well as Cetus extensions required for supporting Fortran 77 codes.

A. Communications analysis

In order to automatically find safe points in the code, communication statements must be analyzed to match sends with their corresponding receives. The proposed approach is similar to a static simulation of the program execution, and focuses on MPI codes, although it would be easily adaptable to other message-passing libraries. Two communications are considered to match if the following conditions hold:

1) Their sets of sources/destinations are the same: if process p_i executes the send statement using process p_j as destination, then p_j executes the receive statement using p_i as source.
2) Their tags are the same or the receive uses the wildcard `MPI_ANY_TAG`.

Constant folding and propagation are carried out in order to statically compare communications statements. Since this operation is very computationally expensive, it is restricted to the set of variables and statements that are relevant to communications.

Some communication-relevant variables are multivalued, that is, they have potentially different values for different MPI processes. As such, the number of processes involved in the execution of the code must be known statically to achieve more accurate results. The constant folding and propagation may be performed together with the communications matching in the same compiler pass. This pass starts at the callgraph root, and analyzes code in execution order. It maintains a buffer to store found communications. Each time a new communication statement s_c is discovered, it is first matched against existing ones in the buffer. If a match is not found, s_c is added to the buffer and the analysis continues. If a match s_m is found, both s_c and s_m are considered linked and removed from the buffer, except when matching a pair of non-blocking sends/receives. In this case, they remain in the buffer in an "unwaited" status until a matching wait 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.

When a procedure call is found, the ongoing analysis is stopped and the compiler begins an on-demand analysis of the callee, using the same communication buffer. However, the compiler also caches separately the communications issued inside the callee. If the procedure does not modify any communication-relevant variable, then this cache may be used when a new call to the procedure is found without reanalyzing the procedure code, but only symbolically analyzing the cached communications with the current set of known constant values. Figure 2 presents a pseudo-code of this analysis.

When dealing with applications featuring ambiguous communications the proposed solution might be unable to find suitable safe points. However, such situations are uncommon, as shown in the experimental assessment of the tool [6].

B. Checkpoint insertion

In order to guarantee execution progress in the presence of failures, checkpoints need to be inserted at places in the code that perform the core of the computation, and thus take the longest time to execute. However, it is not possible to accurately predict the execution time of a section of code without precise knowledge of the hardware executing it. To overcome this issue heuristic cost analyses are employed, using computational metrics to discover critical sections of code. The cost estimation is performed interprocedurally and traversing the Cetus IR. First, an estimated cost value is assigned to leaf nodes (simple statements). Then, the IR is traversed upwards, estimating the cost for executing a parent node by looking at the estimated costs of its children nodes (i.e. a compound statement –further explained in Section II-C– adds together the costs for all its children, a conditional statement calculates the average cost of all conditional branches). The compiler then selects the most relevant loop nests in the application using a dynamic thresholding method. Loop nests above the selected cost threshold are checkpointed, while those below it are deemed not relevant enough to justify checkpoint insertion. The decision to dynamically select the threshold, as opposed to defining it as a hard-coded constant, responds to a desire to take into account how the total computational load of the application is divided into its loop nests before selecting checkpoint locations.

Once the loops in which checkpoints are to be inserted are identified, the results of the communications analysis are used

```
buffer: communications buffer
known_constants: set of code variables
    with known values
procedure communications_analysis
 detect communication relevant variables
 buffer ← ∅
 known_constants ← ∅
 call analyze_procedure( main_procedure )
end procedure communications_analysis

buffer_cached_p : cached results for procedure p
procedure analyze_procedure( p )
 /* try to use previously cached results */
 if buffer_cached_p ≠ null AND
    p does not modify comm. relevant vars. then
  /* this operation merges communications
     issued by p into buffer */
  merge buffer_cached_p into buffer
  return
 fi

 /* if not possible, analyze procedure */
 buffer_cached_p ← ∅
 foreach statement s in p do
  if buffer is empty then
   mark s as a safe point
  fi
  if s is a communication statement then
   buffer_cached_p ← buffer_cached_p ∪ {s}
   call analyze_communication( s )
  elseif s modifies comm. relevant vars. then
   perform symbolic analysis of s and
      update known_constants
  elseif s is a call to a procedure p' then
   call analyze_procedure( p' )
  fi
 done
end procedure analyze_procedure( p )

procedure analyze_communication( c_1 )
 c_2 ← match for c_1 in buffer
 if c_2 = null then
  add c_1 to buffer as unmatched
 elseif c_1 is a wait statement
  remove c_2 from buffer
 else /* c_1 is a send or a recv */
  remove c_2 from buffer
  foreach c in {c_1, c_2} do
   if c is a non-blocking communication then
    add c to buffer as unwaited
   fi
  done
 fi
end procedure analyze_communication( c_1 )
```

Fig. 2. Pseudo-code of the communication analysis

```
generated: set of generated variables
consumed: set of consumed variables
foreach statement s in {c_i ... s_end} do
 consumed ← consumed ∪ (s.consumed − generated)
 generated ← generated ∪ s.generated
done
LV_in(c_i) = consumed
```

Fig. 3. Pseudo-code of the live variable analysis

to insert a checkpoint at the first available safe point in each selected loop nest.

This technique can also be used to detect adequate checkpoint locations when using other application-level checkpointing approaches (e.g. uncoordinated, distributed snapshots, etc.).

C. Registration of restart-relevant variables

As described in Section I-B, the compiler identifies the variables that will be relevant upon restart and marks them for storage in subsequent checkpoints. It is easy to see that, for a checkpoint statement c_i, these variables can be identified by calculating the set $LV_{in}(c_i)$ of variables that are live at c_i. 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 [7].

The traditional liveness analysis works at the basic block level, starting at the last basic block in the application and proceeding up in the control flow graph. In Cetus, the code is not organized in basic blocks, but rather in *compound statements*. Given that a compound statement may contain different control flow paths it is inadvisable to use them as the basic piece of information when calculating live variables in Cetus. CPPC annotates each statement s with its corresponding $USE(s)$ and $DEF(s)$, the sets of variables used and defined by s, respectively. When it needs to obtain the set of live variables at a statement s it will traverse all statements from s up to s_{end}, the last statement in its containing procedure. The pseudo-code for the live variable calculation is shown in Figure 3.

Before each checkpoint c_i, the compiler inserts registers for the variables in $LV_{in}(c_i)$. Also, before each c_i, the compiler unregisters variables in the set $LV_{in}(c_{i-1}) - LV_{in}(c_i)$, that is, the set of variables that are no longer relevant.

Note that a checkpoint c_i may be contained inside any given procedure f. However, the liveness analysis only takes into account the code from c_i to the last statement in f, s_{end}. In order to correctly detect live variables in this situation the compiler first needs to analyze the callgraph and identify the set $G = \{g_1, \ldots, g_n\}$ of procedures containing calls to f. It then proceeds to treat each call s_f in G exactly in the same way as a checkpoint c_i, inserting registers for the variables in $LV_{in}(s_f)$. In this way, the set of locally live variables in each procedure is recovered inside the procedure itself. Further details about state recovery during application restart are given in Section II-E.

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

Fig. 4. Pseudo-code of a non-portable procedure call transformation

The live variable analysis must be performed interprocedurally. Upon finding a call to a procedure h, the compiler performs an on-demand analysis of the code of h. The dataflow effects on procedure parameters and global variables are then cached to be used in subsequent calls to h. When dealing with calls to precompiled procedures located in external libraries, the conservative behavior is to assume all parameters to be of input type. This forces all variables passed as procedure parameters to be generated before the call, either as part of the execution of the code or by means of a variable registration. To avoid this default behavior dataflow information may be included in the function catalog.

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.

D. Identification of non-portable functions

As stated in Section I-A, CPPC recovers non-portable parts of the application state through the re-execution of the code creating such opaque state, such as MPI communicators. Since the compiler will not have access to the code of external library functions, the only way in which information of non-portable calls may be provided is through the use of the function catalog. The catalog includes, among others, information about which function calls must be re-executed when restarting the application. Upon discovery of a non-portable call, the CPPC compiler performs the transformation depicted in Figure 4. It inserts a parameter registration for each input or input-output parameter passed to the call. The control flow information is also available through the function catalog. 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 control flow reaches a checkpoint. The parameter value, however, is stored in volatile memory when the `COMMIT REGISTERS` operation is invoked, and included in all subsequent checkpoint files. Upon restart, the call will be re-executed using the same parameter values as in the original execution. The compiler also adds flow control code to ensure that the program executes the non-portable block and is directed to the next restart-relevant block (RRB, see Section II-E) after executing it.

When the control flow reaches the non-portable block shown in the figure, values for `color` and `key` will be recovered through the parameter registrations inserted. The `comm` variable will be either a basic MPI communicator or will have been previously recovered through a re-execution of non-portable code. 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 communicator will be semantically correct in the new execution environment.

E. Putting it all together: restarting an execution

As previously mentioned, the code inserted by the CPPC compiler does not only create checkpoints during a regular execution, but is also in charge of consistently recovering the application state should a failure occur. Every CPPC application is divided into blocks of code that are relevant during application restart and blocks which are not. State recovery is accomplished through the sequential execution of restart-relevant blocks (RRBs), starting at the application entry point and up to the checkpoint location where the state file used for restarting the execution was created during the original run. Execution of blocks of code that are not restart-relevant is skipped. Figure 5 shows the typical structure of a CPPC application. Without loss of generality, the figure assumes that there is a single checkpoint in the application, inserted into function `f_n`.

The fundamental restart-relevant constructs are non-portable calls, variable registrations, and checkpoints. The CPPC compiler divides the application into structures formed by a block of non-relevant code, a jump target, a block of restart-relevant code, and a "conditional jump" to the next jump target, which will be placed right before the following RRB. A conditional jump is also inserted at the beginning of each instrumented function to correctly direct the execution flow when that function is reached. Conditional jumps are only taken during an application restart. In this way, after a failure, CPPC is able to re-execute only relevant parts of the code, skipping the non-relevant bits. The pieces of application code that are executed during a restart are marked in gray in the figure.

The skeleton shown in Figure 5 illustrates the concepts here described. It consists of $n+1$ nested functions such that `main` calls `f_1`, and each `f_i` performs a call to `f_(i+1)` in turn. A checkpoint is placed inside `f_n`. Upon restart, the conditional jump at the beginning of `main` would execute the RRBs up to the call to `f_1`, which would be performed as in a regular execution, portably recovering a part of the original application stack. Previous to the call, values of live variables in scope will be recovered through variable registration calls. In the figure dashed lines are drawn between each block of registers and the section of the code which is analyzed to determine the variables to be registered. Variable registrations and the stack areas they affect are marked with a striped background. Once inside `f_1`, the initial conditional jump would direct the execution towards its first RRB. Execution would eventually arrive at the call to `f_2`. The process repeats until, in the end, control reaches the checkpoint in `f_n`. At this point, CPPC detects that the execution is at the location where the original state file was created. The stack structure has been recovered, as well as all live variables. CPPC deactivates the conditional jumps, and thus execution resumes normally.

Note that this scheme is generalizable to any number of

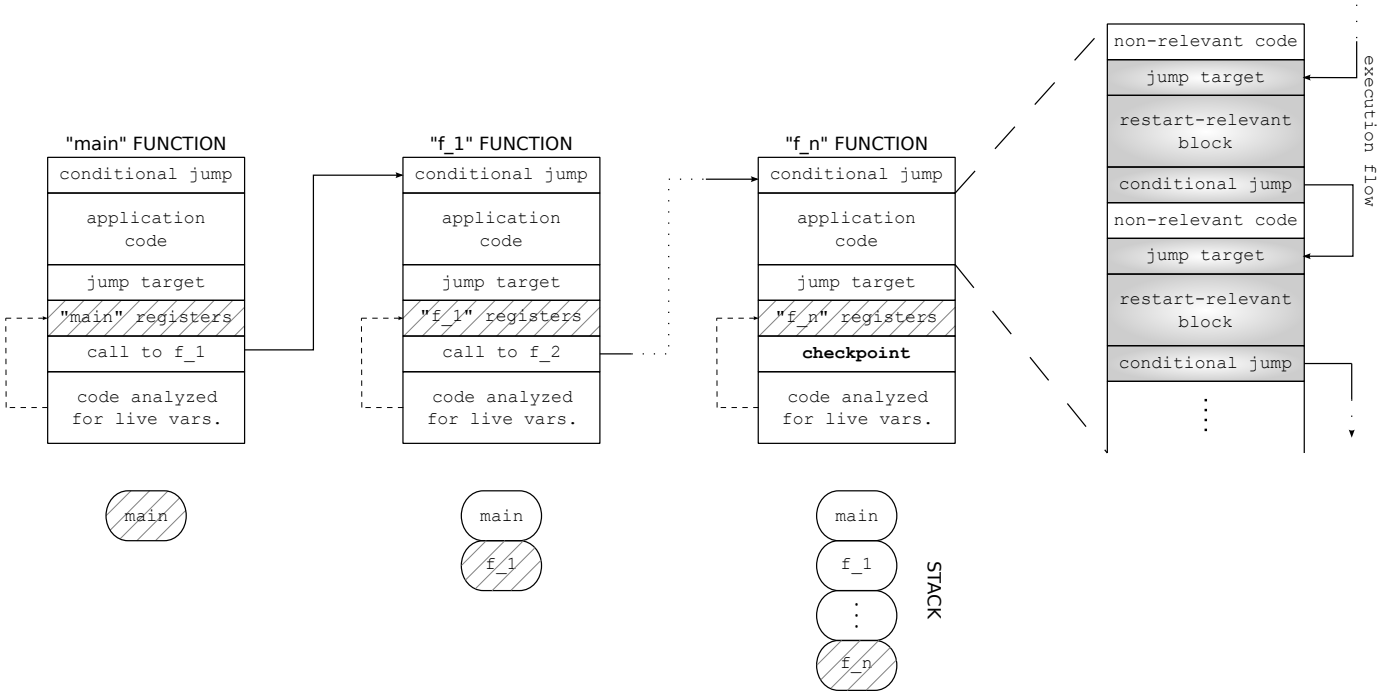

Fig. 5. Basic structure of CPPC-instrumented code

checkpoints arbitrarily spread among any number of (potentially nested) functions.

F. Dealing with Fortran 77 codes

Besides C codes, the CPPC framework also targets scientific codes written in Fortran 77 (F77), not natively supported by the Cetus infrastructure. This section describes the required Cetus extensions for supporting F77 applications. The first step was to write an F77 Antlr grammar in order to create the Cetus IR for these applications. The basic idea behind this extension was to reuse as much as possible the original Cetus IR, which enables the reuse of the C transformation codes. After transformations are performed to the IR, a back-end is in charge of rewriting the IR back to F77 code.

Cetus IR, however, is not 100% Fortran-compatible. Some F77 constructs can be directly mapped to existing IR classes, while others require new ones to be added. In particular, the following F77 constructs are represented using IR extensions:

- Descendants of `cetus.hir.Declaration`: `COMMON` blocks; `DATA`, `DIMENSION`, `EXTERNAL`, `INTRINSIC`, `PARAMETER`, and `SAVE` declarations.
- Descendants of `cetus.hir.Literal`: `DOUBLE` literals.
- Descendants of `cetus.hir.Specifier`: `COMPLEX`, `DOUBLE COMPLEX`, `ARRAY(lbound,ubound)`, and `CHARACTER*N` (string) specifiers.
- Descendants of `cetus.hir.Statement`: Computed `GOTO`s, `FORMAT` statements, Fortran-style `DO` loops, and Implied `DO` loops.
- Descendants of `cetus.hir.Expression`: Expressions appearing in `FORMAT` statements, substring expressions, IO function calls.
- Extensions to `cetus.hir.UnaryOperator`: `LABEL_ADDRESS` operator (`&&`).
- Extensions to `cetus.hir.BinaryOperator`: `F_POWER` (`**`), and `F_CONCAT` (`//`) operators.

In order to reuse transformation code as much as possible, all analyses are written using a template method design pattern. The transformation steps that differ depending on the source code are implemented in subclasses.

III. Concluding remarks

Cetus was used for implementing the CPPC compiler as a natural successor to SUIF [8]. The closed source nature of the SUIF front-end made it a poor choice for a research project such as CPPC, and a decision was made to port the already existing infrastructure to a more transparent framework. Cetus is an attractive choice which sports the following competitive advantages:

- It is implemented in Java. This provides almost limitless portability of the developments made.
- Its front-end is based on an open parser, Antlr. This enhances the creation of new front-ends by following the same design principles used in the original C parser.
- The code is completely open. This allows to perform the necessary transformations on top of each release distribution. For instance, in order to implement a Fortran back-end it is necessary to rewrite some parts of the standard Cetus printing system.
- Cetus uses a high-level representation, which closely resembles the original code. This allows for the implemented analyses to access information which is lost on lower-level IRs, such as the original array/pointer representations.

We have recently implemented a prototype version of the CPPC compiler on top of LLVM [9]. The main reason for this change would be the superior performance offered by an industry-level compiler, while maintaining a simple and very well documented IR (as opposed to other alternatives such as GCC). During the use of the LLVM infrastructure, the following advantages have been identified:

- An LLVM procedure consists of a control flow graph of basic blocks, each of which contains a series of ordered statements such that each one always dominates the subsequent ones. This is a more natural representation that a compound statement containing a series of statements that may or may not introduce different control flow paths.
- The LLVM IR is kept extremely simple. In contrast, some parts of the Cetus IR are redundant and quite obscure (e.g. IDExpression vs. Identifier vs. Name).
- LLVM makes no difference between statements and values at the IR level. Each statement may produce a value that might be used in dominated statements. The same IR object is used to represent the statement inside a basic block and the uses of its produced value in different instructions. This results in a very lightweight IR.

The main disadvantage of the LLVM infrastructure is that its IR is a low-level version of the original code. It is possible to output a transformed C version of the code, but it will be a similarly low-level version of the original code. Comparisons between the original and the output code are therefore quite hard to perform. This is something that may pose a problem during the initial, prototyping stages of an optimizing or parallelizing compiler.

ACKNOWLEDGMENTS

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

REFERENCES

[1] E. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, 2002.

[2] C. Dave, H. Bae, S. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," *IEEE Computer*, vol. 42, no. 12, pp. 36–42, 2009.

[3] G. Rodríguez, M. Martín, P. González, J. Touriño, and R. Doallo, "CPPC: A compiler-assisted tool for portable checkpointing of message-passing applications," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 749–766, 2010.

[4] G. Gibson, B. Schroeder, and J. Digney, "Failure tolerance in petascale computers," *CTWatch Quarterly*, vol. 3, no. 4, pp. 4–10, 2007.

[5] G. Rodríguez, M. Martín, P. González, and J. Touriño, "A heuristic approach for the automatic insertion of checkpoints in message-passing codes," *Journal of Universal Computer Science*, vol. 15, no. 14, pp. 2894–2911, 2009.

[6] ——, "Analysis of performance-impacting factors on checkpointing frameworks: the CPPC case study," *The Computer Journal*, 2011.

[7] J. Plank, M. Beck, and G. Kingsley, "Compiler-assisted memory exclusion for fast checkpointing," *IEEE Technical Committee on Operating Systems and Application Environments*, vol. 7, no. 4, pp. 10–14, 1995.

[8] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, E. Bugnion, and M. Lam, "Maximizing multiprocessor performance with the SUIF compiler," *IEEE Computer*, vol. 29, no. 12, pp. 84–89, 1996.

[9] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis," in *Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO'04)*, San Jose, CA, USA, 2004, pp. 75–88.