International Journal of Parallel Programming manuscript No.
(will be inserted by the editor)

Compiler-assisted checkpointing of parallel codes

The Cetus and LLVM experience

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

Received: date / Accepted: date

Abstract With the evolution of high-performance computing, parallel applications have developed an increasing necessity for fault tolerance, most commonly provided by checkpoint and restart techniques. Checkpointing tools are typically implemented at one of 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 transformations required to enable automatic checkpointing of parallel applications in the CPPC application-level checkpointing framework. These transformations have been implemented on two very different compiler infrastructures: Cetus and LLVM. Cetus is a Java-based compiler infrastructure aiming to provide an easy to use and clean IR and API for program transformation. LLVM is a low-level, SSA-based toolchain. The fundamental differences of both approaches are analyzed from the structural, behavioral and performance perspectives.

Keywords Fault tolerance · checkpointing · parallel programming · message-passing · compiler support · Cetus · LLVM

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

G. Rodríguez · M.J. Martín · P. González · J. Touriño · R. Doallo
Computer Architecture Group, Department of Electronics and Systems, University of A Coruña, Spain
E-mail: grodriguez@udc.es

1 Introduction

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 [9], 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 compiler was originally written using the Cetus compiler infrastructure [8]. It uses a high-level representation of the code, retaining almost completely the original syntax. Afterwards, the compiler was ported to LLVM [12]. The latter uses a low-level set of nodes, close to assembly language. The initial motivation for this reimplementation was a desire to leverage potential performance advantages using LLVM. As will be shown later, this was achieved only in part. This work describes the implementation of the compilation techniques for the automatic insertion of checkpointing instrumentation, paying attention to the differences between the Cetus and LLVM implementations in their functionality, performance, and simplicity.

This paper is organized as follows. Section 2 details the design of CPPC and motivates the use of compilation techniques. Section 3 covers related work. Section 4 details the implementation of the compilation analyses required to instrument checkpointing with CPPC. Section 5 presents the experimental results comparing Cetus and LLVM, and Section 6 concludes the paper.

2 The CPPC checkpointing framework

This section summarizes various fundamental design aspects of the CPPC framework in order to introduce the necessity for compilation techniques: the necessity for portability; the selection of the application state that needs to be stored into state files to achieve a correct restart; and how CPPC operates to achieve consistent operation without runtime communications in SPMD applications. For an in-depth description of CPPC the reader is referred to [23].

2.1 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 4.4 and 4.5.

2.2 Relevant state selection

The solution of large real scientific problems requires the use of large computational resources, both in terms of CPU and memory. For this reason, 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 saving the entire application state, leads to a large storage size, frequently becoming impractical [10]. Besides, the size of the state files is one of the most significant performance-impacting factors when 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*.

2.3 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 [21]. 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*. Opposed to this concept, an *unsafe region* R will be comprised of the code in between two communication statements.

3 Related work

Checkpointing techniques appeared in the late 70s as operating system services, usually focused on recovery of sequential applications. Examples are KeyKOS [11], which performed system-wide checkpointing, storing the entire OS state, and Sentry [24], a UNIX-based implementation which performed

checkpointing and journaling for logging of non-deterministic events on single processes. Sprite [15] dealt with the process migration aspect of checkpoint-and-restart recovery for shared memory applications to idle machines. Being implemented into the OS, these checkpointing solutions were completely ad-hoc, with a lack of emphasis on portability. Their approach to operation efficiency was based on achieving good I/O performance for storing the whole computation state, instead of reducing the amount of data to be stored.

The first obvious disadvantage of OS-based implementations is the hard dependency between fault tolerance and the operating system of choice. Checkpointing facilities, which were a common feature in earlier operating systems, gradually disappeared in the early 90s and were unavailable for popular environments such as UNIX, SunOS or AIX. The desire for flexible solutions which could operate in different environments motivated the emergence of application level solutions (as opposed to system level solutions). In these tools, fault tolerance is achieved by compiling the application program together with the checkpointing code, usually found in a separate library. Checkpointing solutions in this period were still transparent, storing the entire application state. Not being implemented inside the kernel they had to solve important problems when manipulating OS-dependent state. Examples are restoring process identifiers or tracing open files. Also, they had to figure out ways to recover the application stack or heap. These issues made their codes still very dependent on specific operating system features. Usually, this forced developers to restrict the type of OS facilities used by the programs to be checkpointed. Examples of application level, transparent tools include Libckpt [18] and CATCH GCC [13] (a modified version of the GNU C compiler).

Also in the mid-90s some non-transparent solutions tried to apply checkpointing to distributed platforms. Their fundamental drawback was the lack of common ground regarding the interface for interprocess communication, which made these solutions tied to a specific and non-standard interface. Examples of these frameworks are Calypso [3] and extensions to Dome (Distributed Object Migration Environment) [4]. In both cases the programming language used was an extension of C++ with non-standard parallel constructs.

The adoption of MPI as the de-facto standard for parallel programming motivated the appearance of many MPI-based checkpointing tools in the last decade. At first, these used the transparent application level approach, sharing the same drawbacks as their uniprocessor counterparts: lack of data portability and restriction of supported environments, which here refers to the underlying MPI implementation. In fact, checkpointers in this category are generally implemented by modifying a previously existing MPI library. Examples of these types of checkpointers are MPICH-GF [26] and MPICH-V2 [5], both implemented as MPICH drivers, thus forcing all machines to run this specific implementation.

More recently, heterogeneous supercomputing systems have introduced new checkpointing constraints, requiring both data and underlying system portability. To accomplish this, checkpointing solutions must be implemented at a higher level of abstraction and require modifications to the source code of

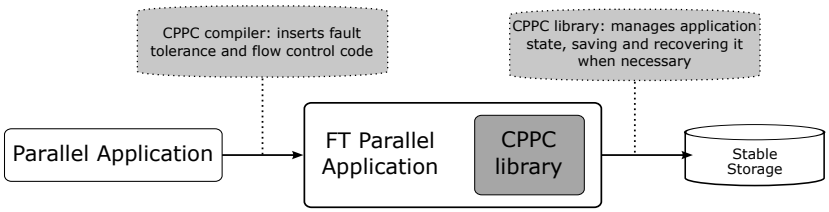

Fig. 1 CPPC framework design

the application. If these modifications are manually performed an undesired burden is placed upon users, who have to undertake tedious data flow analysis and code reengineering. Application level checkpointers have taken to using compilation techniques to free users from these tasks. Most of them, however, leave the actual checkpoint locations to be manually marked by the user. Porch [19] and C^3 [6] are compiler-assisted systems for sequential and parallel applications, respectively, in which the user inserts a call to a checkpoint routine before using the compiler to insert checkpointing instrumentation. These checkpoint calls will only trigger an actual checkpoint according to a frequency timer. These "potential checkpoints" were originally introduced by CATCH GCC, which automated their insertion by introducing a potential checkpoint at the beginning of subroutines and at the first line inside a loop. This checkpoint placement tried to guarantee that potential checkpoint calls were executed often enough as to provide a checkpointing frequency reasonably similar to the desired one. This approach cannot be followed when using a spatial coordination protocol based on safe points such as the one used by CPPC. In this situation, checkpointing frequencies are not defined as a function of time, due to the need to statically coordinate all processes independently of how long they take to progress through the application code. Instead of statically detecting safe points, a checkpointer for parallel applications may employ a coordinated runtime protocol, such as the distributed snapshots [7], to achieve consistency. C^3 uses information piggybacked into sent messages to articulate such a protocol. In this way, every message being sent has to be intercepted and modified.

4 Compilation analyses

In early stages of CPPC, the user was responsible for inserting compiler directives to guide the operation of the runtime library [20]. Currently, all analyses and code transformations are transparently applied by a compiler that translates the application source files into derived code with added checkpointing capabilities. The global process is depicted in Figure 1.

The most relevant transformations applied by the compiler are, in this order: (1) the communications analysis required in order to automatically detect safe points where 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 4.1); which ones generate non-portable state that must be recovered through code re-execution (as shown in Section 4.4); 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 4.3).

The following subsections describe the fundamental transformations performed by the compiler, the main differences between the Cetus and the LLVM implementation, as well as Cetus extensions required for supporting Fortran 77 codes.

4.1 Communications analysis

Statically determining the communications that are performed during runtime in an SPMD application is an undecidable problem. In the general case, message-passing applications may present irregular communication patterns (if sources, destinations, or communication order depend on the input data) or nondeterministic communications (if wildcard receives are used). However, an important subset of scientific applications employs regular communication patterns, which makes the problem decidable by a static code analysis. First, the solution for a regular application is presented. Afterwards, a conservative solution for irregular/nondeterministic codes is proposed.

Without loss of generality, we can assume that SPMD applications with regular communication patterns employ some kind of topological abstraction to describe the virtual layout of the processes participating in the parallel execution. Each process is identified by its set of coordinates in the virtual topology $(c_1, c_2, \ldots, c_k)$. These coordinates will be employed for determining the sources and destinations of the communications required for executing the application code. Since communications are regular, these may not be derived from any dynamic input data or nondeterministic function call, and must therefore be encoded into either the application code or the rank of the process in the parallel execution. The same applies for the variables which encode the implicit communication order (message tags). In this situation, a constant propagation analysis can be employed to discover the specific values used in each of the communication calls. After discovering these values, two communications 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` and no other receive fulfills condition 1.

Our constant propagation algorithm follows the basic iterative algorithm for general data flow frameworks [1], with two particularities. First, it is not necessary to propagate all known values in the application. Only variables used for calculating message sources, destinations and tags are first-order communication-relevant variables. Variables involved in the calculation of first-order communication-relevant variables are second-order communication-relevant variables. The discovery of all n^{th} order communication-relevant variables is an iterative process, and ends when the set is not modified in a given iteration.

The second particularity is related to the fact that, in the general case, an SPMD variable is multivalued, that is, it may have different values for different processes. In this situation, each communication-relevant variable in the original code must be considered to spawn different actual variables, one for each process in the execution. As such, the set of data flow values to be calculated by this constant propagation step is a product lattice with one component for each communication-relevant variable for each process. For this reason, the number N of processes involved in the execution of the code must be known statically. Each variable x gives birth to N different variables:

$$x_i,\ 0 \leq i < N$$

Finally, it has been assumed that communications depend on the coordinates of each process in the virtual topology of processors, and that, in turn, these depend on the rank assigned to the process by the communication framework. The transfer function f associated to each statement s in the program must be modified as shown in Figure 2 to capture rank assignment operations. In the figure, f_s is the transfer function for statement s, and m represents a map which associates each variable v with its known constant value during the execution, $m(v)$. Given any input map, the transfer function returns a map m' such that $m' = f(m)$.

Although both Cetus and LLVM include constant propagation analysis, due to the changes in the data flow set and the transfer function neither of these implementations can be used off-the-shelf. A way to employ the built-in capabilities would be to instrument the code to force the analysis to work as shown in Figure 2 by changing calls to process identifier-returning functions with simple assignments. However, this would require one analysis pass for each of the N processes to execute the application.

For the aforementioned reasons, constant propagation is implemented from scratch. There are no significant differences between the Cetus and the LLVM versions of the code. Since constant propagation is a forward analysis, it may

1. If s is not an assignment statement, then f_s is the identity function.
2. If s is an assignment to variable x_i, them $m'(v) = m(v)$, for all variables $v \neq x_i$, provided one of the following conditions holds:
   (a) If the right-hand-side (RHS) of the statement s is a constant c, then $m'(x_i) = c$.
   (b) If the RHS is of the form $y + z$, being $+$ any arbitrary binary operator, then:

$$m'(x_i) = \begin{cases} m(y_i) + m(z_i) & \text{if } m(y_i) \text{ and } m(z_i) \text{ are constant values} \\ \text{NAC} & \text{if either } m(y_i) \text{ or } m(z_i) \text{ is NAC} \\ \text{UNDEF} & \text{otherwise} \end{cases}$$

   (c) **If the RHS is a call to a function returning a process identifier i in the context of the message-passing framework, then $m'(x_i) = i$.**
   (d) If the RHS is any other expression (e.g. a function call or an assignment through a pointer), then $m'(x) = \text{NAC}$.

Fig. 2 Transfer function for the modified constant propagation algorithm employed for static communication analysis. Modifications with regard to the original proposed transfer function are shown in bold. The special values NAC and UNDEF denote, respectively, that a given variable is *Not A Constant*, and that its value has not yet been defined in terms of the constant propagation process.

be performed together with the communications matching in the same compiler pass, starting at the execution root and analyzing 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 presenting the modified algorithm for constant propagation it was not discussed how to work in a general, interprocedural code. 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 and adapting the map m of variable values to reflect aliases stemming from argument passing. The compiler will also cache separately the communications issued inside the callee. If a procedure p does not modify any communication-relevant variable, that is, the transfer function f_p is the identity, then this cache may be used when a new call to the procedure is found without re-analyzing the procedure code, but only substituting the communication arguments with the values present in the map m of variable values. Figure 3 presents a pseudocode of this analysis.

When dealing with applications featuring ambiguous communications the solution proposed above might be unable to find suitable safe points. Although such situations are uncommon, as shown in the experimental assessment of the tool [22], a feasible solution is proposed in the next subsection.

```
buffer: communications buffer
m: map of variable values
procedure communications_analysis
 detect communication relevant variables
 buffer ← ∅
 foreach variable v in p do
  insert pair (v, UNDEF) into m
 done
 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
    f_p is the identity function
  /* 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 is a call to a procedure p' then
   call analyze_procedure( p' )
  else
   m ← f_s(m) /* apply transfer function */
  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. 3 Pseudo-code of the communications analysis

4.1.1 Dealing with irregular and nondeterministic codes

If the communication patterns of an application are derived from input data, then our assumption that communication-relevant variables may only be derived from constants and coordinates in a virtual topology is not applicable. It would be necessary to have knowledge of the input data to be used, and even if that were possible checkpointing would have to be instrumented in a different way for each input set. Under these circumstances a conservative solution is adopted: each communication is considered to match a set of potential peers. For a given communication statement c, there is a set of potential matches m_i, $0 \leq i < M$. Each single match $c \leftrightarrow m_i$ determines an unsafe region in the code, R_{c,m_i}. The unsafe region associated to communication statement c can be informally expressed as $R_c = \bigcup_i R_{c,m_i}$. Note that, since all unsafe regions of the form R_{c,m_i} have at least statement c in common:

$$\nexists i, j : \ 0 \leq i, j < M \,/ R_{c,m_i} \cap R_{c,m_j} = \emptyset$$

and consequently R_c must be a continuous region. The same approach should be followed when considering communications with wildcards. Shires et al. proposed an algorithm for program flow graph construction capable of determining conservative relationships between communication statements [25].

4.2 Checkpoint insertion

Application-level checkpointing tools often require users to mark places where checkpoint calls are to be inserted. The "potential checkpoints" approach was introduced in Section 3. Not only should checkpoints be placed at locations in the code where calls are executed frequently. It is also important that checkpointing is triggered from the user code, and not when the execution is inside an external library call. This tries to ensure that no opaque, internal state to the library is left unsaved, causing restart errors. When using runtime consistency protocols it is only necessary to insert frequent calls to the potential checkpoint call to ensure that the state is saved with the specified frequency. This is not possible using static coordination: checkpoint calls where the state will be effectively saved need to be previously agreed upon by all processes. In this context, a valid approach consists in statically detecting those loop nests in the code that perform the core of the computation, statically inserting potential checkpoint calls inside them. During runtime, a checkpointing frequency may be defined in terms of number of loop iterations. This frequency may be dynamically adapted by means of lightweight, asynchronous protocols during runtime.

Statically selecting the loop nests that perform the core of the computation presents several challenges. The main one is that it is not possible to accurately predict the execution time of a section of code without precise knowledge of the hardware which is to execute it. To overcome this issue, heuristic cost analyses are employed, using computational metrics to discover critical sections of

code. The sections of code are considered relevant depending not on the time they will take to execute, but on how much computational load they pose when compared to other parts of the application. Thus, the problem of actually estimating execution time is abstracted and converted into a comparison between estimated computational loads of all the loop nests in the application.

The most simple computational metric for a loop nest is the number of instructions it contains. However, this does not take into account the data access pattern of the loop. For instance, the computational load of a loop performing an irregular reduction on a sparse array cannot be compared to that of a loop initializing an array to zero, even if their source codes might contain the same approximate number of instructions. For this reason, the developed heuristic function also takes into account the number of variables a statement accesses. A formal definition of the heuristic function in use by CPPC follows.

Let l be a loop in L, the loop population of the application P, and i and a two functions that return the number of instructions and variable accesses in a given block of code, respectively. Let us define $I(l) = i(l)/i(P)$ 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 computational load value associated to each loop l is calculated as:

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

Equation 1 multiplies $I(l)$ 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. The closer to zero the value, the higher the computing time estimated for the loop. After calculating $h(l)$, $\forall l \in L$, thresholding methods are applied to select the set of loop nests where checkpointing is required [21]. These methods are based on both the shape of the $h(l)$ function and its first and second derivatives.

The cost estimation is performed interprocedurally and traversing the 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. For the Cetus IR, the transfer functions $i(s)$ and $a(s)$ involved in the calculation of $h(s)$ for a given statement s are as follows:

1. If s is a declaration statement, $i(s) = a(s) = 0$.
2. If s is a simple statement (leaf node in the IR):
   - $i(s) = 1$
   - $a(s) =$ #variable accesses in s
3. If s is a call statement to function f; let $x \in \{i, a\}$:

$$x(s) = x(s_f)$$

where s_f is the compound statement that represents the body of f.

4. If s is a conditional statement gating the execution of n statements $s_j,\ 0 \leq j < n$; let $x \in \{i, a\}$:

$$x(s) = \frac{\sum_{j=0}^{n} x(s_j)}{n}$$

5. If s is a compound statement with children statements $s_j,\ 0 \leq j < m$; let $x \in \{i, a\}$:

$$x(s) = \sum_{j=0}^{m} x(s_j)$$

When implementing this analysis in LLVM, the fact that LLVM IR uses three-address code has to be taken into account. In this kind of IR, the number of accesses per instruction remains constant for most instructions. In this situation, $a(l) \simeq 3 \cdot i(l)$, and considering memory accesses in the loop is unnecessary. In this situation, the heuristic computation value $h(l)$ is a constant displacement away from the value $h'(l)$ defined as:

$$h'(l) = -log(I(l))$$

Note that if $a(l) \simeq 3 \cdot i(l)$, $h'(l)$ has approximately the same shape and first and second derivatives than $h(l)$, and therefore it is a good and simpler substitute. The LLVM analysis is implemented following the same basic principles than the Cetus one, but using $h'(l)$ instead. As such, it is only necessary to calculate the number of instructions $i(l)$ inside each loop nest applying the transfer functions previously described. A qualitative comparison between both implementations can be found in Section 5.

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

Once the loops in which checkpoints are to be inserted are identified, the results of the communications analysis are used to insert a checkpoint at the first available safe point in each selected loop nest.

4.3 Registration of restart-relevant variables

As described in Section 2.2, 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 upon execution of statement 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 [17].

This section is organized as follows: Section 4.3.1 details how live variables are intraprocedurally calculated by the CPPC compiler, and why this

```
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. 4 Pseudo-code of the live variable analysis

calculation is different from the traditional one; Section 4.3.2 explains how the intraprocedural live variable analysis is used to insert variable registrations in an interprocedural context.

4.3.1 Live variable analysis

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

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

where $LV_{out}(s)$ is the set of live variables after executing statement s, and $USE(s)$ and $DEF(s)$ are the sets of variables used and defined by s, respectively. This analysis is traditionally performed backwards, being $LV_{out}(s_{end}) = \emptyset$, and s_{end} the last statement of the code. This computation calculates the set of input and output live variables for every basic block in the code.

For the purpose of checkpointing, it is not required to compute the set of live variables for the entire application, but only for those code regions that are executed after restarting from a previously stored checkpoint. Thus, finding live variables on demand only for the relevant regions of code improves performance.

The Cetus version of the compiler does not use the basic block abstraction, but works directly on Cetus IR instead. The LLVM version takes advantage from the identification of statements and references to their results in LLVM IR to perform a fast traversal of the code. CPPC annotates each statement s with its corresponding $USE(s)$ and $DEF(s)$. When it needs to obtain the set of live variables at a statement s it traverses all statements from s up to s_{end}. The pseudo-code for the live variable calculation is shown in Figure 4.

The LLVM version of this analysis works in exactly the same way as the Cetus one. No advantages are obtained from the lower-level representation in SSA form.

4.3.2 Variable registration

Before each checkpoint statement c_i, the compiler inserts annotations to register 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 inserts "unregister" annotations for the variables in the set $LV_{in}(c_{i-1}) - LV_{in}(c_i)$, the set of 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:

1. For a checkpoint statement c_i contained in 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.
2. 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.
3. 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.

Proof of Correctness: Let us consider a variable v which appears in the statements contained in B_{c_i}, and let $s_v \in B_{c_i}$ be the statement where it first appears. There are three different cases to analyze: v can either be an `input`, an `output`, or an `input-output` variable for s_v. Using the definition of the live variable analysis:

- `input`: $v \in USE(s_v) \wedge v \notin DEF(s_v)$

  $$USE(s_v) \subset LV_{in}(s_v) \Rightarrow v \in LV_{in}(s_v)$$

  Since s_v was the first appearance of v in B_{c_i}, there is no previous statement which defines its value, meaning that v belongs to the live variable set for every statement before s_v:

  $$\nexists i / (v \in DEF(s_i)) \wedge (s_i < s_v) \Rightarrow v \in LV_{in}(s_j), \forall j / s_j < s_v$$

In particular, since $c_i < s_v$: $\mathbf{v} \in \mathbf{LV_{in}(c_i)}$.
A register will be inserted before s_v when analyzing its containing procedure. This register will generate the value of v when restarting the application.

- `output`: $v \notin USE(s_v) \wedge v \in DEF(s_v)$
  In this case, it is guaranteed that $v \notin LV_{in}(s_v)$.
  Since s_v was the first appearance of v in B_{c_i}, there is no previous statement which uses its value, meaning that v does not belong to the live variable set for every statement before s_v:

$$\nexists i / (v \in USE(s_i)) \wedge (s_i < s_v) \Rightarrow v \notin LV_{in}(s_j), \forall j / s_j < s_v$$

  In particular, since $c_i < s_v$: $\mathbf{v} \notin \mathbf{LV_{in}(c_i)}$.
  No register will be inserted; v's value will be generated upon reaching s_v, therefore defining it.
- `input-output`: $v \in USE(s_v) \wedge v \in DEF(s_v)$
  This case is similar to the input one.

■

Proof of Termination: The algorithm terminates if all statements contained into B_{c_i} are analyzed. B_{c_i} is a finite set, since its maximum number of elements is equal to the total number of statements in the code being analyzed. The evolution of the cardinality of the unanalyzed set of statements is as follows:

- In the first phase of the algorithm (the analysis of procedure f), the statements in $B^f_{c_i}$ are analyzed. Either $s_1 \in f$ and $\#L^f_{c_i} < \#B_{c_i}$, or c_i is the last statement in f and $\#L^f_{c_i} = \#B_{c_i}$.
- In each of the subsequent phases, the set of unanalyzed statements can be written as:

$$L^{p_n}_{c_i} = \{s^{p_n}_1, \ldots, s^{p_n}_m\} = B_{c_i} - (\bigsqcup_{j=1}^{n} B^{p_j}_{c_i})$$

  where each p_j is the procedure being analyzed in phase j. In phase n, the algorithm analyzes the procedure containing the statement $s^{p_n}_1$. Therefore, at least one statement is analyzed, and $\#L^{p_{n+1}}_{c_i} < \#L^{p_n}_{c_i}$.

Since all the statements must be contained in a procedure in order to be in the initial B_{c_i} and this is a finite set, the termination of the algorithm in a finite number of steps is guaranteed.

■

The live variable analysis takes into account interprocedural data flow. Upon finding a call to a procedure h, the compiler performs an on-demand analysis of the code of h. The data flow 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

```
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. 5 Pseudo-code of a non-portable procedure call transformation

call, either as part of the execution of the code or by means of a variable registration. To avoid this default behavior data flow information may be included in the function catalog.

In this way, the set of locally live variables in each procedure is recovered inside the procedure itself. Figure 6 graphically depicts how the liveness analysis is applied to determine the variables to be registered for a checkpoint inserted at an arbitrary call stack depth. Further details about state recovery during application restart are given in Section 4.5.

4.4 Identification of non-portable functions

As stated in Section 2.1, CPPC recovers non-portable parts of the application state through the re-execution of the code creating such opaque state (e.g. 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 5. It inserts a parameter registration for each input or input-output parameter passed to the call. The data 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 4.5) after executing it.

When the flow of control 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 predefined 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

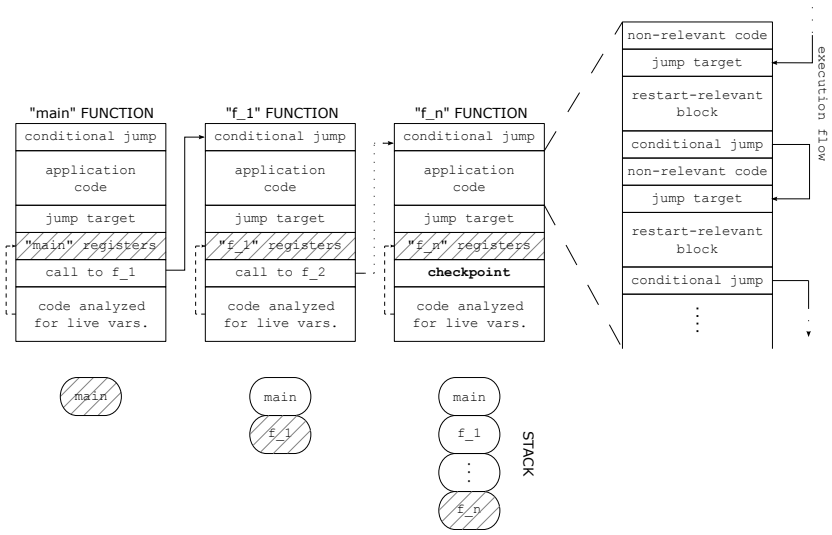

Fig. 6 Basic structure of CPPC-instrumented code

run, but the outcome communicator will be semantically correct in the new execution environment.

This analysis is conceptually equivalent in both the Cetus and LLVM versions of the compiler, and differs only in basic implementation details.

4.5 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 6 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 6 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 approach is generalizable to any number of checkpoints arbitrarily spread among any number of (potentially nested) functions. The restart protocol for a CPPC application remains the same for both the Cetus and LLVM versions of the CPPC compiler.

4.6 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. There are already available front ends that translate an F77 application into LLVM IR. This section describes the required Cetus extensions for supporting F77 applications. The first step was to write an F77 parser to generate 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.

Application	SLOCs	#L	Checkpoint (file:line)
BT	3650	25	`bt.f`:179
CG	1044	13	`cg.f`:441
EP	180	4	`ep.f`:189
FT	1269	20	`ft.f`:159
IS	672	6	`is.c`:976
LU	3086	35	`ssor.f`:78
MG	1618	12	`mg.f`:245
SP	3148	25	`sp.f`::150

Table 1 *Summary of test applications*

- 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.

5 Experimental results

This section compares the performance and results of the two implementations of the analyses presented in this paper: the Cetus- and the LLVM-based implementation. For this purpose, the eight applications of the NPB-MPI v3.1 benchmarks [14] were used. The NPB are well-known and widespread applications that provide a de-facto test suite. All the NPB applications are Fortran codes, except for IS which is written in C. Table 1 shows a summary of the characteristics of the test applications, including the number of source lines of code (SLOCs), number of loop nests in the code ($\#L$), and place (file and line number of the source code) where a checkpoint was manually inserted during the assessment of the test applications.

Besides providing performance figures, this section intends to compare the relative performance of Cetus (based on Java) and LLVM (a C++ infrastructure) for the different stages of the compilation pipeline. It is to be expected that a low-level, SSA representation such as LLVM IR will adapt better to the data flow analyses that are required in order to instrument checkpointing. Experiments were performed on a desktop computer, an Intel Core2 Duo at 3 GHz with 2 GB of RAM. Cetus 1.3 was ran on Sun JDK 1.6.0.26, using 512 MB of memory allocation pool. LLVM 2.9 was used for the LLVM tests. The performance results for the compilation analyses are provided in Table 2.

	Parsing		Communications		Checkpoints	
Application	*Cetus*	*LLVM*	*Cetus*	*LLVM*	*Cetus*	*LLVM*
BT	5828	724	2713	200	916	40
CG	803	101	6762	104	70	2
EP	292	27	106	12	13	1
FT	1816	218	910	60	253	9
IS	766	33	1842	20	73	1
LU	2596	661	1518	200	416	123
MG	1821	364	7865	152	387	15
SP	2239	695	1718	348	778	51
	Registration		**Instrumentation**		**Linking**	
Application	*Cetus*	*LLVM*	*Cetus*	*LLVM*	*Cetus*	*LLVM*
BT	1839	2	1366	2	1364	7564
CG	433	1	145	3	200	3924
EP	222	1	48	2	80	88
FT	508	1	281	1	400	384
IS	119	1	131	1	80	96
LU	929	2	380	3	1180	3924
MG	702	2	420	2	604	1508
SP	2736	1	357	2	1304	13109

Table 2 Runtimes (ms) for the CPPC compiler analyses

Besides the times employed for the checkpointing analyses, this table includes the parsing and linking times, which will be taken into account to compare the total processing time of both toolchains. The remainder of this section is dedicated to a discussion of the obtained performance figures. When relevant, the qualitative results of the analyses are also discussed.

The first step in the compilation process is the parsing of the source code in order to generate the IR. The Fortran parser for Cetus was written by the authors (see Section 4.6), while the C parser is already included in the Cetus bundle. Clang 2.9 and llvm-gfortran 2.9 were used as parsers for LLVM. Given that parsing is a more or less straightforward operation of scanning the source code and creating an IR to represent it, the faster LLVM processing is a consequence of a fundamental difference between the toolchains: the Cetus parsers are written using ANTLR [16] and executed by a Java VM, while Clang and llvm-gfortran are C/C++ code that runs directly on the operating system. Judging by parsing time, it is to be expected that the Java execution will be an order of magnitude slower than the native one.

After parsing the code, the CPPC compiler proceeds to statically match communications. The results of this analysis will be later used during checkpoint insertion. The implementation takes advantage of the use-def chains available in LLVM to streamline the location of the statements that need to be analyzed. The result is that the LLVM version is an order of magnitude faster. Qualitatively, the results are correct and the same for both implementations, even for IS, which presents an irregular communication pattern.

After identifying safe points in the code, loop nests are ranked and those with higher estimated computational loads are selected for checkpoint insertion. While the running times for the LLVM analysis remain an order of magni-

Application	Extra checkpoints (file:line)
IS	`is.c`:425 `is.c`:396
SP	`exact_rhs.f`:23 `initialize.f`:45 `error.f`:26

Table 3 Extra checkpoints inserted by Cetus-CPPC

tude lower (see column labeled as "Checkpoints" in Table 2), the results of the analyses differ in this case. The LLVM version matches the desired checkpoint results shown in Table 1. The Cetus one inserts some extra checkpoints for loop nests in IS and SP, detailed in Table 3. These results are conceptually correct, meaning that all relevant checkpoints are identified, but not optimal, since non-relevant nests are checkpointed as well. This difference emerges from the abstraction levels of the two IRs. The heuristic computational load function $h(l)$ defined in Equation 1 is derived from the number of statements, $I(l)$, and memory accesses, $A(l)$. Using Cetus IR, $I(l)$ is closely related to the number of SLOCs, a measure of code complexity in terms of programmer effort. SLOCs do not necessarily provide a good estimation of computational effort. When using LLVM IR, much closer to assembly code, $I(l)$ is similar to the actual number of machine instructions involved in the execution of the loop code. As such, LLVM IR provides better support for the accurate calculation of $h(l)$. The authors are currently working on developing techniques for checkpointing insertion based on automatic recognition of computational kernels [2]. Kernel recognition benefits from working with a high-level IR such as Cetus IR, and may be used to estimate more accurately the relative computational load of a section of code.

For analyses which are purely data flow oriented, such as variable registration and code instrumentation, LLVM is two to three orders of magnitude faster than Cetus according to the results of Table 2. Besides the faster execution of C++, LLVM IR uses the same object to identify both a statement and a reference to its results, and provides use-def and def-use chains. These functionalities enable faster data flow analyses. During runtime, however, checkpoint files generated by Cetus are smaller than those generated by LLVM. The reason is that the llvm-gfortran front end introduces extra low-level variables to handle Fortran `COMMON` blocks that cannot be identified as non-live by the data flow analyses in CPPC. These get conservatively registered and stored into checkpoint files. The exception is IS, which is a C application. In this case, LLVM generates slightly smaller checkpoints due to scalar optimizations. Generated checkpoint sizes for the 'S' version of the NPB applications are shown in Table 4. The difference in size remains constant for different problem sizes.

After running the compilation analyses, the last step is to generate executable files from the instrumented codes. Measurements for this step are shown in Table 2 and labeled as "Linking". The output of the CPPC Cetus compiler are Fortran or C files instrumented for fault tolerance. The linking

	Checkpoint size (KB)	
Application	*Cetus*	*LLVM*
BT	1441.02	1550.68
CG	3050.48	3061.77
EP	1211.13	1238.07
FT	6225.75	14548.48
IS	1269.29	1243.38
LU	884.95	1052.61
MG	1251.64	1235.93
SP	1221.70	1243.38

Table 4 Size (KB) of the checkpoint files generated by the CPPC library for class 'S' NPB applications instrumented by the Cetus and LLVM versions of the CPPC compiler

	Total time	
Application	*Cetus*	*LLVM*
BT	13911	8559
CG	1713	546
EP	788	168
FT	3548	725
IS	1252	1683
LU	5919	5120
MG	4209	2119
SP	7929	14437

Table 5 Total processing times (ms)

time was calculated as the time it takes GCC 4.5.3 to generate an executable from these modified source files. For LLVM, the linking time is measured as the time for the execution of the `llvm-ld` command that takes the modified LLVM IR and converts it into a native executable. This is a costly operation, particularly for applications which span several source files. When the entire toolchain is taken into consideration, processing times for Cetus and LLVM are generally of the same order, with the fastest tool depending on the selected application. The total accumulated times are summarized in Table 5 for the reader's reference.

6 Concluding remarks

This work has focused on the implementation of the transformations performed by the CPPC compiler to provide fault tolerance for message-passing applications. The required analyses involve a communication analysis, a heuristic computational load estimation to determine places in the code that are appropriate for checkpoint insertion, a liveness analysis to discover the data that will be required when restarting an application, and the insertion of constructs to guide the execution flow during this operation.

Two equivalent implementations of these analyses have been studied, one using Cetus and the other on top of LLVM. Cetus is a compiler infrastructure characterized by its high-level IR and ease of implementation. It was

developed to provide a portable compiler infrastructure, multi-language support and to be extensible. LLVM was born as a research project to provide SSA-based compilation capabilities. By design, these tools have very different approaches, advantages and capabilities. Throughout the paper, several of these characteristics have been highlighted. A brief summary is included in this section for reference.

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 IR-API is simple and consistent. The learning curve of Cetus is purposely very steep, and it is an ideal tool for people with little compiler experience to write compiler passes.
- Cetus uses a high-level representation, which closely resembles the original code. This allows for the implemented analyses to access information which is lost in lower-level IRs, such as the original array/pointer representations. Additionally, the resulting code is similar to the source program, making it easy for a user to review and understand the steps involved in the transformations made.

In contrast, LLVM presents the following advantages:

- It presents good performance due to its C++ implementation.
- The IR is closer to the hardware. While this makes it very difficult to relate it to source code, it simplifies some analyses and makes it easy to relate the IR to what will be ultimately executed. The SSA representation, particularly the availability of use-def and def-use chains, greatly simplifies data flow analyses.
- LLVM makes no difference between expression statements and their values at the IR level. The same memory address may be used in dominated statements to indicate a def-use relationship. This results in a lightweight and very fast IR.

The disadvantages of both infrastructures are readily identifiable as the advantages of each other. On the one hand, some data flow and dependency analyses are hard to implement in Cetus, due to its high-level nature. On the other hand, LLVM decomposes some code concepts to the point of making them nearly irrecoverable (e.g. array indexes are translated to a series of displacements from the array's base address). While LLVM provides very fast IR traversal, it is also very easy to execute an instruction that leaves the IR in an inconsistent state, something that Cetus avoids by design. To summarize, LLVM may be better suited for implementing production-oriented compiler passes. Cetus is a good choice for rapid prototyping and experimentation with compilation techniques in research environments.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, & Tools, pp. 632–638. Pearson Education (2007)
2. Arenaz, M., Touriño, J., Doallo, R.: XARK: an extensible framework for automatic recognition of computational kernels. ACM Transactions on Programming Languages and Systems **30**(6), 32:1–32:56 (2008)
3. Baratloo, A., Dasgupta, P., Kedem, Z.M.: CALYPSO: A novel software system for fault-tolerant parallel processing on distributed platforms. In: Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing (HPDC-4), pp. 122–129 (1995)
4. Beguelin, A., Seligman, E., Stephan, P.: Application level fault tolerance in heterogeneous networks of workstations. Journal of Parallel and Distributed Computing **43**(2), 147–155 (1997)
5. Bouteiller, A., Capello, F., Hérault, T., Krawezik, G., Lemarinier, P., Magniette, F.: MPICH-V2: A fault-tolerant MPI for volatile nodes based on pessimistic sender based message logging. In: Proceedings of the 15th ACM/IEEE Conference on Supercomputing (SC'03), pp. 25–42 (2003)
6. 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 (LCPC'03), pp. 357–373 (2003)
7. Chandy, K.M., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Transactions on Computer Systems **3**(1), 63–75 (1985)
8. Dave, C., Bae, H., Min, S.J., Lee, S., Eigenmann, R., Midkiff, S.: Cetus: A source-to-source compiler infrastructure for multicores. IEEE Computer **42**(12), 36–42 (2009)
9. Elnozahy, E.N., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys **34**(3), 375–408 (2002)
10. Gibson, G., Schroeder, B., Digney, J.: Failure tolerance in petascale computers. CT-Watch Quarterly **3**(4), 4–10 (2007)
11. Landau, C.R.: The checkpoint mechanism in KeyKOS. In: Proceedings of the 2nd International Workshop on Object Orientation on Operating Systems (I-WOOOS'92), pp. 86–91 (1992)
12. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis. In: Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO'04), pp. 75–88 (2004)
13. Li, C.C.J., Stewart, E.M., Fuchs, W.K.: Compiler-assisted full checkpointing. Software: Practice and Experience **24**(10), 871–886 (1994)
14. National Aeronautics and Space Administration: The NAS Parallel Benchmarks (retrieved December 2011). http://www.nas.nasa.gov/publications/npb.html
15. Ousterhout, J.K., Cherenson, A.R., Douglis, F., Nelson, M.N., Welch, B.B.: The Sprite network operating system. IEEE Computer **21**(2), 23–36 (1988)
16. Parr, T.J., Quong, R.W.: ANTLR: a predicated-LL(k) parser generator. Software: Practice and Experience **25**(7), 789–810 (1995)
17. Plank, J.S., Beck, M., Kingsley, G.: Compiler-assisted memory exclusion for fast checkpointing. IEEE Technical Committee on Operating Systems and Application Environments **7**(4), 10–14 (1995)
18. Plank, J.S., Beck, M., Kingsley, G., Li, K.: Libckpt: Transparent checkpointing under Unix. In: Usenix Winter Technical Conference, pp. 213–223 (1995)
19. Ramkumar, B., Strumpen, V.: Portable checkpointing for heterogeneous architectures. In: Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS'97), pp. 58–67 (1997)
20. Rodríguez, G., Martín, M.J., González, P., Touriño, J.: Controller/precompiler for portable checkpointing. IEICE Transactions on Information and Systems **E89-D**(2), 408–417 (2006)
21. Rodríguez, G., Martín, M.J., González, P., Touriño, J.: A heuristic approach for the automatic insertion of checkpoints in message-passing codes. Journal of Universal Computer Science **15**(14), 2894–2911 (2009)

22. Rodríguez, G., Martín, M.J., González, P., Touriño, J.: Analysis of performance-impacting factors on checkpointing frameworks: the CPPC case study. The Computer Journal **54**(11), 1821–1837 (2011)
23. Rodríguez, G., Martín, M.J., González, P., Touriño, J., Doallo, R.: CPPC: A compiler-assisted tool for portable checkpointing of message-passing applications. Concurrency and Computation: Practice and Experience **22**(6), 749–766 (2010)
24. Russinovich, M., Segall, Z.: Fault-tolerance for off-the-shelf applications and hardware. In: Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS'95), pp. 67–71 (1995)
25. Shires, D., Pollock, L., Sprenkle, S.: Program flow graph construction for static analysis of MPI programs. In: Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), pp. 1847–1853 (1999)
26. Woo, N., Jung, H., Yeom, H.Y., Park, T., Park, H.: MPICH-GF: Transparent checkpointing and rollback-recovery for Grid-enabled MPI processes. IEICE Transactions on Information and Systems **E87-D**(7), 1820–1828 (2004)