J Supercomput (2011) 57:227–255
DOI 10.1007/s11227-010-0383-5

A technique for non-invasive application-level checkpointing

Ritu Arora · Purushotham Bangalore · Marjan Mernik

Published online: 16 February 2010

Abstract One of the key elements required for writing self-healing applications for distributed and dynamic computing environments is checkpointing. Checkpointing is a mechanism by which an application is made resilient to failures by storing its state periodically to the disk. The main goal of this research is to enable non-invasive reengineering of existing applications to insert Application-Level Checkpointing (ALC) mechanism. The Domain-Specific Language (DSL) developed in this research serves as a perfect means towards this end and is used for obtaining the ALC-specifications from the end-users. These specifications are used for generating and inserting the actual checkpointing code into the existing application. The performance of the application having the generated checkpointing code is comparable to the performance of the application in which the checkpointing code was inserted manually. With slight modifications, the DSL developed in this research can be used for specifying the ALC mechanism in several base languages (e.g., C/C++, Java, and FORTRAN).

Keywords Fault-tolerance · Application-level checkpointing · Domain-specific language

R. Arora (✉) · P. Bangalore · M. Mernik
Dept. of Computer and Information Sciences, The University of Alabama at Birmingham, 1300 University Blvd., Birmingham, AL 35294-1170, USA
e-mail: ritu@cis.uab.edu

P. Bangalore
e-mail: puri@cis.uab.edu

M. Mernik
Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova 17, 2000 Maribor, Slovenia
e-mail: marjan.mernik@uni-mb.si

1 Introduction

Checkpointing is a mechanism by which an application is made resilient to failures by periodically saving its state to the disk. Scientific applications that take enormous amount of time to execute (e.g., simulation for protein structure prediction [1] or climate modeling [2]) and are run in distributed, dynamic and heterogeneous environments, like a grid, can benefit considerably from checkpointing. In case of failures or changes in the availability of underlying resources, instead of restarting the application from the beginning, the application is restarted from the latest checkpoint. This is achieved by recreating the pre-failure application state from the saved data on the disk. Checkpointing is also an essential component for writing self-healing applications. These applications have the ability of monitoring their own state, detecting faults, and recovering from the faults automatically. Checkpointing or a similar mechanism (e.g., logging) is required to recover from the fault and continue execution without having to restart the whole application.

Writing and reading the application state are the major steps involved in checkpointing. Together, these steps are referred to as Checkpointing and Restart (CaR) through the rest of this paper. The main types of checkpointing techniques, depending upon the level of transparency, are: hardware-level [3], system-level [4], user-level [5], application-level [6–9], and hybrid approaches [10].

- In the *hardware-level* checkpointing, specialized hardware (e.g., redundant arrays of inexpensive disks, custom-designed directory controller, and cache memory) can be integrated into the processors for saving the state of the application.
- The *system-level* checkpointing is done external to the application with the support of the operating system and it involves periodically saving the execution state of the entire application. This typically requires changes to the operating system's kernel and the entire process state is saved since the operating system does not have knowledge about the application semantics.
- The *user-level* checkpointing process is often done by linking the checkpointing libraries to the application code. The programmer is free from the burden of making any changes to the code and no additional code is required to be installed in the kernel as compared to the system-level checkpointing. This approach is usually architecture-dependent.
- The *Application-Level Checkpointing (ALC)* is a type of checkpointing in which an application is made reliable by inserting the fault-tolerance mechanism directly into it. Only the critical variables and data structures are saved to the disk during ALC.
- A *Hybrid Approach* is a combination of more than one type of checkpointing techniques. A hybrid of system-level checkpointing and ALC is presented in [10]. The authors claim that this combination results in higher reliability in real-time systems.

Although ALC requires more end-user involvement than any other form of checkpointing, it has several advantages [11]. As compared to other types of checkpointing, ALC involves lesser storage space (core-dumps are taken in system-level checkpointing), gives more control for selective checkpointing to the end-user and is useful for

writing portable applications for different operating systems. As mentioned in [11], the ALC schemes are language-independent "provided that the base language constructs are present" [11].

One major problem with the current techniques for ALC is that they necessitate invasive reengineering of existing applications for inserting the checkpointing code (typically done by inserting macros in the source code) and thus make software maintenance extremely challenging. If the application code is large, and the number of critical variables is huge, there might be multiple places at which the end-user might be required to make changes in the existing application in order to make it fault-tolerant. Because ALC involves extra read and write operations, the checkpointed version of the application might take conspicuously longer time to run than the non-checkpointed one. In the scenario in which the performance is more critical than fault-tolerance, the stakeholder might want to have the facility to turn off the checkpointing feature. For the convenience of code maintenance and evolution, it is also important to avoid creating multiple versions of the application (with and without checkpointing). Also, the solution space for ALC is constantly evolving. There are many checkpointing libraries and techniques that already exist, each having some special merits over the others [4, 11]. With the emergence of many-core and multi-core architectures, more solutions for fault-tolerance are expected to emerge. Given such a widespread and an evolving solution space, the end-users should not be forced to reengineer their application in the event in which they want to switch from one solution to another. Due to all these reasons, it is desirable that the existing application does not undergo any invasive reengineering in order to become fault-tolerant and the CaR mechanism (to enable ALC) exists as a pluggable feature.

The rest of this paper is ALC-centric and a detailed discussion of other forms of checkpointing is beyond the scope of the paper. Because this research necessitates the involvement of the end-user, it is therefore only a semi-automatic approach to ALC. This work is relevant for both uniprocessor and multiprocessor systems. At a coarse-grain level, it can be said that the ALC-approach presented in this paper is selective (core-dump of the processor's state is not taken), periodic (checkpoints are always taken at a particular frequency), and static (the checkpoints are known before the program is run). In case of checkpointing parallel applications with this approach, depending upon the end-user's choice, checkpointing can either be centralized (only one processor initiates the checkpoint) or distributed (each processor participates in the checkpointing process). Because while taking centralized checkpoints with this approach, it is important that the processors are in a synchronized state, this approach is a coordinated one. However, synchronizing the processors is not a part of the approach presented in this paper. It is the end-user's responsibility to ensure this manually.

The focus of the research presented in this paper is to raise the level of abstraction of ALC such that the end-user is not responsible for manually reengineering the existing application for inserting the checkpointing code. Instead of writing the optimized ALC code by hand or inserting any library calls in the code (which could lead to code tangling [12]), the end-user provides the CaR-specifications (what should be checkpointed and where, along with the frequency of checkpointing) using a high-level language. The optimized code is then generated and inserted into the existing application using a set of domain-specific optimizations (i.e., transformations) [13]. This

approach therefore not only solves the problems related to versioning and maintenance (described above) but also allows the end-users to take advantage of the latest tools and techniques for ALC. Some other advantages of this high-level approach are enhanced code reuse, absence of code restructuring, and highly comprehensible/readable code for the CaR mechanism. The facility to checkpoint code at arbitrary points in the application is also provided through this research. The applications that were made fault-tolerant by inserting the ALC code through the technique demonstrated in this research produce the results with the same accuracy and precision as the non-checkpointed code or the manually checkpointed code. The performance of the application checkpointed by this technique is comparable to the manually checkpointed version of the application.

The background and related work are presented in Sect. 2 of the paper. The key idea behind this research is presented in Sect. 3. The test cases used for validating the results of this research are described in Sect. 4. The results are presented in Sect. 5 of the paper. Conclusion and future work are presented in Sect. 6.

2 Background and related work

Bronevetsky et al. [6–8] have proposed a preprocessor-based approach for ALC. Their work is relevant for both shared memory and distributed memory architectures and their approach consists of two components: a preprocessor, and a checkpointing library. With their approach, the programmer invasively changes the existing application to insert the calls to a predefined function for checkpointing, at the points in the program where checkpointing is desired. An optimized approach to automated ALC is presented in [8], which is helpful for asynchronously checkpointing an application.

Ramkumar et al. [14] have used a source-to-source compilation technique for creating portable checkpoints. In their approach, too, the end-user has to instrument the existing code by renaming functions and by inserting the call to the checkpointing library function. The frequency of checkpointing is controlled using a timer that triggers checkpointing. The state of the program is stored on stacks and this approach doubles the memory requirement for running an application. In case the DRAM cannot hold the data on the stack, the stack is mapped to a local disk and thus extra checkpointing overheads are introduced.

Jiang et al. [15] proposed an ALC technique for shared-memory architectures which they call MigThread. This technique consists of a LEX-based preprocessor and a run-time support module. The preprocessor scans the code and inserts the thread migration primitives, renames the functions and variables and inserts other code required for thread migration [15]. In this technique, parts of computation are assigned to different threads, the computation is paused, the state of the threads (process, computation, communication) is migrated to a different node, and the computation is resumed.

In [16], Czarnul et al. have proposed a user-guided approach for inserting calls to their checkpointing library, either through a dedicated master processor or collectively by all the processors, and call it PARUG. This approach offers the flexibility of selective checkpointing to the end-user but is invasive.

Aspect-Oriented Programming (AOP) [12] technique is another way to achieve the objective of non-invasive ALC and was used during the initial phase of this research [9]. AOP is an advanced form of modular programming that helps the programmer in separating cross-cutting concerns [12]. When a concern (a program feature) is present at multiple places and modules in an application, it is known as cross-cutting. Examples of cross-cutting concerns are logging and ALC. Not only is it difficult to maintain and evolve applications having cross-cutting concerns, it is hard to reuse the code. With the AOP technique, the cross-cutting concerns are isolated in modular units called aspects that can be woven into an application as required. This type of modularization helps in non-invasive reengineering (or transformation) of the applications. The aspects can be plugged into the application on demand without affecting the existing base application. The main AOP concepts (e.g., join-point model [12]) were used to develop the non-invasive and high-level ALC solution presented in this paper.

The major differences between the research presented in this paper and other related works are the non-invasive reengineering of existing applications, separation of the checkpointing concern from the existing application, and the readability/comprehensibility of the generated code. However, the onus is on the end-user to identify the places in the code where checkpointing is required and to specify the checkpointing-frequency. As compared to Bronevetsky et al.'s approach, the research presented in this paper is non-invasive but semi-automatic and the end-user is responsible for ensuring that the processors are in a consistent state before taking the checkpoint. As compared to Ramkumar et al.'s approach, the research presented in this paper is at a very high level of abstraction, gives control to the end-user to select the critical program variables to checkpoint and to select the frequency of checkpointing. As compared to MigThread, the research presented in this paper is relevant for different types of architectures and the transformed code is more comprehensible to the end-user because the original structure is maintained as is with the exception of checkpointing code inserted at the specified places in the application. The AOP-based checkpointing approach adopted initially in this research imposed some limitations that were overcome through the current approach. The AOP language extensions that are available today for C/C++ and Java do not support code transformation at any arbitrary place in the application [9, 17] but with the current approach, presented in this paper, the CaR code can be inserted in applications at arbitrary points. Also, there is no mature AOP language extension available for FORTRAN (another most common language used by scientific community) at the time of writing this paper. The work done in this paper can be extended to support non-invasive ALC of applications written in several base languages, including FORTRAN.

Though the work of Roychoudhury et al. [18] is related to the aspect-weaving domain (and is not specific to the ALC domain), it was the inspiration behind the design of the framework developed in this research. They have demonstrated a technique for constructing aspect-weavers for general-purpose programming languages by combining model-driven engineering with a program transformation system. They capture the aspect-specifications in an abstract manner such that there is no dependency on any one particular program transformation system. Similarly, in the research presented in this paper (which is specific to the ALC domain), the CaR-specifications

are decoupled from the actual implementation of the CaR code in the base application. Because of this decoupling, with slight modifications, the DSL developed in this research can be used for specifying the ALC mechanism in several base languages (e.g., C/C++, Java, and FORTRAN).

3 Overview of the approach

The technique for non-invasive ALC presented in this paper is a two-step process. It involves the following two items:

1. Implementation of abstractions for expressing the CaR-specifications.
2. Implementation of a backend for code instrumentation.

The abstraction for expressing the CaR-specifications can be achieved through a Domain-Specific Language (DSL).

DSLs are specialized, high-level languages that are written to solve problems in a particular application-domain [19]. The DSLs can be said to raise the level of abstraction to that of the problem domain itself such that they are easy to learn and use as compared to the General-Purpose Languages (GPLs). An excellent example of a DSL is Structured Query Language (SQL) which is related to the database domain. For a given domain, a DSL is more expressive than a GPL but unlike the GPLs, has limited features and applicability. In [19], it has been stressed that the usage of a DSL results in the increase in productivity and decrease in software development time and cost.

A DSL for ALC was developed in this research for obtaining the CaR-specifications from the end-users. As shown in Fig. 1, the end-user provides the existing application and the DSL code (CaR-specifications). The remaining steps for generating the checkpointed application are carried out automatically at the backend and are hidden from the end-user. In the backend, the CaR-specifications are used for generating the actual CaR code in the base language and for non-invasively inserting the same into the existing application. The checkpointed application can be compiled and run in the same way as the original non-checkpointed application. This not only raises the level of abstraction of ALC and decouples the specifications from the implementation, but also reduces the time, cost and complexity involved in the ALC of existing

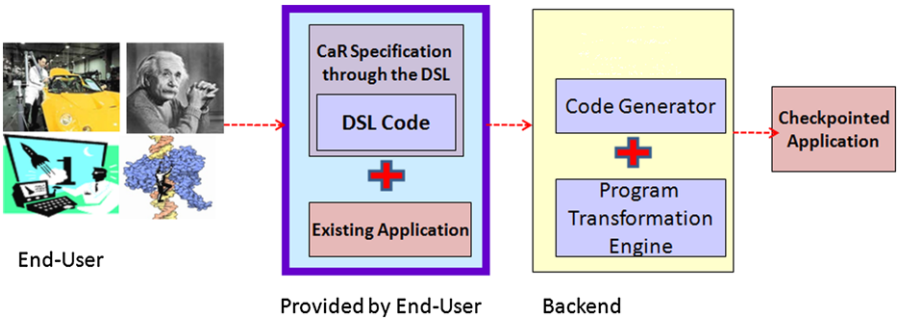

Fig. 1 Steps involved in checkpointing an application

applications, especially if the applications are large [13]. This DSL can be used for checkpointing both sequential and parallel applications from diverse domains. The details about the analysis and design required for developing the DSL, DSL description, and the backend components are provided in the following subsections. Also provided is an example to illustrate the usage of the DSL and its advantages.

3.1 Domain analysis

The first step in the development of a DSL is the analysis of the domain (in this case ALC) for which it is being designed. During the domain analysis phase of developing the DSL, a survey of technical literature and existing implementations [6, 7, 14, 15, 20] was done to obtain an overview of the terminologies and concepts related to the ALC-domain. Commonly used terms and their relationships were used to develop the domain lexicon. Commonalities and differences were observed in the process of implementing the CaR mechanism across applications in various domains and these are referred to as features from this point onward in this paper. Some of the features in the ALC-domain and their relationships are shown as expressions in Table 1.

As shown in Table 1, the feature `ChckptgPack` indicates that this DSL package allows two types of activities, `Checkpoint` and `Restart`. If the user wants to `Checkpoint` an application then the checkpoint condition, `CheckPointCondition`, and the code that should be checkpointed, `CheckPointCode`, are specified. The `CheckPointCondition` includes the specification of the points where the code for checkpointing should be inserted (`Hook` and `Pattern`). It also includes the frequency of checkpointing (`Frequency`) and the type of CaR (`CaRType`). The expression `Hook` is made up of `HookType` and `HookElement`. Together with the `Pattern` (which is a search string), these two syntax elements are used to identify the places in the application code where the checkpointing code should

Table 1 Excerpt of the features identified in the ALC-domain

```
ChckptgPack: all(Checkpoint, Restart)
Checkpoint: all (CheckPointCondition, CheckPointCode)
CheckPointCondition: all(Hook,Pattern,Frequency, loopVar?,CaRType)
CaRType: one-of(Centralized, Distributed, Sequential)
CheckPointCode: all(SaveVarType, saveVarArg)
SaveVarType: one-of (SaveInt, SaveDouble, SaveChar,…)
Restart: all (RestartCondition, RestartCode)
RestartCondition: all(Hook, pattern)
RestartCode: all(ReadVarType, restartVarArg)
ReadVarType: one-of (ReadIntVarFromFile, ReadDoubleVarFromFile,…)
Hook: all (HookType, HookElement)
HookType: one-of(afterHookType, beforeHookType, aroundHookType)
HookElement: one-of(Call, Execution, Statement)
```

Table 2 Some APIs in the DSL for CaR-specification

SaveInt(<variable name>, <file name>)
SaveIntArray1D(<array name>, <number of columns>, <file name>)
SaveIntArray2D(<array name>, <number of rows>, <number of columns>, <file name>)
ReadIntVarFromFile(<variable name>, <file name>)
ReadIntArray1DFromFile(<array name>, <number of columns>, <file name>)
ReadIntArray2DFromFile(<array name>, <number of rows>, <number of columns>, <file name>)
…

be inserted. In case, checkpointing is required inside a loop, the name of the loop variable, `loopVar`, should also be specified. This is an optional feature and is represented by "`?`". The desired type of CaR (`Centralized`, `Distributed` or `Sequential`) should also be specified as a part of `CheckPointCondition`. The `CheckPointCode` includes the specification of the variable or data structure to be checkpointed. Depending upon the type of the variable or data structure, the user is expected to specify a list of parameters. For example, if the user intends to save an integer variable, `SaveInt` is selected from the list of `SaveVarType`. The other parameters required from the user in this case would be the name of the variable, and the name of the file in which the variable needs to be saved.

A set of APIs was developed for capturing the details about the variable or data-structure to be saved and some of them are presented in Table 2. If the user intends to save a two-dimensional array of type integer (specified by `SaveIntArray2D`), then apart from the name of the array and the file name, the dimension of the array also needs to be specified.

Likewise, during the restart phase, as per the expression for the feature `Restart`, the user should specify the `RestartCondition` and the `RestartCode`. As in the case of `CheckPointCondition`, the `RestartCondition` includes the specification of the `Hook` and `Pattern`. The `Hook` and `Pattern` are together used to identify the place where the restart code should be inserted. The `RestartCode` specification includes the description of the type of the variable or data structure being read, the name of the variable to be initialized with the value stored in the restart file and the name of the restart file. If the restart file exists, then the variable is initialized by the value stored in the restart file; else, the program proceeds with the normal initialization process.

3.2 DSL design

The DSL for ALC was designed from scratch with no commonality with the existing language. However, this DSL borrows some concepts and constructs from the AOP techniques. In AOP:

1. A *join point* is a well-defined point in the program flow [12], e.g., a method call (point in the program from where a method is called) and a method execution (point in the program where the code in the method is executed).
2. A *pointcut* is used to identify a set of join points [12] that are of interest in the program flow.
3. An *advice* is a combination of a *pointcut* and the code that should be run at the *join point* [12]. There can be several kinds of *advice* but the main ones are *after advice, before advice,* and *around advice.*
4. An *after advice* runs after the program proceeds with a particular *join point*.
5. A *before advice* runs before the program proceeds with a particular *join point*.
6. An *around advice* runs as soon as the *join point* is reached, and depending upon the code in the advice, the program may proceed with the *join point* or it can skip the execution of the *join point*, running only the advice code.

Similar to the concept of *advice* in AOP, the DSL for ALC has a notion of a well-defined `Hook` (shown in Table 1) which is used as a handle to a specific point in the program flow. A `Hook` can be of one of the following types: *after*, *before*, and *around*. A `Hook` of type *after* has the same significance as an *after advice* in AOP. The *before* and *around* type correspond to the *before advice* and the *around advice* in AOP. Apart from the type, a hook definition also includes the specification of the *pointcut*. Unlike many language extensions of AOP, in this DSL any syntactically correct program statement can be specified as a *join point*. A partial list of the type of *join points* that can be specified using this DSL, are: function call, function execution, expression statement, compound statement, selection statement, and iteration statement. These different *join points* give different granularity of control to the end-user. For example, in case the *join point* is of type function execution, the end-user gets control of the execution point of the function such that the behavior and structure of the entire code in the function body can be modified if desired. As opposed to function execution, if any one particular statement in the function needs to be modified, the *join point* should be of type statement (examples of allowed statement types are expression statement and iteration statement). Based on the way the function execution and function call *join points* are implemented, they can differ in the scope of action. The scope of function call type of *join point* starts with the call to the function and lasts till the program control returns from the function. The scope of function execution type of *join point* starts with the execution of the code in the body of the function and lasts till the last line of the code in the function body. The DSL keywords for expressing a *pointcut* are *call*, *execution*, and *statement* along with a search pattern. An example of a `Hook` definition along with the search pattern would be:

`around statement` (“`start = 0;`”)

In this example, the statement (“`start = 0;`”) serves as a *join point* of type *around*.

One of the most important steps during the DSL design stage is to choose a structure for DSL code constructs. In this DSL, the variant features are specified by the user and the constant features are automatically generated in the editor. As per the DSL design, the conditions and the code for checkpointing should be provided by the user in the code block following the keyword `beginCheckpointing`. The

```
beginCheckpointing:
<Hook> <Pattern> && (Frequency = "<#>") &&(loopVar =  "<>")
&& <CaRType>{
<checkpointing code>
}
beginInitialization:
<Hook> {
<restart code>
}
```

Fig. 2 Basic structure of the DSL code

conditions and the code for restart should be provided by the user in the code block following the keyword `beginInitialization`. Apart from deciding the structure of the language constructs, the valid and invalid combinations of the features were also identified in the design phase. For example, any attempt to specify the code pertaining to the restart mechanism (e.g., `ReadIntVarFromFile`) should not be allowed in the block following the keyword `beginCheckpointing`. Therefore, `beginCheckpointing` and `ReadIntVarFromFile` are invalid combinations of the DSL features. The valid and invalid combination of features is called configuration knowledge [13] and is required during the DSL implementation phase.

3.3 DSL description

The basic structure of the DSL code is shown in Fig. 2. The place-holder for the variant part, which should be provided by the end-user, is depicted by "< >". The *Hook* is a statement, or a function call, or function execution before, after or around which the checkpointing or restart functionality is desired. The *Pattern* of the *Hook* and the `Frequency` of checkpointing, which is an integer value, are also required as a part of the CaR-specification. The `&&` operator is used to create a powerful expression for CaR-specifications. The **`loopVar`** shown in Fig. 2 is an optional structural element and is used only if the variable or data structure meant to be checkpointed is inside a loop. The datastructures and variables to be checkpointed are specified within "{" and "}".

3.4 Sample DSL code

A simple function, `computepi`, for computing the value of pi (i.e., π) using C++/MPI [21] is shown in Fig. 3 to illustrate the DSL code to be provided. If the variable `mysum` needs to be checkpointed after the execution of the statement on line # 5, at a frequency of every 10 iterations of the `for` loop on line # 3 of Fig. 3, then the corresponding DSL code for specifying this intention is shown in Fig. 4. The keyword **`beginCheckpointing:`** on line # 1 of the code marks the beginning of the checkpointing block and is compulsory. The code on lines # 2–4 of the Fig. 4 is used to express the checkpointing condition. The code on line # 6 of Fig. 4 means that the

```
1. double computepi(int start, int end, double h) {
2.   double mysum = 0.0;
3.   for (int i=start; i<=end; i++) {
4.     double x = h * ((double)i - 0.5);
5.     mysum += 4.0 / (1.0 + x*x);
6.   }
7.   return h*mysum;
8. }
```

Fig. 3 Function to compute the value of pi

```
1. beginCheckpointing:
2. after statement("mysum += 4.0 / (1.0 + x*x);")
3. && (frequency = 10)
4. && (loopVar = "i" )&& (CaRType = Distributed)
5. {
6.   SaveDouble(mysum, "restartMysum")
7.   SaveInt(i, "restartI")
8. }
```

Fig. 4 Sample DSL code for checkpointing

```
1. beginInitialization:
2. after statement ("double mysum = 0.0;")
3. {
4.   ReadIntVarFromFile (start, "restartI")
5.   ReadDoubleVarFromFile (mysum, "restartMysum")
6. }
```

Fig. 5 Sample DSL code for restart

variable named `mysum` of type `double` is being saved in a file named `restartMysum`. The iteration number is also being saved and the code for the same is on line # 7 of Fig. 4.

During the restart phase, the variable `mysum` and the starting value of iteration count, `start`, are initialized from latest checkpoint stored in the files `restartMysum` and `restartI`. The DSL code for specifying this intent is shown in Fig. 5. The keyword **`beginInitialization:`** on line # 1 of the code is compulsory. As per the DSL design, if the end-user attempts to provide the CaR-specifications without providing the necessary keywords, the parser will complain about it and the code generation process will not proceed.

3.5 Backend development

The DSL source code needs to be translated into the language in which the base application is written (which is called the base language). The DSL code in this research is transformed into the base language (C/C++) source code via a Program Transformation Engine (PTE) [13, 22] and transformation languages. First, the DSL code is translated into an intermediate code for the PTE via the Atlas Transformation Language (ATL) [23]. Using the intermediate code, the PTE generates the code

```
1.      double computepi(int start, int end, double h) {
2.             FILE* newInputFile1, newInputFile2;
3.             char *addString1, *addString2;
4.             /*other code including restart code,code for setting
                   file names dynamically- e.g., addstring1 is set to
                   (restartMysum+processor id) & addstring2 is set to
                   (restartI+processor id)*/
5.             double mysum = 0.0;
6.             for (int i=start; i<=end; i++) {
7.               double x = h * ((double)i - 0.5);
8.               mysum += 4.0 / (1.0 + x*x);
9.               if (i % 10 == 0){
10.               newInputFile1 = fopen(addString1, "w");
11.               newInputFile2 = fopen(addString2, "w");
12.               fprintf(newInputFile1, "%lf  ", mysum);
13.               fprintf(newInputFile2, "%d  ", i);
14.               fclose(newInputFile1);
15.               fclose(newInputFile2);
16.              }
17.            }
18.            return h*mysum;
19.     }
```

Fig. 6 Checkpointed function to compute the value of pi

in the base language (C/C++) and inserts it automatically into the base application. The example code shown in Fig. 3 is checkpointed by the PTE, on the basis of the specifications provided in Figs. 4 and 5. The output is shown in Fig. 6. The code for saving the values of the critical variables is on lines # 9–16 of Fig. 6. The restart code has been omitted for clarity and brevity.

The PTE is capable of doing non-invasive source-to-source transformation and, therefore, could have been used directly to transform the existing application into the checkpointed one. This would have obviated the extra effort spent in developing the DSL. However, PTEs are complex and difficult to learn and use. Therefore, an extra layer of abstraction in the form of a DSL was absolutely necessary in this research. Additional details on the design and development of this technique for ALC are beyond the scope of this paper but have been discussed in [24] and [25]. With minimum effort, the DSL can be extended to add the facility to checkpoint additional data structures that are currently not covered in its present scope.

In order to promote DSL code correctness and to reduce coding complexity, a wizard-driven GUI for DSL code generation (Fig. 7) has been developed. The end-user can enter the CaR-specifications through the GUI instead of typing them manually as DSL code. For example, the end-user can select one of the features from the list of `ReadVarType` features and provide the parameters (like variable name, restart file name). The corresponding DSL code is generated automatically. On the basis of the selections made in the panel for providing checkpointing-specifications, the panel for restart-specifications is generated dynamically. An outline of the workflow involved in providing the CaR-specifications is shown in the panel on the left-hand side of the GUI. A summary page showing the CaR-specifications can be presented to the end-user in the end for the purpose of overview.

This GUI was developed using the APIs and user-interface from SwingLabs, a subproject supported by Open source Java projects, an open source initiative from Sun Microsystems and hosted at https://wizard.dev.java.net/quickstart.html. Because

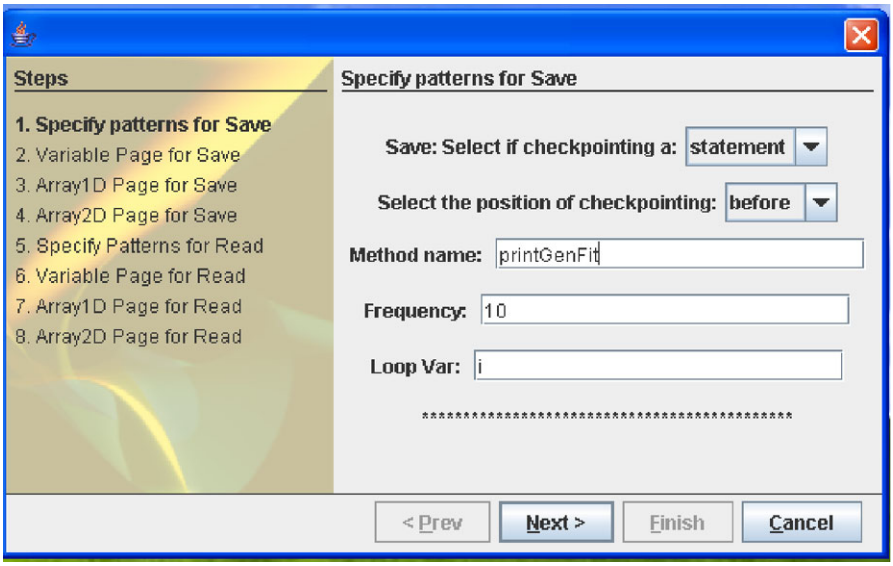

Fig. 7 Wizard for generating the DSL code

wizard content needs to vary dynamically (contents on the next panel depends upon the contents of the previous panel/panels), nesting of wizards within wizards was done. Input validation can be easily programmed and the process of developing this wizard-driven GUI was itself wizard-driven! This wizard can be run from any platform that has a Java virtual machine installed.

3.6 Benefits of using the DSL

The main advantages of using the DSL for ALC can be summarized as follows:

1. Non-invasive ALC of existing applications.
2. Mitigation of the complexity associated with the usage of a PTE.
3. High-level of abstraction of source-to-source transformation.
4. Decoupling of the problem and solution space, i.e., the CaR-specifications are decoupled from the actual implementation of the CaR mechanism.
5. Prevention of code tangling and thus reduction in the effort involved in software maintenance.

4 Case-studies

While implementing ALC, the end-user should identify the main data structures or variables from which the entire execution state of the application can be recreated in case of a failure. Since the ALC involves saving the state of the critical data structures or variables by writing to a disk, it can incur extra run-time overheads. Therefore, the frequency at which the checkpoint is taken is also important. The place in the

application where the checkpoint should be taken can affect the accuracy of the results in case of the restart. The end-user should therefore be familiar with the logic of the application in which the CaR mechanism needs to be inserted.

Applications from diverse domains were selected for the validation of the approach presented in this paper. Some of them are Genetic Algorithm (GA) for Content Based Image Retrieval (CBIR), Poisson's Solver, and the Circuit Satisfiability problem. A brief description of these applications is presented in this section. Checkpointing was implemented in both the sequential and parallel versions of the applications.

4.1 GA for CBIR

The CBIR technique is used for searching images in large databases on the basis of the image content instead of the image captions [26]. The images are sliced into smaller semantic regions and are stored as blobs in the database. Each segment represents an individual semantic region of the original image (e.g., grass, tiger, butterfly). The next step involves the extraction of features (color, texture, shape) for each image segment. Because the amount of image data here is very large, clustering is used to preprocess the data and reduce the search space in the image retrieval process. The clustering is performed on image segments and therefore if a segment belongs to the cluster, so does the image containing the segment. The type of clustering performed here is GA-based. A typical experiment involved using 9,800 images with 82,556 regions and these image regions were divided into 100 clusters. Additional details and steps involved in the CBIR procedure can be found in [26].

The GA for CBIR is an excellent test case because it is computation-intensive. Also, the GA can get stuck in local optima and hence should be run for large number of generations to get the globally optimal results. Therefore, it is imperative to checkpoint the application, especially when it is run in a dynamic and distributed environment. Depending upon the implementation scheme (type of load-balancing and design pattern) of the GA and the end-user's preference, the population and the fitness value of the chromosomes can be saved after certain number of generations or even during the last generation [9]. In this experiment, if the GA is run for 100 generations or greater, it produces better quality of clusters. The accuracy with which the images in the database are matched with the search image improves with improvement in the quality of clusters. The state of the executing GA application depends upon the current or initial population, and the seed value of the random-number generator function. Therefore, for checkpointing the GA, the current population and the value of the seed used to initialize the random-number generator function are stored in a file. The time of the day is passed as the seed value to the random-number generator function in this application. To restart the program from any point in execution, the GA can be made to read the values of the seed of the random number generator and the current population from the restart files.

The code snippet for the GA is shown in Fig. 8. Consider the case in which the checkpointing code should be inserted into this base code after line # 8. The intention is to save the state of the critical variables after every 10 iterations. Figure 9 shows the DSL code for describing this intention.

```
1. /*other code*/
2. for(i=0;i<numGenerations;i++){
3.   printf("Gen: %d ", i);
4.   pickchroms(fitness,popcurrent,popnext);
5.   mutation(popnext,popcurrent);
6.   equate(popcurrent, popnext);
7.   evaluatePop(popcurrent,mydata,fitness);
8.   printGenFit(popcurrent,fitness,(int)time1);
9.  }
10.   /*other code*/
```

Fig. 8 Code snippet of the sequential GA base code

```
1. beginCheckpointing:
2. after execution("printGenFit")&& (frequency = 10)
   && (loopVar = "i" ) && (CaRType = Sequential){
3. SaveInt(time1,"restartTime1")
4. SaveIntArray2D(popcurrent, numChrom, numCentroid,
                                  "restartPopcurrent")
5. }
```

Fig. 9 DSL code snippet for describing checkpointing in the sequential GA code

```
1. beginInitialization: around execution ("dataInitialize"){
2.    ReadIntVarFromFile(time1, "restartTime1")
3.    ReadIntArray2DFromFile(popcurrent, numChrom, numCentroid,
7.   return h*mysum;
8. }
```

Fig. 10 DSL code snippet for describing the restart mechanism in the sequential GA code

The frequency of checkpointing, the CaR type (**`Sequential`**), and the loop variable `i` are specified in the DSL code along with the name of the function `printGenFit after` whose `execution` the checkpointing code should be inserted. The restart mechanism is also specified through the DSL and is shown in Fig. 10. Through this code, the `execution` of the function `dataInitialize` is intercepted. Due to this interception, instead of the execution of the initialization code in the function body, the array `popcurrent` is initialized with the values read from the file, `restartPopcurrent`. If the restart file is not present then the array is initialized using the values read from the file `initial`. The option of reading from one of these two files is expressed by using "|". The variable `time1`, which is passed as a seed to the random number generator, is initialized by the values read from the file `restartTime1`. The variables `numChrom` and `numCentroid` in Figs. 9 and 10 are the dimensions of the array popcurrent and are provided by the user.

```
1. /*other code*/
2.    if((inputfile = fopen("restartPopCurrent", "r")) ==
                                                        NULL){
3.      srand48((long)time1);
4.      inputfile = fopen("initial", "r");
5.    }else{
6.      storeVar = fopen("restartTime1", "r");
7.      fscanf(storeVar, "%d", &time1);
8.      srand48((long)time1);
9.      inputfile = fopen("restartPopCurrent", "r");
10.   }
11.   /*other code*/
12.   for(i=0;i<numGenerations;i++){
13.     printf("Gen: %d ", i);
14.     pickchroms(fitness,popcurrent,popnext);
15.     mutation(popnext,popcurrent);
16.     equate(popcurrent, popnext);
17.     evaluatePop(popcurrent,mydata,fitness);
18.     printGenFit(popcurrent,fitness,(int)time1);
19.     if (i % 10 == 0){
20.      newInputFile = fopen("restartPopcurrent", "w");
21.      storeVar = fopen("restartTime1", "w");
22.      fprintf(storeVar, "%d  ", time1);
23.      for (ii = 0; ii < numChrom; ii++){
24.        for (jj = 0; jj < numCentroid; j++){
25.         fprintf(newInputFile, "%d ",popcurrent[ii][jj]);
26.        }
27.        fprintf(newInputFile, "\n");
28.      }
29.      fclose(newInputFile);
30.      fclose(storeVar);
31.     }
32.   }
33.   /*other code*/
```

Fig. 11 Code snippet of the checkpointed version of the sequential GA

The CaR mechanism described through the DSL, as shown in Figs. 9 and 10, is translated into intermediate code that a PTE can understand to carry out the non-invasive transformation of the existing application into a checkpointed one. As per the specification, the PTE generates the base language code for file I/O. Two files, `restartTime1` and `restartPopcurrent`, are opened and the value of the variable `time1` (which is the seed value) and the contents of the array `popcurrent` are saved to these files. The code snippet of the checkpointed code is shown in Fig. 11. Lines # 2–10 and lines # 19–31 in Fig. 11 are generated and inserted by the PTE for the purpose of file I/O required for the CaR mechanism.

The parallel version of the code for the GA for CBIR is checkpointed in the same manner as the sequential one. The base code for the parallel version is shown in Fig. 12. The CaR-specifications remain almost the same as in Figs. 9 and 10 except that the type of CaR is `Centralized` instead of `Sequential`. The end-user is required to assure that all the partial computation results have been collected from the processors and that the processors are in a synchronized state at the time of taking the checkpoint. The inserted checkpointing code is on lines # 14–28 in Fig. 13.

```
1. /*other code*/
2. for(i=0;i<numGenerations;i++){
3.    if (rank==0){
4.      printf("Generation #: %d",i);
5.    }
6.    popnext=pickchroms(fitness,popcurrent,popnext,start_x_y,…);
7.    MPI_Allgatherv(&popnext[0][0],(start_x_y.ystart_x_y.x)*…,…);
8.    if(rank==0){
9.      popcurrent=mutation(popcurrent,numOfrecords,…);
10.     }
11.   MPI_Bcast(&popcurrent[0][0],numChrom*numCentroid,…);
12.   evaluatePop(popcurrent,mydata,fitness,start_x_y2,…);
13.   printGenFit(popcurrent,fitness,(int)time1,i,rank);
14.   }
```

Fig. 12 Code snippet of the parallel GA base code

```
1. /*other code*/
2. for(i=0;i<numGenerations;i++){
3.    if (rank==0){
4.      printf("Generation #: %d",i);
5.    }
6.    popnext=pickchroms(fitness,popcurrent,popnext,start_x_y,…);
7.    MPI_Allgatherv(&popnext[0][0],(start_x_y.ystart_x_y.x)*…,…);
8.    if(rank==0){
9.      popcurrent=mutation(popcurrent,numOfrecords,…);
10.     }
11.   MPI_Bcast(&popcurrent[0][0],numChrom*numCentroid,…);
12.   evaluatePop(popcurrent,mydata,fitness,start_x_y2,…);
13.   printGenFit(popcurrent,fitness,(int)time1,i,rank);
14.   if(rank==0){
15.     if (i % 10 == 0){
16.       newInputFile = fopen("restartPopcurrent", "w");
17.       storeVar = fopen("restartTime1", "w");
18.       fprintf(storeVar, "%d  ", time1);
19.       for (ii = 0; ii < numChrom; ii++){
20.         for (jj = 0; jj < numCentroid; jj++){
21.           fprintf(newInputFile, "%d  ", popcurrent[ii][jj]);
22.         }
23.         fprintf(newInputFile, "\n");
24.       }
25.       fclose(newInputFile);
26.       fclose(storeVar);
27.     }
28.   }
29.   }
```

Fig. 13 Code snippet of the checkpointed version of the parallel GA

4.2 Poisson's Solver

Solution to partial differential equations is one of the most common computational tasks performed in the Computational Fluid Dynamics (CFD) and the Poisson Solver is a representative application that illustrates the communication and computation patterns in a typical CFD application. In this paper we have considered the solution to

```
1. start = 0;
2. /*other code*/
3. for (k = start; k < NTIMES && norm >= tolerance; k++) {
4. b = compute(a, f, b, N, N);
5. /*other code*/
6. // compute norm
7. norm = normdiff(b, a, N, N);
8. }
```

Fig. 14 Base code snippet of the sequential Poisson's Solver

```
1. beginCheckpointing:
2. before statement ("b = compute(a, f, b, N, N);")
3. && (frequency = 10)
4. && (loopVar="k") && (CaRType = Sequential){
5.   SaveDoubleArray2D(a,N,N,restartA)
6.   SaveDoubleArray2D(f,N,N,restartF)
7.   SaveDouble (norm,restartNorm)
8.   SaveInt (k,restartK)
9. }
```

Fig. 15 DSL code snippet for describing checkpointing in the sequential Poisson's Solver

a 2-D Poisson problem with a five-point stencil [27]. The solution involves computing the value at each point in the computational domain using the neighboring cells from the previous iterations. The matrix holding these computed values, size of the matrix, and the current iteration count need to be stored in a file after every certain number of iterations till a solution converges. In case the application needs to restart from a particular iteration in future, it can be done using the intermediate values of the variables and matrices stored in the file rather than restarting the program.

The base code snippet for the Poisson's Solver is presented in Fig. 14. The critical variables and data structures for this application are matrices `a` and `f`, the number of iterations which is `k`, and the `norm`. The best place to insert the checkpointing code in this application is before line # 4 of the code in Fig. 14. The DSL code for checkpointing this application is shown in Fig. 15. The code on line # 2 of Fig. 15 is the specification of the `Hook` (which is line # 4 of the code in Fig. 14) and is required for pattern matching in the abstract syntax tree of the application code. It is necessary to write the variables and data structures to appropriate files and this intent is expressed by the code on lines # 5–8 of Fig. 15.

The DSL code for specifying the restart mechanism for this application is shown in Fig. 16. Lines # 3–4 of the code imply that the matrices `a` and `f` should be initialized from the values read from the files `restartA` and `restartF`. In case these restart files are not present, the matrices are initialized by calling `initMatrix<double>` (`a`, `N`, `N`, `value`) and `initMatrix<double>` (`f`, `N`, `N`, `value`) respectively. The inserted CaR code is shown on lines # 1–10 and 13–32 in Fig. 17.

The DSL code for checkpointing the parallel version of the Poisson's Solver is slightly different from that of the sequential version (instead of checkpointing the matrices `a` and `f`, matrices `aBig` and `fBig` are being checkpointed here) and is therefore shown in Fig. 19. The code snippet of the parallel version of the Poisson's

```
1. beginInitialization:
2. around statement ("start = 0;"){
3. ReadDoubleArray2DFromFile (a,N,N,"restartA") |
          initMatrix <double> (a, N, N, value)
4. ReadDoubleArray2DFromFile (f,N,N,"restartF") |
          initMatrix<double>(f, N, N, value)
5. ReadDoubleVarFromFile (norm,"restartNorm")
6. ReadIntVarFromFile (start,"restartK")
7. }
```

Fig. 16 DSL code snippet for describing the restart mechanism in sequential Poisson's Solver

```
1. if (!restart) {
2.   start = 0;
3.   initMatrix<double>(a, N, N, value);
4.   initMatrix<double>(f, N, N, value);
5. } else { // read a, f, norm and start from restart file
6.   readMatrix(a,N,N, "restartA");
7.   readMatrix(f,N,N, "restartF",);
8.   readVar(&norm, "restartNorm");
9.   readVar(&start, "restartK");
10.   }
11.   /*other code*/
12.   for (k = start; k < NTIMES && norm >= tolerance; k++){
13.   if(k % 10 == 0){
14.     inputfile1 = fopen("restartA", "w");
15.     inputfile2 = fopen("restartF", "w");
16.     inputfile3 = fopen("restartNorm", "w");
17.     inputfile4 = fopen("restartK", "w");
18.     for (ii = 0; ii < N; ii++){
19.       for (jj = 0; jj < N; jj++){
20.         fprintf(inputfile1, "%lf  ", a[ii][jj]);
21.         fprintf(inputfile2, "%lf  ", f[ii][jj]);
22.       }
23.       fprintf(inputfile1, "\n");
24.       fprintf(inputfile2, "\n");
25.     }
26.     fprintf(inputfile3, "%lf  ", norm);
27.     fprintf(inputfile4, "%d  ", k);
28.     fclose(inputfile1);
29.     fclose(inputfile2);
30.     fclose(inputfile3);
31.     fclose(inputfile4);
32.   }
33.   b = compute(a, f, b, N, N);
34.   /*other code*/
35.   // compute norm
36.   norm = normdiff(b, a, N, N);
37.   }
```

Fig. 17 Code snippet of the checkpointed version of the sequential Poisson's Solver

Solver is shown in Fig. 18. Lines # 2–23 of the code snippet in Fig. 20 show the checkpointing code for the parallel version of the Poisson's Solver. The restart code for both the sequential and parallel versions is the same and hence omitted from the output code snippet.

```
1. for (k = start;k < NTIMES && norm >= tolerance;k++){
2.   b = compute(a, f, b, myrows, mycols);
3.   b = exchange<double>(b, myrows+2, mycols+2,…);
4.   /*code for swapping pointers goes here*/
5.   // compute norm
6.   mynorm = normdiff(b, a, myrows, mycols);
7.   MPI_Allreduce(&mynorm,&norm,1,MPI_DOUBLE,MPI_SUM,…);
8.   aBig = collect<double>(a, aBig, M, N,…);
9.   fBig = collect<double>(f, fBig, M, N,…);
10.   }
```

Fig. 18 Base code snippet of the parallel Poisson's Solver

```
1. beginCheckpointing:
2. before statement("b = compute(a, f, b, N, N);")")
3. && (frequency = 3000)
4. && (loopVar="k") && (CaRType = Centralized)
5. {
6.   SaveInt (k,restartK)
7.   SaveDouble (norm,restartNorm)
8.   SaveDoubleArray2D(aBig ,M+2,N,restartA)
9.   SaveDoubleArray2D(fBig ,N+2,N,restartF)
10.   }
```

Fig. 19 DSL code snippet for describing checkpointing in parallel Poisson's Solver

4.3 Circuit Satisfiability

This is an embarrassingly parallel application that is adapted from Michael Quinn's book "Parallel programming in C with MPI and OpenMP" [28]. The application simulates the actual circuit and determines whether a combination of inputs to the circuit of logical gates produces an output of 1. The application involves an exhaustive search of all the possible combinations of the specified number of bits in the input. For example, for a circuit having 30 bits of input, the search space would involve 2^{30} combinations of the bits, which is 1,073,741,824 possibilities. The parallel version of the application requires distributed checkpointing, unlike the previous two case-studies, which involved centralized or sequential checkpointing.

The base code snippet of the Circuit Satisfiability application is presented in Fig. 21. The critical variables for this application are `upper_limit`, the iteration number which is `i`, and the number of solutions found (which is `mySolutions` in the code). However, in this paper, for the sake of brevity, `mySolutions` is not being considered for the illustration of the checkpointing technique and only the values of `upper_limit` and `i` are being shown to be saved.

It is best to insert the checkpointing code after lines # 8 and 13 of the code in Fig. 21. The DSL code for checkpointing this application is shown in Fig. 22. Because this involves a **`Distributed`** type of checkpoint, each processor is responsible for saving the state of the critical variables in separate files. The restart code in Fig. 23 illustrates the usage of **`after statement`** type of `Hook`. The instrumented code is shown in Fig. 24. As can be noticed from the code, the file names for saving and

```
1. for (k = start;k < NTIMES && norm >= tolerance;k++){
2. if (rank==0){
3.   if(k % 3000 == 0){
4.     inputfile1 = fopen("restartA", "w");
5.     inputfile2 = fopen("restartF", "w");
6.     inputfile3 = fopen("restartNorm", "w");
7.     inputfile4 = fopen("restartK", "w");
8.     fprintf(inputfile3, "%lf  ", norm);
9.     fprintf(inputfile4, "%d  ", k);
10.       for (ii = 0; ii < M+2; ii++){
11.         for (jj = 0; jj < N+2; jj++){
12.           fprintf(inputfile1, "%lf  ", aBig[ii][jj]);
13.           fprintf(inputfile2, "%lf  ", fBig[ii][jj]);
14.         }
15.         fprintf(inputfile1, "\n");
16.         fprintf(inputfile2, "\n");
17.       }
18.       fclose(inputfile1);
19.       fclose(inputfile2);
20.       fclose(inputfile3);
21.       fclose(inputfile4);
22.     }
23.   }
24.   b = compute(a, f, b, myrows, mycols);
25.   b = exchange<double>(b, myrows+2, mycols+2,…);
26.   /*code for swapping pointers goes here*/
27.   // compute norm
28.   mynorm = normdiff(b, a, myrows, mycols);
29.   MPI_Allreduce(&mynorm,&norm,1,MPI_DOUBLE,MPI_SUM,…);
30.   aBig = collect<double>(a, aBig, M, N, P, Q,…);
31.   fBig = collect<double>(f, fBig, M, N, P, Q,…);
32. }
```

Fig. 20 Code snippet of the checkpointed version of the parallel Poisson's Solver

reading the critical variables are generated dynamically for the **`Distributed`** CaR type.

5 Results and analysis

The experiments for this research were run on a 32 node dual-processor Opteron cluster (Everest) and a 128 node dual-processor Xeon cluster (Olympus) in the Department of Computer and Information Sciences at the University of Alabama at Birmingham. In the Everest cluster, each node has 2 GB of RAM, 80 GB of hard drive, and gigabit Ethernet connected with gigabit switch. In the Olympus cluster, each node has 4 GB of RAM, low-latency Infiniband network, and 4 terabytes of disk space. The sequential versions of the applications selected in the case-studies were implemented in C/C++. The parallel versions were implemented in C/C++ and MPI [21]. The process of the code generation (for both the DSL and the CaR) is hidden from the end-user; so essentially, the end-user effort is restricted to providing the CaR-specifications through the wizard-driven GUI or writing the DSL code. In all

```
1. //other code
2. MPI_Init ( &argc, &argv );
3. MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
4. MPI_Comm_size ( MPI_COMM_WORLD, &size );
5. //other code
6. maxLimit= pow(2, input_bits);
7. lower_limit = (minLimit(size-rank)+(rank*maxLimit))/size;
8. upper_limit=(minLimit(size-rank-1)+(rank+1)*maxLimit)/size;
9. for (i = lower_limit; i < upper_limit; i++ ){
10.      convertToBinary( i, input_bits, binary_vector );
11.      value = circuit_value (input_bits, binary_vector );
12.      if ( value == 1 ){
13.        mySolutions++;
        /* write the iteration number & solution to output file*/
14.    }
15.   }
16.   MPI_Reduce(&mySolutions,&globalSolutions,1,MPI_INT,… );
```

Fig. 21 Base code snippet of the parallel Circuit Satisfiability application

```
1. beginCheckpointing:
2. after statement("upper_limit = (minLimit(size-rank-1) +
   (rank+1)*maxLimit)/size;")
3. && (frequency = 1) && (CaRType = Distributed){
4.   SaveLong (upper_limit, restartUpperLimit)
5. }

6. beginCheckpointing:
7. after statement("mySolutions++;") && (frequency = 100)
   && (loopVar="i") && (CaRType = Distributed){
8.   SaveLong (i, restartLowerLimit)
9. }
```

Fig. 22 DSL code for checkpointing the Circuit Satisfiability application

```
1. beginInitialization:
   after statement ("maxLimit= pow(2, input_bits);"){
2.    ReadLongVarFromFile(minLimit, "restartLowerLimit")
3.    ReadLongVarFromFile(maxLimit, "restartUpperLimit")
4. }
```

Fig. 23 DSL code for specifying the restart code for the Circuit Satisfiability application

the experiments, as expected, it was observed that the overhead due to checkpointing decreases with the decrease in the frequency of checkpointing.

Both the sequential and parallel versions of the GA for CBIR were checkpointed through the DSL and also manually for comparison purposes. There were 82,556 image segments involved in the experiment. The population size considered in the experiment was 50 chromosomes and the number of centroids on each chromosome was 100. A comparison of the execution time (in seconds) of the sequential GA having the manual and the generated version of the CaR code is shown in Fig. 25. The

```
1. //other code
2. char fname1[20] = "restartLowerLimit";
3. char fname2[20] = "restartUpperLimit";
4. char buf[5];
5. char *addString1, *addString2;
6. //other code
7. MPI_Init ( &argc, &argv );
8. MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
9. MPI_Comm_size ( MPI_COMM_WORLD, &size );
10.   addString1=(char*)malloc((strlen(fname1)+
                                  strlen(buf)+1)*sizeof(char));
11.   strcpy(addString1,fname1);
12.   sprintf(buf,"%d",rank);
13.   strcat(addString1,buf);
14.   addString2=(char*)malloc((strlen(fname2)+
                                  strlen(buf)+1)*sizeof(char));
15.   //other code
16.   maxLimit= pow(2, input_bits);
17.   inputfile1 = fopen(addString1, "r");
18.   if(inputfile1 !=NULL){
19.    fscanf(inputfile1 , "%lld", &minLimit);
20.    fclose(inputfile1);
21.   }
22.   inputfile2 = fopen(addString2, "r");
23.   if(inputfile2 !=NULL){
24.    fscanf(inputfile2, "%lld", &maxLimit);
25.    fclose(inputfile2);
26.   }
27.   lower_limit = (minLimit(size-rank)+(rank*maxLimit))/size;
28.   upper_limit=(minLimit(size-rank-1)+(rank+1)*maxLimit)/size;
29.   inputfile1 = fopen(addString2, "w");
30.   fprintf(inputfile1, "%lld  ", upper_limit);
31.   fclose(inputfile1);
32.   for (i = lower_limit; i < upper_limit; i++ ){
33.      convertToBinary( i, input_bits, binary_vector );
34.      value = circuit_value (input_bits, binary_vector );
35.      if ( value == 1 ){
36.        mySolutions++;
37.        if(i % 100 == 0){
38.         inputfile2 = fopen(addString1, "w");
39.         fprintf(inputfile2, "%lld  ", i);
40.         fclose(inputfile2);
41.        }
42.       /*write the iteration number & solution to output file*/
43.      }
44.   }
45.   MPI_Reduce(&mySolutions,&globalSolutions,1,MPI_INT,… );
```

Fig. 24 Code snippet of the checkpointed version of the Circuit Satisfiability application

sequential GA was run for 100 generations and the checkpointing was done every 10, 20, and 30 iterations. The performance of the sequential application in which the checkpointing code was generated through the DSL-specifications is comparable to the performance of the application in which the checkpointing code was inserted manually. The code generation process did not cause any significant loss in performance.

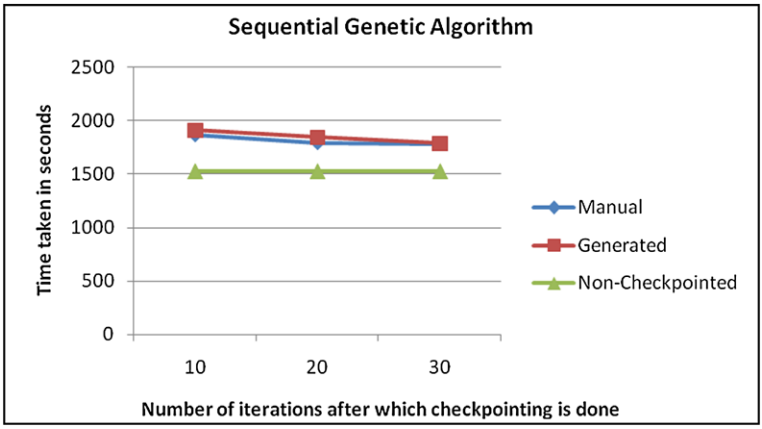

Fig. 25 Run-time comparison for sequential genetic algorithm for CBIR

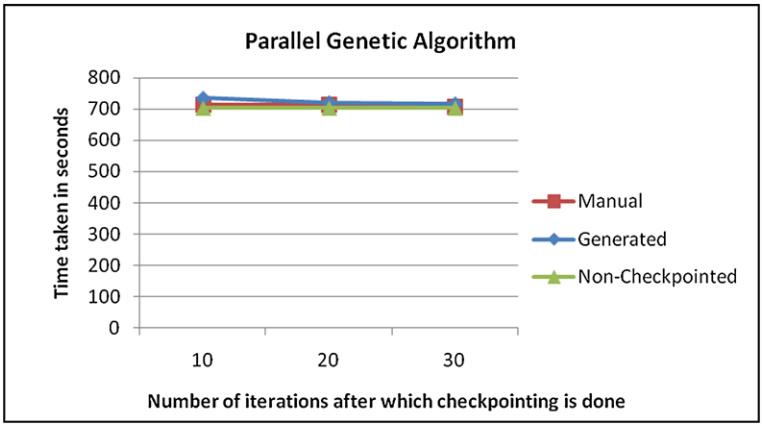

Fig. 26 Run-time comparison for parallel genetic algorithm for CBIR

The parallel version of the GA for CBIR was run for 1000 generations on 50 processors. Because the parallel version of the GA executes very quickly (if run for 100 generations, and with dynamic load-balancing, it takes just 98 seconds to execute on 50 processors), it was run for greater number of iterations to study the impact of checkpointing. The checkpointing was done after every 10, 20, and 30 iterations. A comparison between the manual and the generated version of the checkpointed code of the parallel GA is shown in Fig. 26. The performance of the parallel GA with the generated checkpointing code is comparable to that of the manually checkpointed parallel GA.

Both the sequential and parallel versions of the Poisson's Solver were also checkpointed manually and through the DSL. The applications were run for 50,000 iterations for a 1000×1000 matrix. In both the versions, the convergence is reached after 41,218 iterations. The checkpointing is done after every 3000, 5000, and 10,000 iterations for the sequential version because the execution time is very large. The com-

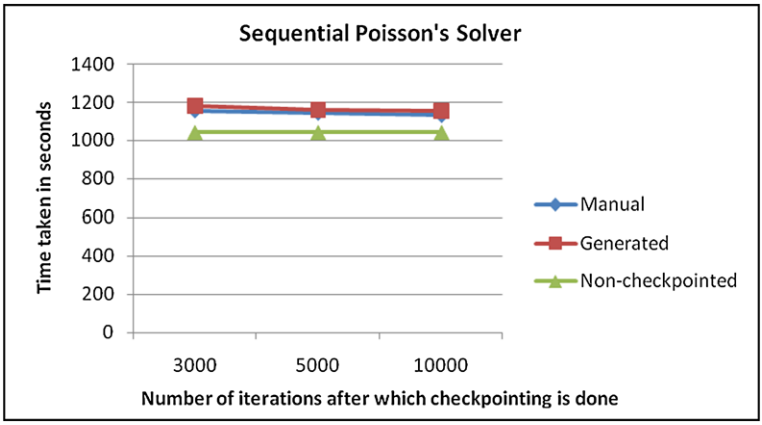

Fig. 27 Run-time comparison of sequential Poisson's Solver

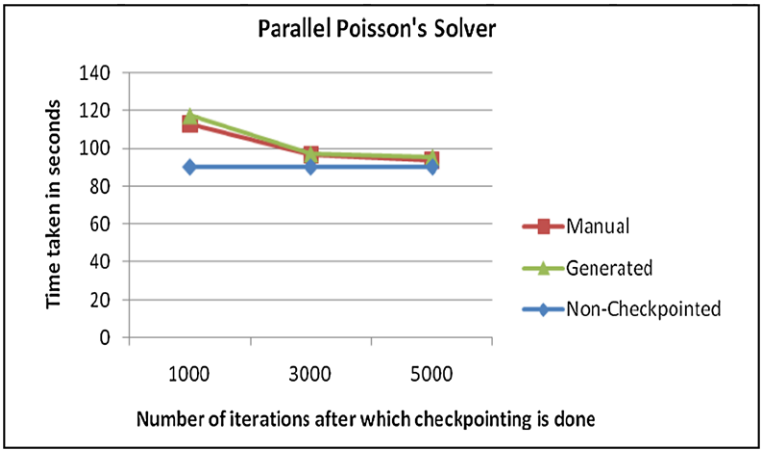

Fig. 28 Run-time comparison of parallel Poisson's Solver

parison chart of the execution time of the sequential version is presented in Fig. 27 and that of the parallel version is presented in Fig. 28. The parallel version was run on 40 processors and the frequency of checkpointing was every 1000, 3000, and 5000 iterations.

The results of checkpointing the sequential and parallel versions of the Circuit Satisfiability application are presented in Figs. 29 and 30. The parallel version of the application was run on 10 processors with 30 input bits. The total number of solutions that satisfied the circuit was 1920. The checkpointing was done every 10,000, 20,000, and 30,000 iterations in the parallel version. In the sequential version of the application, the checkpointing was done every 30,000, 50,000, and 60,000 iterations.

The performance of the version in which the CaR mechanism was generated through the DSL is within 5% of the version in which the CaR mechanism was inserted manually for all the test cases used in this research. The difference between the

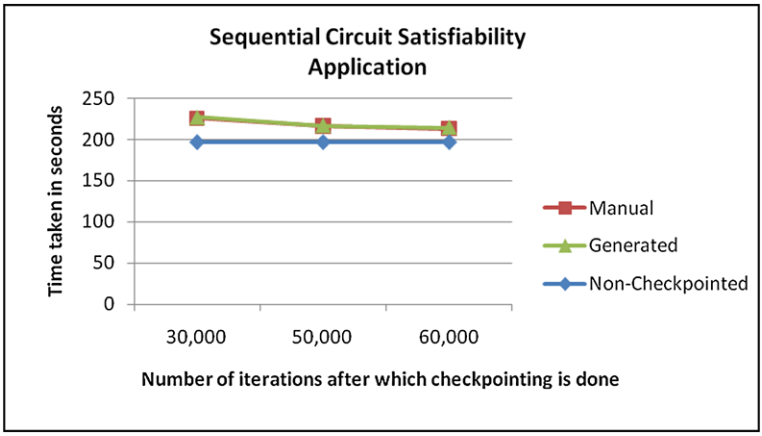

Fig. 29 Run-time comparison of sequential Circuit Satisfiability application

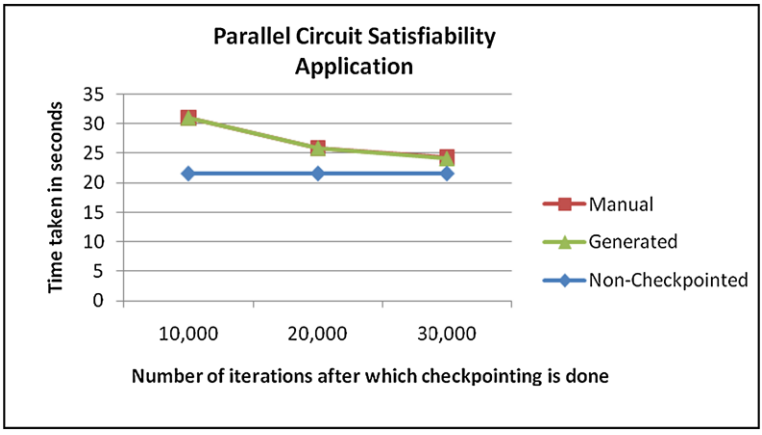

Fig. 30 Run-time comparison of parallel Circuit Satisfiability application

performance overheads of the generated and manually written code (which is maximum 5% in the worst case) seems to be less apparent if the code is run for very large number of iterations and the checkpointing is done at a very low frequency. Though the GUI developed in this research can be used for generating the CaR-specifications (i.e., the DSL code) in a wizard-driven manner, the end-user can also specify the DSL code manually. The CaR specifications written for the sequential version of some types of applications can be reused, with a single modification, for the parallel version as well. For example, in case of the GA, the only difference in the DSL code for the sequential and parallel versions was in specifying the `CaRType`. Hence, the end-user effort is reduced in terms of increase in code reuse.

The CaR code could have also been generated directly through the PTE without using the DSL. However, the time and complexity involved in learning and using the PTE necessitated a higher level of abstraction and the DSL developed in this re-

search provides the same. Inserting the CaR mechanism using the DSL and the PTE is a cost-effective and hardware-independent option for non-invasive reengineering of large legacy applications to make them fault-tolerant. The problems related to maintaining different copies of the application are also overcome and it is easy to evolve the application. Because the original application does not undergo any restructuring, the readability and understandability of the legacy application are also maintained. The checkpointed application can be migrated from one resource to another without affecting the accuracy of the results [9]. If the resources are comparable, no significant loss in performance is observed [9]. This DSL-based ALC-technique can be extremely useful in dynamic environments, like the grid, where small size of checkpoints and platform-independence are of prime importance [29].

6 Conclusion and future work

This research demonstrates a non-invasive technique for ALC of existing applications by using a combination of a DSL and a PTE. The DSL is used in the front-end for capturing the CaR-specifications. The PTE works in the backend to generate and insert the code for CaR into the applications on the basis of the specifications provided through the DSL code. This research is relevant for checkpointing both sequential and parallel applications and is independent of the underlying machine-architecture. The differences between checkpointing the sequential and parallel implementations of an application are illustrated in Sect. 4 of the paper. The DSL has been updated since [25] was written and the facility of taking distributed and centralized checkpoints has been provided now.

During the checkpointing process, the accuracy of the results and the original program structure are maintained with the exception that the additional advantage of fault-tolerance is provided in the applications. The performance of the application with the checkpointing code generated through this approach is comparable to the performance of the application in which the CaR code is inserted manually. If a parallel application requires centralized checkpointing, it is the end-user's responsibility to make sure that the checkpoint is taken only after the processors are in a synchronized or a consistent global state. Therefore, no facility to monitor and save messages in global context is required in the DSL per se.

The DSL for capturing CaR-specifications promotes code reusability, correctness (checkpointing code exists as tested components) and expressiveness. This approach is also helpful in resolving the major issues identified previously in the paper (i.e., the checkpointing should exist as a pluggable feature, invasive reengineering of the legacy applications should be eliminated, and the specifications should be separated from the implementation). Because the CaR-specifications are decoupled from its implementation, instead of using the code generator for inserting the checkpointing code into the application, off-the-shelf ready-to-use checkpointing libraries (e.g., [15]) can also be used in the backend. In which case, the CaR-specifications can be translated into the library calls.

The wizard-driven GUI for capturing the CaR-specifications helps the end-user by alleviating the need to learn the DSL syntax. The wizard-driven GUI will be improved

in future so that the end-user can provide the CaR-specifications with reduced effort. Efforts are under way to optimize the fault-tolerance mechanism. The approaches to integrate the schemes for optimizing the checkpointing-related I/O [8, 30] are being explored and are part of the future work. In [30], an analytical model of the checkpointing process has been proposed on the basis of the mean-time-to-failure of the system, amount of memory being checkpointed, I/O bandwidth, and the frequency of checkpointing. Such an analytical model can be integrated in the approach presented in this paper for suggesting an optimal frequency of checkpointing to the end-user.

Acknowledgements We would like to thank Dr. Jeff Gray, Dr. Frédéric Jouault, and Dr. Suman Roychoudhury for their guidance on the model-driven engineering, usage of the PTE and metamodeling used in this research. We are also grateful to Ms. Saraswathi Mukkai for helping in the coding required for the wizard-driven GUI developed in this research.

This work was made possible in part by a grant of high performance computing resources from the Department of Computer and Information Sciences at the University of Alabama at Birmingham, the School of Natural Sciences and Mathematics at the University of Alabama at Birmingham, and the National Science Foundation Award CNS-0420614. We are also grateful to the Alabama Supercomputer Center for providing us the computational resources required for developing and testing parts of this work.

References

1. Das R, Qian B, Raman S, Vernon R, Thompson J, Bradley P, Khare S, Tyka M, Bhat D, Chivian D, Kim D, Sheffler W, Malmström L, Wollacott A, Wang C, Andre I, Baker D (2007) Structure prediction for CASP7 targets using extensive all-atom refinement with Rosetta@home. Proteins 69(S8):118–128
2. Chen Q, Laminie J, Rousseau A, Temam R, Tribbia J (2007) A 2.5 model for the equations of the ocean and the atmosphere. Anal Appl 5(3):199–229
3. Prvulovic M, Zhang Z, Torrellas J (2002) Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In the Proceedings of international symposium on computer architecture, pp 111–122
4. Duell J (2005) The design and implementation of Berkeley Lab's Linux checkpoint/restart. Lawrence Berkeley National Laboratory, Paper LBNL-54941. http://crd.lbl.gov/~jcduell/papers/blcr.pdf
5. Litzkow M, Tannenbaum T, Basney J, Livny M (1997) Checkpoint and migration of Unix processes in the condor distributed processing system. Technical report 1346, University of Wisconsin-Madison Computer Science Technical Report #1346
6. Bronevetsky G, Marques D, Pingali K, Stodghill P (2003) Automated application-level checkpointing of MPI programs. In: Symposium on principles and practice of parallel programming (PPOPP 2003), pp 84–94
7. Bronevetsky G, Marques D, Pingali K, Szwed PK, Schulz M (2004) Application-level checkpointing for shared memory programs. In: Architectural support for programming languages and operating systems (ASPLOS 2004), pp 235–247
8. Bronevetsky G, Daniel M, Pingali K, Radu R (2008) Compiler-enhanced incremental checkpointing. In: Languages and compilers for parallel computing: 20th international workshop, LCPC 2007, pp 1–15
9. Arora R, Bangalore PV (2008) Using aspect-oriented programming for checkpointing a parallel application. In: Parallel and distributed processing techniques and applications conference, Las Vegas, Nevada, pp 955–961
10. Haines J, Lakamraju V, Koren I, Krishna CM (2000) Application-level fault tolerance as a complement to system-level fault tolerance. J Supercomput 16(1–2):53–68
11. Walters JP, Chaudhary V (2006) Application-level checkpointing techniques for parallel programs. In: International conference on distributed computing and Internet technologies (ICDCIT 2006), pp 221–234
12. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier J-M, Irwin J (1997) Aspect-oriented programming. In: ECOOP'97—object-oriented programming, 11th European conference. Lecture notes in computer science, vol 1241. Springer, Berlin, pp 220–242

13. Czarnecki K, Eisenecker U (2000) Generative programming: methods, tools, and applications. Addison-Wesley Professional, Reading
14. Ramkumar B, Strumpen V (1997) Portable checkpointing for heterogeneous architectures. In: 27th International symposium on fault-tolerant computing—digest of papers, Seattle, WA, pp 58–67
15. Jiang H, Chaudhary V (2002) MigThread: compile/runtime support for thread migration. In: Proceedings of international parallel and distributed processing symposium, IPDPS 2002, pp 58–66
16. Czarnul P, Fraczak M (2005) New user-guided and ckpt-based checkpointing libraries for parallel MPI applications. In: Proceedings of Euro PVM/MPI 2005, 12th European PVM/MPI users' group meeting. Lecture notes in computer science, vol 3666. Springer, Berlin, pp 351–358
17. Harbulot B, Gurd J (2004) Using AspectJ to separate concerns in parallel scientific Java code. In: Proceedings of the 3rd international conference on aspect-oriented software development, Lancaster, UK, pp 122–131
18. Roychoudhury S, Jouault F, Gray J (2007) Model-based aspect weaver construction. In: 4th International workshop on language engineering (ATEM), held at MODELS 2007, Nashville, TN, pp 117–126
19. Mernik M, Heering J, Sloane AM (2005) When and how to develop domain-specific languages. ACM Comput Surv 37(4):316–344
20. Kalaiselvi S, Rajaraman V (2000) A survey of checkpointing algorithms for parallel and distributed computers. Sadhana 25(5):489–510
21. Message Passing Interface Forum (1998) MPI2: A message-passing interface standard. Int J Supercomput Appl High Perform Comput 12(1/2):1–299. Special Issue
22. Baxter I (1992) Design maintenance systems. Commun ACM 35(4):73–89
23. Jouault F, Kurtev I (2005) Transforming models with ATL. In: Model transformations in practice workshop at MoDELS, Montego Bay, Jamaica, pp 128–138
24. Arora R, Mernik M, Bangalore P, Roychoudhury S, Mukkai S (2008) A domain-specific language for application-level checkpointing. In: International conference on distributed computing and Internet technologies (ICDCIT 2008), New Delhi, India, pp 26–38
25. Arora R, Bangalore P, Mernik M (2009) Developing scientific applications using generative programming. In: 2009 International conference on software engineering workshop on software engineering for computational science and engineering, Vancouver, Canada, pp 51–58
26. Chengcui Z, Xin C (2005) Region based image clustering and retrieval using multiple instance learning. In: Image/video annotation and clustering. Lecture notes in computer science. Springer, Berlin/Heidelberg, pp 194–204
27. Chung TJ (2002) Computational fluid dynamics, 1st edn. Cambridge University Press, Cambridge
28. Quinn M (2004) Parallel programming in C with MPI and OpenMP. McGraw-Hill, New York
29. Krishnan S, Gannon D (2004) Checkpoint and restart for distributed components in XCAT3. In: Proceedings of the fifth IEEE/ACM international workshop on grid computing (GRID 2004), pp 281–288
30. Subramaniyan R, Grobelny E, Studham S, George AD (2008) Optimization of checkpointing-related i/o for high-performance parallel and distributed computing. J Supercomput 46(2):150–180