J Supercomput (2014) 68:1378–1401
DOI 10.1007/s11227-014-1091-3

The BonaFide C Analyzer: automatic loop-level characterization and coverage measurement

Sergio Aldea · Diego R. Llanos · Arturo Gonzalez-Escribano

Published online: 21 January 2014
© Springer Science+Business Media New York 2014

Abstract The advent of multicore technologies has increased the interest in parallelization techniques for existing sequential applications. These techniques include the need of detecting loops that are good candidates for parallelization, and classifying all variables of these loops according to their use, a task surprisingly hard to be carried out manually. In this paper, we introduce the BonaFide C Analyzer, an XML-based framework that combines static analysis of source code with profiling information to generate complete reports regarding all loops in a C application, including loop coverage, loop suitability for parallelization, a classification of all variables inside loops based on their accesses, and other hurdles that restrict the parallelization. This information allows to analyze how particular language constructs are used in real-world applications, and helps the programmer to parallelize the code. To show the features of the framework, we present the results of an in-depth loop characterization of C applications that are part of the SPEC CPU2006 benchmark suite. Our study shows that 47.72 % of loops present in the applications analyzed are potentially parallelizable with existent parallel programming models such as OpenMP, while an additional 37.7 % of loops could be run in parallel with the help of runtime speculative parallelization techniques.

Keywords Automatic parallelization · Code analysis · Compiler framework · Profiling information · Source code representation · Source transformation · XML

S. Aldea (✉)· D. R. Llanos · A. Gonzalez-Escribano
Dpto. Informática, Universidad de Valladolid, Campus Miguel Delibes, 47011 Valladolid, Spain
e-mail: sergio@infor.uva.es

D. R. Llanos
e-mail: diego@infor.uva.es

A. Gonzalez-Escribano
e-mail: arturo@infor.uva.es

1 Introduction

Multicore technologies have increased the peak performance of computing systems during the last decade. However, unlike previous advances in computer architecture, existent code cannot immediately take advantage of these architecture improvements. To fully exploit multicore capabilities, programmers should parallelize their applications, a difficult task that requires an in-depth knowledge of both the application and the underlying computer architecture [10].

Fortunately, there exist different shared-memory parallel programming models that aim to facilitate parallel programming, being OpenMP [11] the most popular one. With OpenMP, the programmer can exploit loop-level parallelization by simply adding an `OMP PARALLEL` directive just before the target loop.

However, and despite their usefulness, these parallel programming models need the programmer to address two critical issues. The first one is the decision of which loop is more profitable to be parallelized. To answer this question, it is necessary to know the percentage of the total execution time consumed within each loop of the application, known as the *loop coverage* [35]. Loops which represent a significant amount of execution time compared with the total execution time of the program are usually good candidates, because their effective parallelization may lead to a significant improvement in the execution time of the whole program. Since this information usually depends on the application control flow as well as its input data, the loop coverage cannot be obtained with static analysis alone. Thus, auxiliary profiling tools that return loop coverage are required.

Once a candidate loop has been chosen, programmers face a second problem: To ensure that the loop can be safely run in parallel. Informally speaking, only loops whose iterations do not depend on other iterations can be parallelized. To ensure that the code can be run in parallel, the programmer should be able to classify all variables present in the code into “private” variables (i.e., variables that are always written in an iteration before being used in the same iteration), and “read-only shared” variables, that are only read and not written in any iteration. If all variables inside a loop are either private or read-only shared, then the loop can be safely parallelized[1]. Figure 1 shows an example of such parallelizable loop. If a single variable is found that does not fit in these two categories, then the loop is not parallelizable at compile time, and we have to draw on other techniques such as software-based speculative parallelization. It is easy to see that this *dependence analysis* is a tedious and error-prone task, difficult to be done by hand if the target loop has more than a few dozen lines of code.

In this paper, we address the problem of obtaining the characterization and coverage of target loops automatically. To do so, we have developed an experimental framework that solves both issues, merging static analysis with dynamic information. Our framework, whose preliminary version was presented in PDP’11 [2], transforms the source code of a C application into a single XML [8] tree, in which every element of the source code is represented using XML nodes and attributes. Our framework, partially based on the Cetus source-to-source C compiler [21], works as follows:

[1] Further analysis may be required to ensure that, after parallel execution, final values stored in private variables meet sequential semantics.

```
for ( i = 0; i < 100; i++ ) {  // i controls the loop and it is private
   v[i] = a[i] + i;  // v is private (only written), and a is read-only shared
}
```

Fig. 1 Example of a loop with private (`i` and `v[]`) and read-only shared (`a[]`) data structures

1. We have extended Cetus to develop a new tool called XMLCetus, that generates an XML tree of the sequential source code based on Cetus Intermediate Representation (IR).
2. This XML tree is then automatically augmented with profiling information obtained by running the sequential code.
3. The resulting XML tree is later explored using XPath [6] capabilities, to perform different analyses, including the characterization of all loops in terms of coverage, together with the definition and use of all variables inside all loops of the application.

The final result is a complete report regarding all loops in the application, including loop coverage, loop suitability for parallelization with OpenMP directives, and a classification on the definition and usage of all variables inside all loops. Besides, Bonafide C Analyzer (BFCA) is designed to locate and quantify some hurdles that affect the parallelization. Thus, these reports can also be used to guide the automatic parallelization of the code.

In order to evaluate our approach, we have conducted an extensive study of the C applications present in the SPEC CPU2006 benchmark suite [29]. The study not only characterizes in both quantitative and qualitative terms the loops of these applications regarding their suitability for parallel execution, but it also reports to what extent the use of automatic parallelization techniques may help to further reduce the execution time. The study also classifies all loops in these benchmarks according to different characteristics that may affect their parallelization, including the use of pointer arithmetic, I/O and memory management calls, and dependencies of static and global variables, together with their aggregate coverage. This kind of information, extremely hard to obtain by other means, can also be used to guide future developments in the field of automatic parallelization.

The main contributions of this paper are the following:

- This work combines the compile-time analysis and the loop-based, runtime profiling information of a source code in a single, XML-based representation. As far as we know, this approach is unique with respect to the related work described in Sect. 2, and helps to close a gap described in the literature, since traditional profilers focus primarily on functions and inner loops [45].
- The resulting XML tree can be easily used for a variety of purposes. We have used the combined static and dynamic information of the code to characterize all loops with respect of the loop-based parallelization opportunities they offer, not only in terms of loop coverage but also with respect to the possibility of a dependence violation among iterations.
- The flexibility offered by the XML representation allows to extend this framework for other purposes, such as automatic source code optimization.

The rest of the paper is organized as follows:

Section 2 describes some related approaches. Section 3 describes the overall architecture of the framework developed, together with its components.

Section 4 presents some background knowledge needed to understand Cetus, and how we have modified it to build the XML tree. Section 5 shows how to merge the XML tree that represents the original program with the profiling information obtained after a test run of the application. Section 6 describes how the augmented XML tree can be queried to obtain both the coverage of *FOR* loops and the definition and use of all relevant variables inside each loop. Section 7 makes an introduction to XSLT capabilities, and how template rules can be used to transform the XML tree back to C. Section 8 shows a detailed analysis of the opportunities for parallelization in the C applications of the SPEC CPU2006 benchmark suite. Section 9 briefly compares the capabilities of our framework with respect to the use of Cetus for the same purpose. Section 10 compares the performance of our framework with respect to previous approaches, including Cetus. Finally, Sect. 11 concludes our paper.

2 Related work

The related work can be subdivided in two branches. One branch is focused in parallelism discovery into sequential code, while the other explores ways to represent code using XML.

2.1 Parallelism discovery and loops selection

A correct selection of loops to be parallelized can have noteworthy benefits in the overall performance of the applications. In order to get an accurate source code parallelization, the profiling information has been proved as an invaluable tool to achieve a correct loop selection [55]. Although extracting and using correctly this information has a performance penalty, and includes scalability problems [36], many works have shown that it is not pointless. For example, obtaining an optimal loop selection, Wang et al. [55] get speedups of 20 % in SPEC2000 integer benchmarks, and Packirisamy [46] reports speedups of 60 % in SPEC CPU2006 benchmarks.

Some recent papers present results about which loops have to be selected in a speculative parallelization context. Focusing on hardware-based approaches, there are many that benefit from the selection of loops [12,30,33,37,38,56]. Johnson et al. [33] make the selection of loops and the decision on the number of threads to execute them at the same time the profile run executes. Following a different approach to the problem, Luo et al. [38] estimate the parallel performance of each loop in terms of the probability and cost of speculation failure. BFCA is not only suitable to detect speculative parallelization niches, but also reveals hurdles that may affect any kind of parallelization. Finally, POSH [37] is a compiler targeted to hardware-based TLS architectures. POSH uses a compiler pass to discard ineffective loops on the basis of some heuristics that are previously calculated. The profiling information returned by POSH is related to the use of hardware resources in those architectures, such as cache and register usage. By contrast, the data collected by our solution are not related to hardware resources, but to the source code itself.

Other approaches rely on cost models which use the information extracted by a profiler to select loops, on the basis of the density of data dependence [48,58], the cost of re-executing iterations due to dependence violations [24], the Amdahl's law [9], the different overheads of a parallel execution [23], the frequency of dependencies [52], or speedup estimations based on graphs [27,55]. BFCA does not use theoretical cost models. Instead, it characterizes loop coverages using real executions and classifies based on the potential dependence violations, with the aim of guiding programmers to parallelize the code using this information.

As many of the papers of the literature expose, pure static loop analysis is insufficient. This kind of analysis requires complex models and results in inherent inaccuracy estimations. As a direct consequence, a lower number of loops are parallelized [36]. That is the reason why we propose to complete static information with dynamic information, obtained through profiling, which has been demonstrated as a very efficient technique to improve loop selection. As a novelty feature, BFCA goes further, not only pointing which loops are better to be parallelized, but also locating which hurdles presented in a code may affect the parallelization.

2.2 XML representation

BFCA takes advantage of the benefits derived from using XML. With XML, we can directly represent, analyze and manipulate the program structure. As McArthur et al. [43] pointed out, as long as the granularity of the details of the source code in the XML is higher than in plain text representation, it is possible to create a huge variety of tools to manipulate, transform and extract information from source codes. There are several examples of these useful tasks, such as counting all the occurrences of a syntactic construct, even in a particular context; finding the number of functions called by a function; or finding the number of functions calling a particular function. Such tasks cannot easily be done with plain text representations. Examples of more advanced tasks are refactoring [18,44], exchanging [7,31], differencing [14], program slicing [28], generation of UML models [50], addressing source code [15], source code transformations [19], or fact extraction [16,17].

There are some works that use XML to represent source code to extract some information. One of the first XML representations of source code is JavaML [5], used to describe Java source codes. Like BFCA, JavaML directly represents the structure of source codes by nesting XML nodes, not preserving formatting information. Other approaches do store formatting information, such as JavaML 2.0 [22], srcML [40], XSDML [42], JaML [26] and PALEX [39]. In consequence, storing all this information requires much more space than BFCA, which only preserves structural information of the code. Preservation of formatting information is not a priority to our framework, since it is focused on the analysis of source code.

BFCA only needs the Abstract Syntax Tree (AST) that represents the code for its purposes, and thus, BFCA's XML files only represent explicitly the entire AST, removing the formatting information such as OOML [41] and Zou's and Kontogiannis' proposal [59]. Other data, including the flow information, are implicitly stored. This feature allows BFCA to save memory resources, unlike other approaches, such as the

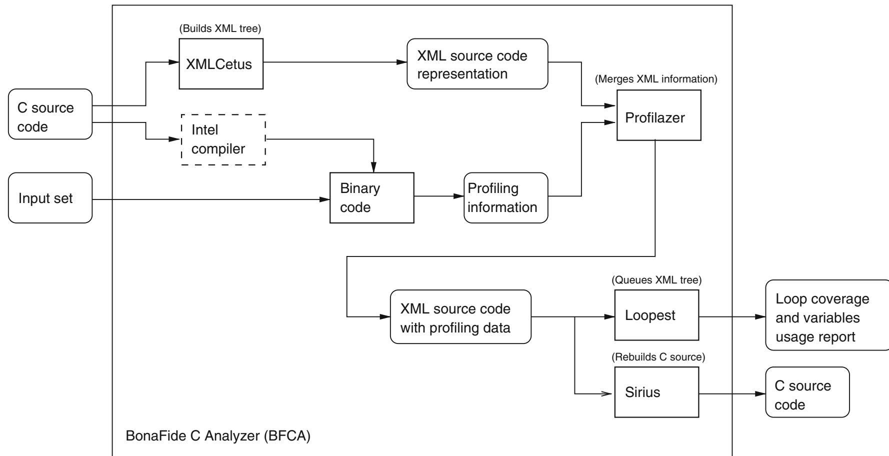

Fig. 2 Architecture of the BonaFide C Analyzer

XREF model [4], which stores the relations across multiple files; ACML [28], which generates XML trees more than a hundred times larger than the original source codes to store all the syntactic and semantic information; or the proposals of Al-Ekgram and Kontogiannis [1], and Putro and Liem [49], which represent higher level abstractions that are not needed by the BFCA operation such as the Control Flow Graph (CFG), the Program Dependence Graph (PDG), and the Call Graph.

On the other hand, there are approaches that store partial ASTs, only marking selected nodes, such as XMLizer [43], or Cordy's proposal [20]. These approaches generate XML trees that are smaller than BFCA's, but they do not contain all the information needed to the kind of analysis that BFCA does. In [51], Sun et al. propose to obtain the XML representation directly and faster than using an AST or a compilation process. This solution is not applicable in our approach, since we have based the XML transformation on the Cetus IR obtained through compiling the source code.

Finally, to avoid the low scalability associated with the bottom-up parsers, and presented in some proposals such as Power's and Malloy's [47], BFCA follows a top-down approach. Moreover, BFCA is not graph-based, as GraX [25], GXL [31], or Wagner's et al. [54] approaches, which are not intended to represent the exact program code, but instead its higher-level structure. This level of representation is closer to others, for example control flow graphs.

After briefly describing the relevant literature, we will now discuss the architecture of BFCA and all its components (XMLCetus, Profilazer, Loopest and Sirius) in more detail.

3 BFCA framework architecture

Figure 2 shows the architecture of the BFCA. The inputs of the framework are the C source files and an example input set. The original C code is used in two ways. A module called XMLCetus builds an XML tree representing the original C

code, with all the information needed to later rebuild the source code. XMLCetus is a modified version of Cetus [21], a source-to-source compiler infrastructure. XMLCetus extends Cetus' functionality by building an XML representation of Cetus' Intermediate Representation (IR) tree.

Using XML to represent source code has several advantages. Besides being simple, extensible and a standard format to exchange data, XML is well suited for representing hierarchical data. Thus, the structure of the source code is explicitly reflected in the nesting of elements into the XML document. This structure can be easily explored using XPath queries, and even transformed by XSLT rules. Exploiting these technologies, we merge static analysis with profiling information.

In order to obtain runtime information, the C code is compiled with the Intel® C compiler. The reason to use this commercial compiler to instrument codes instead of open-source solutions is that the Intel compiler has a feature not available elsewhere: the ability of providing profiling information on each loop in an XML format, in which each loop is represented by a node, and its attributes are used to store relevant information, including the *inclusive* and *exclusive* execution times consumed by each loop[2]. This differs from what other profilers do, as Sun's or OProfile, which generate a text file with only the instructions and functions of the source file, annotated with execution times. This text file needs to be post-processed to obtain the loop coverage, and information about how these loops are nested. This is an unnecessary step if we use the XML file generated by the Intel® compiler.

A second module, called Profilazer, receives the XML file generated by XMLCetus and, using the profiling information for each loop, augments the XML tree with the inclusive and exclusive execution times of every *FOR* loop, together with the number of executions of all loops in the code. Merging the XML representation of the source code with the execution times of the *FOR* loops provides useful information about the structure and nesting of these loops.

This augmented XML tree is received by a third module called Loopest. This module uses a collection of XPath expressions to query the XML DOM tree. We have implemented queries that perform a dependence analysis of scalar variables, arrays, structures, and function parameters, also looking for other constructs (such as memory management, I/O function calls, pointer arithmetic, static variables) in the context of every single *FOR* loop. These queries generate a complete analysis report that can be used to either parallelize the application using OpenMP directives or to guide the development of new automatic parallelization tools. Thus, as a direct application of these reports, Loopest is able to augment the XML tree with additional statements, and hence, allowing the instrumentation of the code using the extracted information. This opens a door to the automatic parallelization of the code, by either inserting OpenMP directives or special code constructs to handle the speculative parallelization of promising loops.

Finally, we have developed a tool that converts the XML tree back into C language, to check the correctness of the process, and to take advantage of the possible aug-

[2] Inclusive execution time of a loop is the amount of time that the loop consumes, including the time spent by its nested loops and functions called from this loop. By contrast, exclusive execution time of a loop is the time that the loop consumes by itself, excluding the time spent by its nested loops and function calls.

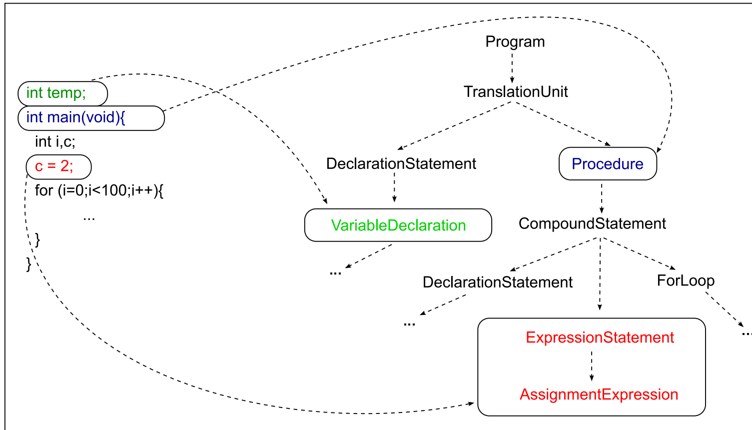

Fig. 3 IR tree structure example. Each part of the source code corresponds with a node in the tree. A "TranslationUnit" is a file containing source code

mented codes produces by Loopest. We have developed a module that performs such transformation, called Sirius. The code generated by Sirius is equivalent to the original code.

One of the main advantages of using BFCA for program analysis instead of other alternatives (including Cetus) is extensibility. New functionalities can be easily added, inserting new XPath queries into Loopest. As we will see in Sect. 9, the development of the XPath queries needed for the present functionalities of BFCA is much simpler than directly modifying Cetus for the same purpose.

4 XMLCetus: building the XML tree

BFCA has a modular architecture composed by four components. XMLCetus is the first component and it is based on Cetus. Cetus [21] is a compiler infrastructure written in Java for source-to-source transformation of C programs developed by Purdue University. Cetus builds an Intermediate Representation (IR), an abstract representation that holds the block structure of a C program. The IR is implemented in the form of a class hierarchy and accessed through its class member functions. Figure 3 shows an example of Cetus IR from a C source code.

Although Cetus is a powerful tool, adding new functionalities requires an in-depth knowledge of Java, Cetus IR, and its associated data structures. Due to both simplicity and extensibility reasons, instead of using Cetus capabilities to develop our compiler framework, we modify it to build an XML representation of its IR, and we use XML standard tools to perform queries and modifications to the structure.

XMLCetus is a modification of Cetus that generates an XML DOM tree based on Cetus IR. The main changes to Cetus are made just after Cetus has finished the analysis of the C source and has generated the IR. At this point, the XML tree is created having the *Program* node as first node, and traversing the Cetus IR in pre-order, depth-first search. Every node of the IR will have a corresponding representation in the XML

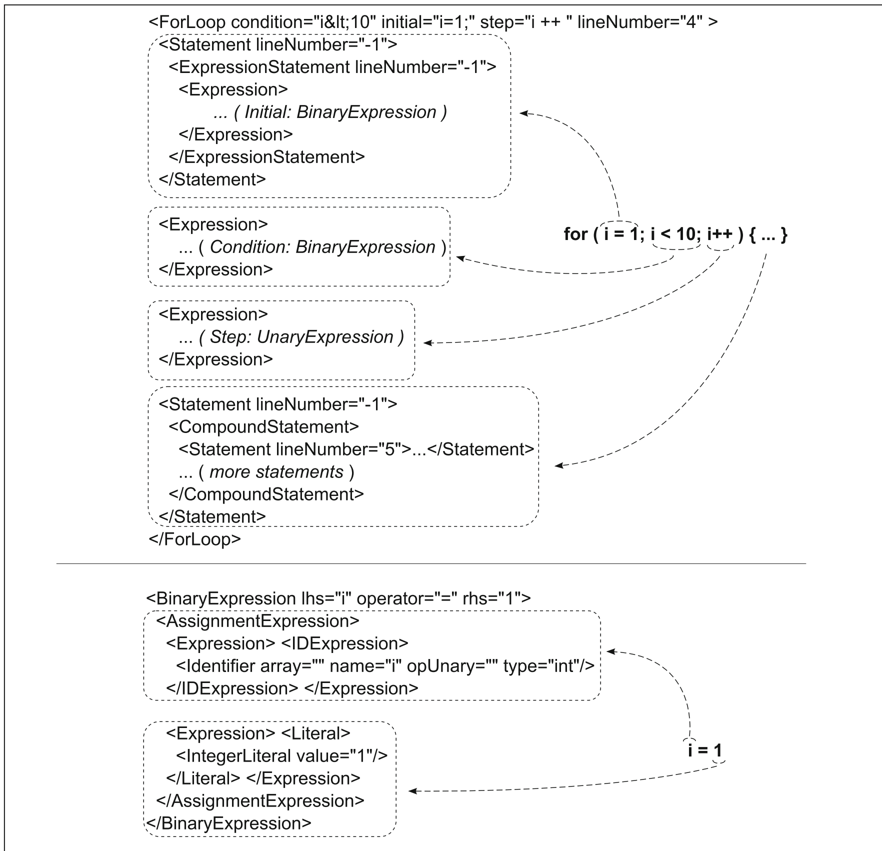

Fig. 4 XML code generated by XMLCetus for a *FOR* loop statement (*top*), and the binary expression that represents the loop initialization (*bottom*)

DOM tree, thus preserving the original structure of the Cetus IR, as well as the name of the IR elements. This procedure generates an XML document that represents the DOM tree, and it is printed into a new XML file. Figure 4 shows an example of this XML generated. A more detailed description of this process can be found in [2].

5 Profilazer: augmenting the XML tree with profiling information

The output XML file generated by XMLCetus is passed to Profilazer, which uses the profiling information provided by the Intel® compiler to generate a new XML tree that is augmented with the inclusive and exclusive execution times of every *FOR* loop in the original source code.

It is not straightforward to calculate the coverage of every loop in the code in terms of execution times. Most profiling tools only return execution times consumed by function calls and C statements. Fortunately, the Intel® compiler has a *-profile-*

loops=all option, that allows the executable file to generate an XML report with both inclusive and exclusive execution times of every loop. Since we already have an XML file describing the source code, combining both XML trees is conceptually straightforward. This combination is made by Profilazer. Both XML trees contain XML nodes representing the *FOR* loops with attributes that identify them properly. Therefore, for each of these nodes in the XML tree generated by XMLCetus, Profilazer assigns the execution times of the corresponding *FOR* loop encoded in the XML tree generated by the profiler.

Obtaining the relevance of each loop in terms of execution time is very useful to choose which loop is more profitable to parallelize. An XML tree representing the source code, augmented with profiling information, satisfies this need not only for manual parallelization purposes, but also to build heuristics to automatically choose target loops.

6 Loopest: querying and modifying the XML tree

Loopest is a Java module that analyzes the augmented XML tree and provides three sets of functionalities: (a) generation of reports on the aggregate coverage of every *FOR* loop, (b) generation of reports on the definition and use of all variables in the context of every *FOR* loop, and (c) the ability of modifying the XML tree according to this analysis, inserting automatic parallelization directives.

Loopest relies on XPath capabilities to perform queries on the augmented XML tree returned by Profilazer. XPath syntax is easy to learn, and allows to build complex queries with few words or lines. The result of these queries may be new node-sets that can be combined into new queries. XPath queries work in a similar way than recursive searches in a directory-based file-system structure, allowing to select nodes or set of nodes in an XML document, based on the nodes' attributes. As an example, Fig. 5 shows the queries used in Loopest to isolate variables that are written inside a loop. Such queries are much simpler to develop than directly modifying the Java code that manages the IR structure in Cetus. For example, the first query in Fig. 5 has three different parts that isolate the variable `x` as written in the statements `x = 2`, `int x = 0`, and `function(&x)`, respectively.

The simplicity of XPath syntax allows a fast prototyping of new solutions. As a result, Loopest can be modified to detect other features in a source code by implementing new XPath queries, either to describe new rules in the variables classification, or to detect new languages constructions. This property guarantees the extensibility of Loopest.

In order to classify variable usage inside *FOR* loops, Loopest executes a set of XPath queries that determine the variables being read, written, or read and then written. These results are combined with other queries using set theory operations[3] to classify variables into private, read-only shared, and speculative classes, including the detection of private variables that are used after the end of the loop. We have

[3] This feature has been added with the help of the *ListUtils* package, provided by Apache Commons [32], that allows to apply operations as *union* or *intersection* over sets of variables.

```
// (1) QUERY TO ISOLATE VARIABLES WRITTEN.
// VARIABLES AS INDEXES IN AN ArrayAccess ARE NOT CONSIDERED TO BE WRITTEN.
     Statement[2]/CompoundStatement//
           ( (AssignmentExpression/Expression[1]//Identifier[
               not(ancestor::AccessExpression) and not(ancestor::ArrayAccess)
              ])
           | (VariableDeclarator[ descendant::Initializer ]/
               Expression//Identifier[
                   not(ancestor::AccessExpression) and not(ancestor::ArrayAccess)
              ])
           | (FunctionCall/Expression[ position()!=1 ]//
               UnaryExpression[ @operator='&' ]//Identifier[
                   not(ancestor::AccessExpression) and not(ancestor::ArrayAccess)
              ])
           )

// (2) ArrayAccess WRITTEN.
     Statement[2]/CompoundStatement//
           ( (AssignmentExpression/Expression[1]//
               ArrayAccess/Expression[1]//
                  Identifier[ not(ancestor::AccessExpression) ])
           | (VariableDeclarator[ descendant::Initializer ]/
               Expression//ArrayAccess/Expression[1]//
                  Identifier[ not(ancestor::AccessExpression) ])
           | (FunctionCall/Expression[position()!=1]//
               UnaryExpression[ @operator='&' ]//ArrayAccess/Expression[1]//
                  Identifier[ not(ancestor::AccessExpression) ])
           )

// (3,4) AccessExpression VARIABLES WRITTEN.
// THE OPTION "VARIABLEDECLARATOR + INITIALIZER" IS NOT CONSIDERED BECAUSE THAT
// CONSTRUCTION IS NOT POSSIBLE. (EXAMPLE: long date.r1 = 5).
     Statement[2]/CompoundStatement//
           ( (AssignmentExpression/Expression[1]//
               AccessExpression[ not(ancestor::AccessExpression) ])
           | (FunctionCall/Expression[position()!=1]//
               UnaryExpression[@operator='&']//
                   AccessExpression[ not(ancestor::AccessExpression) ])
           )

// (5) VARIABLES READ AND WRITTEN (FROM UNARY INCREMENTS OR DECREMENTS).
     Statement[2]//ExpressionStatement/Expression//(UnaryExpression
           [ @operator='post ++' or @operator='post --'
               or @operator='pre ++' or @operator='pre --'
          ]) //
           ( (Identifier[ not(ancestor::AccessExpression) ])
              | (AccessExpression[ not(ancestor::AccessExpression) ])
           )
```

Fig. 5 XPath code that searches for variables written inside a loop. This code includes queries which detects *1* variables written as a result of being located at the left-hand side of an assignment expression, *2* writes to array elements, *3* writes to fields inside data structures, *4* variables affected by an address operator, and *5* implicit writes due to unary increments or decrements

used set theory operations to simplify queries, and obtain a better mapping with the classification rules. For example, a variable that is written and also read is detected when we perform the union of the list of variables written and the list of variables read, both obtained through XPath queries. More precisely, private variables are: (1) variables that control the execution of a loop; (2) variables that are always written in an iteration before being read, and are not read after the loop execution; (3) variables that are only written and read after the loop, or written before being read; and (4) data structures that only contains private variables. Shared variables are: (1) variables that

are only read; (2) variables that are involved in the main loop control (but they are not the control variable), as long as they are read-only variables; (3) data structures that all their elements are shared; and (4) global variables that are only read inside the loop. Variables that are not private nor shared are considered *speculative*, because their definition and use may lead to race conditions during a parallel execution. Loopest detects variables that match each subtype described above, and then applies several set operations, such as union or intersection, to obtain the final variable classification.

The XPath queries used by Loopest process all *FOR* loops present in the code, regardless of their depth level, and all user functions called by them. All these data provide the enough information that can be used to guide loop-level speculative parallelization of the code.

As we have stated above, Loopest has three different functionalities. The first one is to use profiling information of every *FOR* loop, provided by the augmented XML file, to perform a classification of loops by their relevance in terms of execution time. The second one is to generate a report with a taxonomy of variable usage for a target loop. Finally, a third functionality offered by Loopest is the possibility of augmenting the XML tree with additional branches, thus allowing the instrumentation of the code using the acquired information. This opens a door to the automatic parallelization of the code, either inserting OpenMP directives or function calls that allow the speculative parallelization of promising loops [13]. Our research group is currently exploring this possibility.

7 Sirius: regenerating the C code

After building and analyzing the XML tree, the last part of the process is to convert this representation back into C code, for testing purposes (i.e., to check whether the generated C file has the same functionality than the original). To do so, we have developed a Java module called Sirius, that receives an XML document describing either the original C code as produced by XMLCetus or the augmented C code produced by Loopest.

Sirius is based on XSLT [34] capabilities, and uses the template rules to correctly translate the XML document back to C. As a result of the transformation made by Sirius, a C source file is generated. Since all formattings (indentation, spaces, and line breaks) have been lost in the process, we use the GNU tool *indent* to format the output file and make it more pleasant for the reader.

To apply the XSLT transformation rules, we have chosen the Saxon tool [53], due to its open-source nature and because it implements XPath and XSLT 2.0.

8 A case study: loop characterization of SPEC CPU2006 C benchmarks

To demonstrate the capabilities of our analysis framework, we will characterize a set of C benchmarks to measure the potential gain that could be obtained with the parallelization of some of their loops. As long as we focus on loop-level parallelization, both the coverage of *FOR* loops in terms of execution time and the definition and use

of all variables inside them should be taken into account. For this study, we will use some of the C benchmarks provided by the SPEC CPU2006 benchmark suite [29].

Our study has several parts. First, we will discuss the opportunities for parallelization of each SPEC CPU2006 C benchmarks considered, classifying *FOR* loops in different categories with respect to their suitability for parallel execution and taking into account the existence of potential dependence violations. Second, a more in-depth analysis is performed to characterize all *FOR* loops with respect to different situations that may affect parallelization, such as the use of pointer arithmetic and variables, memory management function calls, I/O activity, or static variables.

In order to obtain the more accurate results, we use the three input sets that SPEC CPU2006 provides for each benchmark: *Test*, which is used to check for the correct execution of the benchmark; *Train*, which involves a bigger workload and it is used to optimize benchmarks by feedback; and *Reference*, which is the biggest workload set used to obtain execution times and hence, the final performance results. The conditions of use of the benchmarks suite include the mandatory use of these input sets, with the aim of ensuring that the observed level of performance can be reproduced by other researchers. A detailed description for each input set can be found in [29].

Regarding the experimental evaluation, we have run each input set three times and obtained the average percentage. Since the evaluation of the results in terms of precision or recall is not feasible to large source codes (because it requires a manual checking), we have used regression tests designed to cover the whole search space. With these tests, we try to cover every possible situation in a code, manually checking whether the results are correct in these tests.

8.1 Loop characterization with respect to potential dependence violations

Loops are one of the main sources of parallelism because of their repetitive nature. However, not all loops are parallelizable. There are several reasons for this. The most important one is the possibility of the occurrence of dependence violations. This possibility may force the in-order execution of different instructions, thus limiting the amount of parallelism that could be extracted. In fact, parallelizing compilers conservatively refuse to parallelize loops that may incur in dependence violations. Other reasons that limit the amount of parallelism to be extracted include the presence of system calls that should be carried out in order, the use of pointer arithmetic, or memory management functions.

In this section, we will use BFCA capabilities to isolate *FOR* loops that do not present potential dependence violations, and therefore are valid candidates to be parallelized at compile time. Once a candidate loop is detected, programmers can decide to parallelize this loop using standard, shared-memory APIs such as OpenMP. BFCA gives a first estimation of the degree of parallelism that could be extracted with compile-time techniques. This estimation is only an upper bound of that degree, because, as we stated before, there are several additional factors that limit in practice how much parallelism can be obtained. These factors will be examined in the next section.

We have analyzed all benchmarks using the three input sets provided by SPEC CPU2006 for each benchmark, to obtain loop coverages in different circumstances. We

Table 1 Well-formed *FOR* loops

Application	Well-formed *FOR* Loops			
	% of Loops	Coverage test	Coverage test	Coverage reference
401.bzip2	93.33	35.69	29.8	31.43
429.mcf	63.64	7.39	3.69	2
433.milc	64.11	2.5	2.1	1.9
456.hmmer	88.23	94.29	98	98.1
458.sjeng	91.67	19.2	15.9	19.59
462.libquantum	92.13	22.5	19.7	21.4
464.h264ref	95.31	75.69	76.09	77.59
470.lbm	100	96.8	99.8	100
482.sphinx3	80.4	40.8	64.69	81.99
Average	85.42	43.87	45.53	48.22

Table 2 Room for compile-time and runtime parallelization techniques

Application	Well-formed *FOR* loops parallelizable at compile time				Well-formed *FOR* loops potentially parallelizable at runtime			
	% of Loops	Cov. Test	Cov. Train	Cov. Ref.	% of Loops	Cov. Test	Cov. Train	Cov. Ref
401.bzip2	46.67	16.94	7.56	13.71	46.66	18.75	22.24	17.72
429.mcf	30.3	3.5	2	1.2	33.34	3.89	1.69	0.8
433.milc	33.25	1.4	1.3	1.3	30.86	1.1	0.8	0.6
456.hmmer	46.55	18.5	1.8	1.1	41.68	75.79	96.2	97
458.sjeng	51.39	17.5	14.5	17.7	40.28	1.7	1.4	1.89
462.libquantum	48.31	0.2	0.1	0.1	43.82	22.3	19.6	21.3
464.h264ref	57.59	37.6	34.69	39.1	37.72	38.09	41.04	38.49
470.lbm	69.57	94.9	98.7	99.8	30.43	1.9	1.1	0.2
482.sphinx3	45.86	1.8	2.89	4.9	34.54	39	61.8	77.09
Average	47.72	21.37	18.17	19.88	37.7	22.5	27.36	28.34

are aware that the coverage obtained with a particular input set cannot be extrapolated to other input sets, but we believe that this information is still useful to guide the choice of loops to be parallelized. As long as the loop coverage can be heavily influenced by the input set provided, users should select a “representative” input set for their applications.

As an example of the capabilities of the framework, we have accumulated the information regarding each particular *FOR* loop returned by Loopest to show the degree of parallelism present in each application. Tables 1 and 2 summarize the results of the study of our SPEC CPU2006 C benchmarks. For each benchmark considered, the tables show the following information:

```
for (i=0;i<100;i++) {        for (i=0;i<100;i++) {        for (i=0;i<100;i++) {
  b = function(i);             v[i] = a[i] + i;             v[i] = v[function(i)];
}                            }                            }
        (a)                          (b)                          (c)
```

Fig. 6 **a** Example of a "well-formed" *FOR* loop. **b** Loop that only holds private (`i`, and `v[]`) and read-only (`a[]`) data structures. **c** Loop not safely parallelizable at compile time, because the statement can lead to a dependence violation

- Summary of "well-formed" *FOR* loops (Fig. 6a), i.e., loops that (a) have a single control variable, (b) all three fields of the *FOR* structure (initialization, conditional evaluation, and increment) are being used, and (c) perform no changes to the control variable inside the loop body. These loops are much easier to be parallelized than other loops, mostly because the iteration space is known in advance. According to BFCA results, these loops represent more than 85 % of all loops present in the benchmarks considered.
  Table 1 also shows the percentages of execution time that these "well-formed" *FOR* loops represent with respect to the total running time, for each one of the three working sets defined in the SPEC CPU2006 benchmark suite. For some benchmarks, such as 401.bzip2, larger working sets consist of several executions of the same executable with different input files. In these cases, we have calculated the average percentages of all executions. As can be seen in the table, these loops account for 43–48 % of the total running time.
- Summary of loops that only holds private and read-only shared variables; i.e, loops that are valid candidates to be parallelized at compile time (Fig. 6b). As can be seen in the table, roughly half of the loops fall into this category. As we stated before, it might not be profitable to parallelize the smallest loops due to thread management overheads. The table also summarizes their coverage for each input set, accounting for approximately 20 % of the total execution time.
- Summary of loops that cannot be safely parallelizable at compile time (Fig. 6c). This does not imply that these loops must be executed sequentially. Indeed, they usually present some degree of parallelism, but the BFCA analysis has shown a potential dependence violation that prevents them to be parallelized at compile time. In these cases, the use of runtime, software-based speculative parallelization techniques may help to extract their inherent parallelism [13].
  According to our results, speculative parallelization techniques could extract some degree of parallelism from an average 37.7 % of all loops present in the benchmarks considered, covering around 26 % of the execution time on average. These results highlight the importance of runtime techniques to further parallelize sequential applications.

8.2 Loop characterization with respect to parallelization hurdles

As we stated in the previous section, it is not enough for a loop to be free of potential dependence violations to be parallelizable at compile time. There are other parallelization hurdles as well, such as the use of pointer arithmetic and/or the existence

Table 3 Relevance of challenges for parallelization techniques: pointers and memory management

Application	Number of FOR loops	Loops with pointer arithmetic				Loops with memory management			
		%	Cov. Test	Cov. Train	Cov. Ref.	%	Cov. Test	Cov. Train	Cov. Ref
401.bzip2	120	88.33	45.2	41.09	40.88	0.83	0	0	0
429.mcf	33	96.97	66.1	33.19	39.3	0	0	0	0
433.milc	418	87.56	72.2	71.8	72.6	2.39	0	0	0
456.hmmer	739	95.81	94.29	98	98.14	3.52	0.5	0.1	0
458.sjeng	216	10.19	9.4	8	9.6	0	0	0	0
462.libquantum	89	87.64	22.8	19.9	21.4	0	0	0	0
464.h264ref	1,792	88.23	75.89	76.19	77.89	3.23	0.2	0	0
470.lbm	23	78.26	3.5	0.3	0	0	0	0	0
482.sphinx3	556	93.71	49.2	71.89	86.09	0.54	0	0	0
Average	443	80.74	48.73	46.71	49.54	1.17	0.08	0.01	0

Table 4 Relevance of challenges for parallelization techniques: I/O activity and use of static variables

Application	Number of FOR loops	Loops with I/O activity				Loops affected by static variables			
		%	Cov. Test	Cov. Train	Cov. Ref.	%	Cov. Test	Cov. Train	Cov. Ref
401.bzip2	120	7.5	0.6	9.86	1.61	0	0	0	0
429.mcf	33	12.12	2.6	0.8	0	6.06	53.3	26.8	34.2
433.milc	418	19.86	0.1	0.2	0.1	3.11	0.1	0.2	0.8
456.hmmer	739	13.67	0	0.1	0	8.25	5.7	0.4	0.25
458.sjeng	216	4.17	0	0	0	18.98	8.5	6.7	8.6
462.libquantum	89	7.87	0	0	0	26.97	0.4	0	0
464.h264ref	1,792	0.89	0	0	0	8.26	0.9	0.8	0.7
470.lbm	23	43.48	1.3	0.1	0	4.35	0	0	0
482.sphinx3	556	14.2	13.7	6.5	2.6	12.05	1.9	3.6	4.5
Average	443	13.75	2.03	1.95	0.48	9.78	7.87	4.28	5.45

of memory management function calls, that inconveniences the static analysis of the code at compile time; the existence of I/O function calls that should be carried out in order; or the presence of static variables in user-space function calls that forces to a certain execution order to meet sequential semantics. Note that the analysis described here is extremely difficult, if not impossible, to be carried out by other means.

Tables 3 and 4 summarize the results obtained for the different characteristics described above. For each considered benchmark, the table shows the following information:

– Summary of loops that use pointer arithmetic (Fig. 7a). This situation is detected using the data type of the variables present in the loop, described in the field of the XML element that describe a particular variable. The average number of loops

```
int * p, q;             int * p;                                              static s;
for (i=0;i<100;i++)     for (i=0;i<100;i++)     for (i=0;i<100;i++)          for (i=0;i<100;i++)
{                       {                       {                             {
    ...                         ...                     ...                           ...
  p = &q;                p=(int*)malloc(..);     printf(...);                  s = function(i);
    ...                  free (p);                      ...                           ...
                                ...
}                       }                       }                             }
       (a)                     (b)                     (c)                           (d)
```

Fig. 7 **a** Example of a loop using pointer arithmetic. **b** Loop containing memory management functions. **c** Loop containing I/O function calls. **d** Loop affected by static variables

(around 80 %) reflects the importance of supporting pointer arithmetic in compile-time or runtime parallelization schemes, a problem that has not been solved yet in the general case. Execution time coverages of these loops are also high: only 458.sjeng and 470.lbm have coverage values lower than 10 %. Note that this analysis explains why, despite the high coverage of "well-formed", parallelizable loops in the benchmarks considered (as shown in the previous section), parallelizing compilers still obtain only marginal improvements in the execution time of these benchmarks [3].

– Summary of loops that perform calls to memory management functions, such as *malloc()* or *free()* (Fig. 7b). In order to detect this hurdle, we look for XML nodes inside the loop that represent standard, memory-management function calls, such as `malloc()` or `free()`. On average, only 1.17 % of loops make use of dynamic memory capabilities, with a negligible coverage in terms of execution time. This result suggests that dynamic memory management may not be a priority in the list of problems that automatic parallelization techniques should solve to speedup these benchmarks.
– Summary of loops that contain I/O function calls (Fig. 7c). As in the previous case, we perform a search for standard, I/O system calls. Loops with such system calls cannot be parallelized at compile time, and runtime speculative techniques are neither capable of handling speculative I/O calls. Therefore, these loops cannot be parallelized with existent tools. As happens with loops that present memory management function calls, this category of loops is not quite representative, being 13.75 % of the total number of loops, with around 2 % of execution time coverage. These results suggest that the solution to this problem may not be a priority for automatic parallelization techniques.
– Percentage of loops that are affected by static variables (Fig. 7d). These loops are detected searching for static variables being written not only in the loop itself, but also inside the functions called inside the loop body, and recursively, functions called by these functions. These are variables whose lifetime extends across the entire run of the program, and thus, writing values on these variables should be maintained after the *FOR* loop. This circumstance conditions parallelism and it may be taken into account by any automatic parallelization mechanism. Writings on static variables are only contained in a 9.78 % of the loops considered, although for some benchmarks this percentage reaches a 27 %. Their coverage for all input sets considered is not so high, even null or almost zero in some benchmarks as 401.bzip2, 433.milc, 462.libquantum or 470.lbm.

```
FOR loop (file sphinx3.c; line 12504; inclusive time 2.0%; exclusive time 0.2%)
 FOR loop (file sphinx3.c; line 12505; inclusive time 0.1%; exclusive time 0.1%)
  fe_frame_to_fea (file sphinx3.c; line 12510)
   fe_spec_magnitude (file sphinx3.c; line: 12895)
    FOR loop (file sphinx3.c, line 12928; inclusive time 0.0%; exclusive time 0.0%)
     FOR loop (file sphinx3.c; line 12932; inclusive time 0.0%; exclusive time 0.0%)
      FOR loop (file sphinx3.c; line 12939; inclusive time 0.0%; exclusive time 0.0%)
       FOR loop (file sphinx3.c; line 12943; inclusive time 0.0%; exclusive time 0.0%)
        fe_fft (file sphinx3.c; line 12950)
         FOR loop (file sphinx3.c; line 13037; inclusive time 0.0%; exclusive time 0.0%)
          (static int) k (file sphinx3.c)
```

Fig. 8 Excerpt of the report returned by Loopest for 482.sphinx, notifying the use of static variable `k` and describing both the functions and loops affected, together with their coverage

8.3 Characterization of the usage of static variables

Static variables declaration can be easily found with standard Unix tools, such as `grep`. However, until now it was extremely difficult to know which loops and functions were affected by the declaration of a particular static variable. The identification of all loops and function calls affected by the existence of static variables is another example of the powerful capabilities of BFCA framework. This feature can be considered critical when working with applications such as 429.mcf, where loops affected by static variables reach a 53.3 % of coverage for the Test input set.

The BFCA framework includes a Loopest's XPath rule to detect static variables, reporting all *FOR* loops and functions affected by them. An example of such a report can be seen in Fig. 8, showing an excerpt of the report generated by BFCA when analyzing 482.sphinx. The fragment shown indicates in its last line that a static variable `k` has been found, and shows all the *FOR* loops and functions (in our case, functions `fe_frame_to_fea()`, `fe_spec_magnitude()`, and `fe_fft()`) affected by that declaration, together with their inclusive and exclusive coverage. None of these loops or function calls can be safely parallelized without taking this variable `k` into account.

These are just some examples of the studies that can be conducted with our framework. The flexibility provided by XML tools makes easy to modify XPath queries to further investigate the possibility of using emerging parallelization techniques.

9 Why we do not use Cetus to detect variables usage

It is important to highlight the differences between our framework and a system based exclusively on Cetus that detects variables usage. The main differences between both approaches are simplicity and extensibility. Detection of private and read-only shared variables is within Cetus capabilities, but the code required to implement this functionality is much longer and complex than Loopest's code. Modifying Cetus requires a deeper knowledge of Java, Cetus IR, and its associated data structures. In our system, adding new functionalities can be done simply adding new XPath queries, that just requires some basic knowledge about XPath and Java to combine the results into meaningful reports. Using the number of code lines needed as an effort indicator, in

Table 5 Generation times of XML documents for SPEC CPU2006 benchmarks, rates of slowdown respect to Cetus' original execution, and file sizes

Application	Lines of code	Time in seconds		Rate of slowdown	Size on Kilobytes		
		Cetus	XMLCetus		Source	XML	Rate
401.bzip2	7,292	5.67	7.16	1.26	200	4,320	21.60
429.mcf	2,044	2.69	3.33	1.24	40	648	16.20
433.milc	12,837	7.67	10.10	1.32	400	7,560	18.90
456.hmmer	33,210	9.00	12.82	1.42	1,024	13,616	13.30
458.sjeng	13,291	6.33	8.17	1.29	288	5,188	18.01
462.libquantum	3,454	2.99	4.46	1.49	76	1,612	21.21
464.h264ref	46,142	10.64	15.91	1.50	1,448	25,648	17.71
470.lbm	875	4.25	4.77	1.12	36	1,852	51.44
482.sphinx3	18,280	7.09	9.10	1.28	516	7,676	14.88
Average	15,269	6.26	8.42	1.32	448	7,569	21.47

Cetus at least eight Java classes take part directly to locate the private variables of a given loop, representing 2,573 lines of code (calculated with SLOCCount [57]). Meanwhile, Loopest only needs 425 lines of lower-complexity code to carry out the same task, representing a reduction of around 83 %.

Regarding extensibility, making changes to Cetus' functionalities requires also a deep knowledge about Cetus software and its intermediate representation. Changes in Loopest software are much easier, because it is developed with XPath, not even requiring a widespread knowledge about Java or XML. In fact, our framework can be easily adapted to other transformation tasks not directly related with automatic parallelization, just modifying or creating new XPath queries or XSLT transformations.

Finally, the combination of our XML representation of the source code and the profile-based analysis provided by Intel® compiler is straightforward. This greatly increases the possibilities of our framework. As an example, it is possible to extract execution times of every loop with XPath and set new attributes with this information in the same XML tree.

10 BFCA performance considerations

In this section, we will compare the performance of the BFCA framework with other solutions, including Cetus, both in terms of execution time and size of the generated files. To do so, we will analyze the performance of each phase of the BFCA framework separately.

10.1 XMLCetus performance

Table 5 resumes the execution times of both XMLCetus and the original Cetus framework. The right part of the table shows the sizes in Kilobytes of both the original code

Table 6 Execution times of Profilazer and Loopest for SPEC CPU2006 C benchmarks.

Application	Lines of code	FOR Loops	Time in seconds	
			Profilazer	Loopest
401.bzip2	7,292	120	49.66	3.56
429.mcf	2,044	33	3.72	1.73
433.milc	12,837	418	71.78	2.27
456.hmmer	33,210	739	111.09	7.53
458.sjeng	13,291	216	121.49	9.29
462.libquantum	3,454	89	18.68	2.15
464.h264ref	46,142	1792	1,064.19	45.92
470.lbm	875	23	5.32	5.16
482.sphinx3	18,280	556	227.11	7.99
Average	15,269	443	185.89	9.51

and its corresponding XML representation for each analyzed benchmark. As expected, the larger the source code, the longer the XMLCetus takes to transform the source file into an XML-based representation. However, a single line can be very simple, thus generating a few XML nodes, or very complex, generating a large number. Therefore, complexity of source codes also affects the generation times. This explains the lower time consumed by benchmarks that have more code lines.

Differences between the time needed by XMLCetus and Cetus are not so large, with a rate of slowdown of 1.32 on average for SPEC CPU2006 benchmarks, with a peak value of 1.50 for 464.h264ref. These differences, which do not alter the user experience, are significantly reduced respect to results described in Power and Malloy's work [47], where the rate of slowdown reaches an 25.2 on average for different benchmarks.

Regarding file sizes, as Power [47], Maruyama [42], and Maletic [40] remarked, XML-based representations of source codes require larger files to contain them, because of all tags, attributes, and the expanded way of representing source structures that detail each particular element. XMLCetus creates XML files that are on average 21.5 times larger than original source code documents for SPEC CPU2006 benchmarks. XML file sizes generated by BFCA are bigger than files generated by McArthur's et al. [43], where XML files are 6.25 times larger on average, because they only represent partial ASTs, not the entire AST as BFCA does. BFCA also generates bigger XML files than srcML's—five times larger on average than the original source code [40]—because BFCA does not stop the creation of XML nodes at the expression level. This approach is good to obtain smaller XML file sizes, but makes more difficult the XPath searches and analysis that we implement in BFCA. On the other hand, BFCA XML file sizes are smaller than those obtained with ACML representation [28], with XML files more than a hundred times larger than the original source code.

10.2 Profilazer and Loopest performance

Once XMLCetus has generated the XML-based representations of source code, Profilazer and Loopest are executed. Sizes of XML files created by Profilazer, after adding

profiling information to the XMLCetus' source code representation, are quite similar to XML files created by XMLCetus, because Profilazer only adds a few attributes in *For Loop* nodes. It hardly means a few bytes to the total size, and thus, these sizes are not shown in this study.

Table 6 shows Profilazer and Loopest execution times. As it occurs with XMLCetus execution, in general terms, the larger the source code, the longer the time needed by Profilazer and Loopest. However, an important factor that affects executions of both applications is the number of *FOR* loops that benchmarks contain. Since *FOR* loops are the focus of the analysis performed by Loopest, and the focus of the Profilazer operation, it is clear that the performance of both applications will also depend on the number of these loops.

In the case of Loopest, its performance is not only affected by the number of *FOR* loops, but also by their complexity. Since Loopest should report how statements, functions and variables inside all loops are related with the rest of the source code, the more complex *FOR* loops are, the longer Loopest takes to run.

11 Conclusions

This paper addresses the problem of automatic characterization and coverage of sequential loops. We handle this problem with the development of the BFCA, a flexible and robust framework that provides complete and helpful reports that characterize the loops in a code. This information, including loop coverage and variable usage, allows fast prototyping of new solutions regarding code analysis and/or guiding the parallelization of the application, using either shared-memory programming models such as OpenMP, or speculative parallelization techniques. Our framework, which is based on Cetus, takes an advantage of the XML representation and its associated analysis and transformation tools. The resulting system extends Cetus capabilities in a much more flexible way: as an example, the use of our system leads to an 83 % reduction on the number of code lines needed to perform private variable analysis.

Our current and future work include the use of this framework with three purposes. First, to discover parallelization niches in widely used benchmarks that may benefit from software-based speculative parallelization. Second, to use this information to decide what limitations of current parallelization schemes ought to be faced first. Third, to instrument the augmented XML code with runtime parallelization function calls, to automatically transform the code into a speculative, parallel version of the code, and letting Sirius translate the modified XML tree into C language, finishing the process. We hope that the use of this framework will help runtime parallelization technology to be mature enough for its inclusion in mainstream compilers.

The BFCA framework is available under request. Please, contact the authors for more information.

Acknowledgments This research is partly supported by the Castilla-Leon Regional Government (VA172A12-2); Ministerio de Industria, Spain (CENIT OCEANLIDER); MICINN (Spain) and the European Union FEDER (MOGECOPP project TIN2011-25639, CAPAP-H3 network TIN2010-12011-E, CAPAP-H4 network TIN2011-15734-E). Sergio Aldea is supported by a research grant (EDU/1204/2010) of Consejería de Educación, Junta de Castilla y León, Spain, and the European Social Fund.

References

1. Al-Ekram R, Kontogiannis K (2005) An XML-Based framework for language neutral program representation and generic analysis. In: Proceedings of CSMR'05, pp 42–51
2. Aldea, S, Llanos DR, Gonzalez-Escribano A (2011) Towards a compiler framework for thread-level speculation. In: Proceeding of PDP'11, pp 267–271
3. Aldea S, Llanos DR, Gonzalez-Escribano A (2012) Using SPEC CPU2006 to evaluate the sequential and parallel code generated by commercial and open-source compilers. J Supercomput 2012(59):486–498
4. Atsumi N, Kobayashi T, Yamamoto S, Agusa K (2011) An XML C source code interchange format for CASE tools. In: Proceedings of COMPSAC'11, pp 498–503
5. Badros GJ (2000) JavaML: a markup language for java source code. Comput Netw 33(1):159–177
6. Berglund A, Boag S, Chamberlin D, Fernandez MF, Kay M, Robie J, Simon J (2010) XML Path language (XPath) 2.0 (Second edition). W3C recommendation 14 Dec 2010. http://www.w3.org/TR/xpath20/. Accessed December 2013
7. Boshernitsan M, Graham SL (2000) Designing an XML-Based exchange format for Harmonia. In: Proceedings of WCRE'00, pp 287–289
8. Bray T, Paoli J, Sperberg-McQueen CM, Maler E, Yergeau F (2008) Extensible Markup Language (XML). W3C Recommendation 26 November 2008. http://www.w3.org/TR/xml/. Accessed December 2013
9. Campanoni S, Jones TM, Holloway G, Wei GY, Brooks D (2012) Helix: making the extraction of thread-level parallelism mainstream. IEEE Micro 32(4):8–18
10. Chamberlain BL, Callahan D, Zima HP (2007) Parallel programmability and the chapel language. Int J HPC Appl 21(3):291–312
11. Chandra R, Menon R, Dagum L, Kohr D, Maydan D, McDonald, J (2000) Parallel Programming in OpenMP, 1 edn. Morgan Kaufmann, Burlington
12. Chen MK, Olukotun K (2003) The Jrpm system for dynamically parallelizing java programs. In: Proceedings of ISCA'03, pp 434–445
13. Cintra M, Llanos D (2005) Design space exploration of a software speculative parallelization scheme. IEEE Trans Parallel Distrib Syst 16(6):562–576
14. Collard ML (2003) An infrastructure to support meta-differencing and refactoring of source code. In: Proceedings of ASE'03, pp 377–380
15. Collard ML (2005) Addressing source code using srcML. In: IWCP'05 working session textual views of source code to support comprehension, pp 1–3
16. Collard ML, Decker MJ, Maletic JI (2011) Lightweight transformation and fact extraction with the srcML toolkit. In: Proceedings of SCAM'11, pp 173–184
17. Collard ML, Kagdi HH, Maletic, JI (2003) An XML-Based lightweight C++ fact extractor. In: Proceedings of IWPC'03, pp 134–143
18. Collard ML, Maletic JI (2004) Document-oriented source code transformation using XML. In: Proceedings of SET'04, pp 11–14
19. Collard ML, Maletic JI, Robinson BP (2010) A lightweight transformational approach to support large scale adaptive changes. In: Proceedings of ICSM'10, pp 1–10
20. Cordy JR (2003) Generalized selective XML markup of source code using agile parsing. In: Proceedings of IWPC'03, pp 144–153
21. Dave C, Bae H, Min S, Lee S, Eigenmann R, Midkiff S (2009) Cetus: A Source-to-Source compiler infrastructure for multicores. Computer 42(12):36–42
22. David G, Badros G, Aguiar A (2004) JavaML 2.0: Enriching the markup language for java source code. XML Aplicaes e Tecnologias Associadas (XATA'04)
23. Dou J, Cintra M (2007) A compiler cost model for speculative parallelization. ACM Trans Architect Code Optim 4(2)
24. Du Z, Lim C, Li X, Yang C, Zhao Q, Ngai T (2004) A cost-driven compilation framework for speculative parallelization of sequential programs. In: Proceedings of PLDI'04, pp 71–81
25. Ebert J, Kullbach B, Winter A (1999) GraX an interchange format for reengineering tools. In: Proceedings of WCRE'99, pp 89–98
26. Fischer G, Lusiardi J (2008) JaML—an XML Representation of Java Source code. University of Wrzburg, Tech. Rep

27. Garcia S, Jeon D, Louie CM, Taylor MB (2012) The kremlin oracle for sequential code parallelization. IEEE Micro 32(4):42–53
28. Gondow K, Kawashima H (2002) Towards ANSI C program slicing using XML. Electron Notes Theor Comput Sci 65(3):30–49
29. Henning JL (2006) SPEC CPU2006 benchmark descriptions. SIGARCH Comput Archit News 34(4):1–17
30. Hertzberg B, Olukotun K (2011) Runtime automatic speculative parallelization. In: Proceedings of CGO'11, pp 64–73
31. Holt RC, Winter A, Schrr A (2000) GXL: toward a standard exchange format. In: Proceedings of WCRE'00, pp 162–171
32. Iverson W (2005) Apache Jakarta Commons: reusable Java(TM) components. Prentice Hall PTR, Upper Saddle River, pp 154–156
33. Johnson TA, Eigenmann R, Vijaykumar TN (2007) Speculative thread decomposition through empirical optimization. In: Proceedings of PPoPP'07, pp 205–214
34. Kay M (2013) XSL Transformations (XSLT) version 2.0. W3C recommendation 23 january 2007. http://www.w3.org/TR/xslt20/. Accessed December 2013
35. Kejariwal A, Tian X, Girkar M, Li W, Kozhukhov S, Banerjee U, Nicolau A, Veidenbaum AV, Polychronopoulos CD (2007) Tight analysis of the performance potential of thread speculation using SPEC CPU 2006. In: Proceedings of PPoPP'07, pp 215–225
36. Kim M, Kim H, Luk C (2010) SD3: a scalable approach to dynamic data-dependence profiling. In: Proceedings of MICRO'43, pp 535–546
37. Liu W, Tuck J, Ceze L, Ahn W, Strauss K, Renau J, Torrellas J (2006) POSH: a TLS compiler that exploits program structure. In: Proceedings of PPoPP'06, pp 158–167
38. Luo Y, Packirisamy V, Hsu W, Zhai A, Mungre N, Tarkas A (2009) Dynamic performance tuning for speculative threads. In: Proceedings of ISPA'09, pp 462–473
39. Maeda K (2007) Experience of XML-based source code representation with parsing actions. In: Proceeding of SoMeT'07, pp 330–339
40. Maletic JI, Collard ML, Kagdi H (2004) Leveraging XML technologies in developing program analysis tools. In: Proceedings of ACSE'04, pp 80–85
41. Mamas E, Kontogiannis K (2000) Towards portable source code representations using XML. In: Proceedings of WCRE'00, pp 172–182
42. Maruyama K, Yamamoto S (2004) A CASE tool platform using an XML representation of java source code. In: Proceedings of SCAM'04, pp 158–167
43. McArthur G, Mylopoulos J, Ng SKK (2002) An extensible tool for source code representation using XML. In: Proceedings of WCRE'02, pp 199–208
44. Mendonca NC, Maia PHM, Fonseca LA, Andrade RMC (2004) RefaX: a refactoring framework based on XML. In: Proceedings of ICSM'04, pp 147–156
45. Moseley T, Connors DA, Grunwald D, Peri R (2007) Identifying potential parallelism via loop-centric profiling. In: Proceedings of CF'07, pp 143–152
46. Packirisamy V, Zhai A, Hsu W, Yew P, Ngai T (2009) Exploring speculative parallelism in SPEC2006. In: Proceedings of ISPASS'09, pp 77–88
47. Power JF, Malloy BA (2002) Program annotation in XML: a Parse-Tree based approach. In: Proceedings of WCRE'02, pp 190–198
48. von Praun C, Bordawekar R, Cascaval C (2008) Modeling optimistic concurrency using quantitative dependence analysis. In: Proceedings of PPoPP'08, pp 185–196
49. Putro HP, Liem I (2011) XML representations of program code. In: Proceedings of ICEEI'11, pp 1–6
50. Salinas-Mendoza A, Juarez-Martinez U, Alor-Hernandez G, Olivares-Zepahua BA (2011) Designing an XML-based representation for CaesarJ source code. In: Proceedings of CERMA'11, pp 427–432
51. Sun YX, Chen HY, Tse TH (2008) Lean implementations of software testing tools using XML representations of source codes. In: Proceedings of CSSE'08, pp 708–711
52. Tian C, Feng M, Gupta R (2010) Speculative parallelization using state separation and multiple value prediction. In: Proceedings of ISMM'10, pp 63–72
53. Tidwell D (2008) XSLT, 2nd 2 edn. O'Reilly Media, Sebastopol
54. Wagner C, Margaria T, Pagendarm HG (2009) Analysis and code model extraction for C/C++ source code. In: Proceedings of ICECCS'09, pp 110–119
55. Wang S, Dai X, Yellajyosula KS, Zhai A, Yew P (2006) Loop selection for thread-level speculation. In: Proceedings of LCPC'05, pp 289–303

56. Whaley J, Kozyrakis C (2005) Heuristics for profile-driven method-level speculative parallelization. In: Proceedings of ICPP'05, pp 147–156
57. Wheeler DA (2004) Sloccount: counting source lines of code. http://www.dwheeler.com/sloccount/. Accessed December 2013
58. Wu P, Kejariwal A, Cacaval C (2008) Compiler-driven dependence profiling to guide program parallelization. In: Amaral J (ed) Languages and compilers for parallel computing. Lecture notes in computer science, vol 5335, pp 232–248
59. Zou Y, Kontogiannis K (2001) A framework for migrating procedural code to Object-Oriented platforms. In: Proceedings of APSEC'01, APSEC '01, pp 390–399

Copyright of Journal of Supercomputing is the property of Springer Science & Business Media B.V. and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.