Proceedings of the 15th International Conference
on Computational and Mathematical Methods
in Science and Engineering, CMMSE 2015
3–7July, 2015.

Automatic synthesis of parallel code with TLS capabilities

Sergio Aldea[1], Diego R. Llanos[1] and Arturo Gonzalez-Escribano[1]

[1] *Departamento de Informática, Universidad de Valladolid*

emails: sergio@infor.uva.es, diego@infor.uva.es, arturo@infor.uva.es

Abstract

Parallelization of sequential applications requires extracting information about the loops and how their variables are accessed, and afterwards, augmenting the source code with extra code depending on such information. In this paper we propose a framework that avoids such an error-prone, time-consuming task. Our solution leverages the compile-time information extracted from the source code to classify all variables used inside each loop according with their accesses. Then, our system, called BFCA+, automatically instruments the source code with the necessary OpenMP directives and clauses to allow its parallel execution, using the standard *shared* and *private* clauses for variable classification. The framework is also capable of instrumenting loops for speculative parallelization, with the help of the ATLaS runtime system, that defines a new *speculative* clause to point out those variables that may lead to a dependency violation. As a result, the target loop is guaranteed to correctly run in parallel, ensuring that its execution follows sequential semantics even in the presence of dependence violations.

Key words: Automatic parallelization, code analysis, compiler framework, OpenMP, source synthesis, source transformation, speculative parallelization, XML.

1 Introduction

One of the main concerns of current computer science is the study of parallel capabilities for both programs and processors that execute them. Due to the huge number of sequential programs already written for many decades until now, complexity of parallel programming languages, and knowledge required to parallelize source code, a technique that automatically parallelize them is quite desirable. However, automatic parallelization techniques currently implemented in many commercial compilers are not able to parallelize most of the loops because of data dependencies. Therefore, searching for new techniques that obtain a profit

 ISBN: 978-84-617-2230-3

from potential parallel capabilities of current hardware and CMP (Chips MultiProcessor) architectures is still necessary.

Speculative Parallelization (SP), also called Thread-Level Speculation (TLS) [7, 8, 11] or Optimistic Parallelization [15], is a runtime technique to extract parallelism from fragments of code that cannot be analyzed at compile time. This technique aims to automatically extract loop- and task-level parallelism when a compile-time dependence analysis can not guarantee that a given sequential code is safely parallelizable. TLS optimistically assumes that the code can be executed in parallel, relying on a runtime monitor to ensure correctness. The original code is augmented with function calls that distribute iterations among processors, monitor the use of all variables that may lead to a dependency violation, and perform in-order commits to store the results obtained by successful iterations. If a dependency violation appears at runtime, these library functions stop the offending threads and restart them in order to use the updated values, thus preserving sequential semantics.

Current speculative techniques are experimental and require manual intervention of expert programmers. These programmers firstly need to extract certain information about the source code that they aim to parallelize. Without automatic tools, programmers have to manually extract that information, such as variable usages within each loop, or I/O functions that complicate, or even preclude, the parallelization. More important, they should determine whether it is worth parallelizing a loop or if the thread-management overheads would be larger than the benefit of parallelizing. This information extraction is the first step to speculatively parallelize a source code. The second step is to add all the functions and structures needed to handle the speculative execution. So far, this process should also be carried out manually.

This paper addresses both problems. Relying on our previous framework, called BFCA [4], that extracts loop-based profiling and dependence information from source codes, we propose BFCA+, a solution that takes advantage of this information to automatically add the OpenMP directives needed to run a chosen loop in parallel. BFCA+ not only handles parallelizable loops (that is, loops where iterations can be run in any order without breaking sequential semantics), but also parallelizes loops with potential dependences among iterations. To do so, BFCA+ relies on the ATLaS framework [10, **?**], a Thread-Level Speculation (TLS) compile and runtime system developed by our group, that extends OpenMP functionalities with a new *speculative* clause to handle those variables that may lead to a dependence violation. During parallel execution, variables that were labeled as *speculative* are monitored at runtime. If a dependence violation occur, the thread that has consumed an incorrect value is stopped and restarted, ensuring sequential semantics.

BFCA+ usage is simple. A first run obtains profile information of all loops in the application, together with a classification of variables usage in all of them. After examining these results, the user should choose a loop for parallel execution. A second run of BFCA+ with the line number of the chosen loop generates an OpenMP-based parallel version of the

 ISBN: 978-84-617-2230-3

loop, using the *shared*, *private*, and the non-standard *speculative* clauses to classify loop variables according with their usage. Thanks to this solution, *any* target loop is guaranteed to correctly run in parallel while preserving its sequential semantics.

The rest of the paper is organized as follows. Section 2 describes some related approaches. Section 3 describes the overall architecture of BFCA+ and its main building blocks, BFCA and ATLaS. Section 4 describes in detail the transformation process for a given loop. And finally, Sect. 5 summarizes our conclusions.

2 Related work

The generation of source code is a problem that concerns different areas, such as refactoring, optimization, and parallelization of source codes. In the case of BFCA+, there are many different proposals to the automatic parallelization of source codes, and more particularly, focused on the synthesis of OpenMP constructs.

One of the first attempts to automatize the generation of OpenMP constructs is the POST project [1], that provides a simple environment that also allows the intervention of the user. A more advanced system is ParaWise/CAPO [12, 13, 14], which uses a dependency analysis to create the appropriate OpenMP directives to parallelize simple and nested loops in Fortran applications. It also applies a certain level of optimization transformations to enhance the quality of the generated code. This is also the case of PLuTo [6], a source-to-source framework that uses the polyhedral model to optimize the code and generate OpenMP parallel code automatically.

The polyhedral model [5] is used by several proposals to enhance the code and obtain the information needed to create the OpenMP directives automatically. Graphite [19] is a branch of GCC that applies the polyhedral model to different purposes, including the generation of parallel code, and proposes an auto-parallelization option for GCC that uses OpenMP structures to define parallel sections.

Unlike most of the approaches, which are source-to-source parallelizers that automatically generate OpenMP directives and clauses (e.g. Liao *et al.* [16], Cetus [9], or several of the proposals seen above), Gaspard2 [18] follows a model-to-source approach. Gaspard2 is a graphical framework that needs that the code and the available parallelism be expressed by the user with an UML-based model, which is then transformed into an OpenMP model that generates the parallel version of the source code. Finally, YAO [17] is a graph-based framework, focused on the data assimilation mostly for geophysical applications, able to generate not only OpenMP constructs to parallelize code regions, but also *atomic* directives to avoid race conditions.

As we will see, like most of the approaches described above, BFCA+ follows a source-to-source approach, leveraging its XML-based representation of the source code to analyze and augment the code with OpenMP parallel constructs, including our *speculative* clause [2].

 ISBN: 978-84-617-2230-3

This differs from other approaches: BFCA+ is able to synthesize the code needed to handle the speculative execution of a certain program, creating an OpenMP-based parallel version from the sequential source code.

3 BFCA+ building blocks

Our proposal has been built upon two different solutions that, until now, worked independently of each other. The first solution is BFCA [4], 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 taxonomy of their variables based on their accesses, as well as other hurdles that restrict the parallelization. BFCA architecture is depicted inside a dashed-line box in Fig. 1. BFCA+ extends this framework to enable the synthesis of source code augmented with OpenMP directives and clauses, intended not only to automatically classify variables according with their usage (*shared* or *private*), but also to insert appropriate OpenMP directives to enable the automatic parallelization of the target loop (see the main solid box of Fig. 1).

In certain situations, however, the parallel execution of a given loop may fail due to the existence of runtime dependences among iterations. Note that, although this situation may actually occur or not in a given execution, depending on the control flow of the program, its mere possibility makes unsafe the loop parallelization. This is where speculative parallelization comes into scene. When BFCA+ detects such a situation, it is also able to label as *speculative* all variables whose definition or use may potentially lead to a dependence violation. Unlike *shared* and *private* clauses, the *speculative* clause is not part of the OpenMP standard, but a new clause first proposed by our group [3]. This proposal was later implemented in the ATLaS framework, the second solution BFCA+ is built upon.

With respect to the ATLaS compile and runtime framework [10, **?**], it is composed by two main elements. The first one is the compiler support, thanks to a specialized GCC compiler plugin that detects the use of the *speculative* clause, and augments the source code with software that controls the speculative execution of the loop. The second one is the runtime support, with functions that monitors at runtime the existence of dependence violations, and performs corrective actions if such a violation takes place.

These two building blocks, BFCA and ATLaS are used together in BFCA+ to give a complete solution of the automatic synthesis of parallel versions of loops. The user should first run the framework to see which loops can be parallelized, choose one, and let the automatic transformation system to generate an executable that is guaranteed to run correctly in parallel. Without BFCA+, programmers needed to manually classify variables of the loop that they aim to parallelize, and then, insert all the OpenMP constructs required by ATLaS to parallelize it speculatively. BFCA+ also solves this problem, freeing programmers from

 ISBN: 978-84-617-2230-3

this error-prone, tedious task.

Figure 1 shows the architecture of BFCA+, and how it generates an OpenMP-based source code that is compilable by ATLaS to generate a speculatively parallel version of the code. As can be seen in the figure, BFCA+ relies heavily on BFCA, composed in turn by three main subsystems: (1) XMLCetus, a modified version of Cetus [9] that builds an XML tree representing the original C source code; (2) Profilazer, that executes the code and augments the XML tree with profiling information; and (3) Loopest, that exploits XML capabilities to characterize every loop in the source code, and performs a taxonomy of the variables based on how they are accessed.

BFCA+ adds to this solution a new module, called OMPlizer, that synthesizes OpenMP-based constructs to speculatively parallelize the code. Finally, a fifth module, called Sirius, transforms the XML representation back to C.

Figure 2 summarizes the process that transforms a sequential source code into a parallel one. In a first execution, BFCA+ reports about each loop and how its variables are accessed. Then, the programmer only needs to point out the line number of the loop to be parallelized. In a second execution, BFCA+ uses the information on the variable accesses to automatically augment the XML representation of the code by using OMPlizer. OMPlizer modifies the XML node that represents the *FOR* loop, and inserts a new XML node with the OpenMP parallel directive and the corresponding clauses, according to the variable classification that Loopest creates. This includes the insertion of the *speculative* clause with those variables that may lead to a dependency violation.

Once OMPlizer has augmented the XML representation of the source code, Sirius transforms it back into a C representation. During this process, the original sequential code has been annotated with OpenMP constructs that parallelize it. Figure 7 depicts the result of this transformation process. Finally, ATLaS processes these OpenMP annotations, and performs all the changes needed in the loop to be run in parallel using our TLS runtime library, including the replacement of the accesses over speculative variables with the corresponding speculative versions of these accesses. It is important to remark that, if the *FOR* loop being parallelized does not contain speculative variables, BFCA+ generates the OpenMP constructs in order to be parallelized according to the OpenMP standard.

4 BFCA+ transformation process

As it has been described above, the process of augmenting a parallel *FOR* loop using BFCA+ is split in two steps. Figure 4 shows how the programmer should run BFCA+ to parallelize the loop in line 5 of the synthetic benchmark shown in Fig. 3. First, the programmer should run BFCA+ with the source code, to obtain a report with the characterization of each loop. During its execution, BFCA+ generates an XML representation of the source code, shown in Fig. 5. BFCA+ processes this XML tree to obtain the information that is shown to the

 ISBN: 978-84-617-2230-3

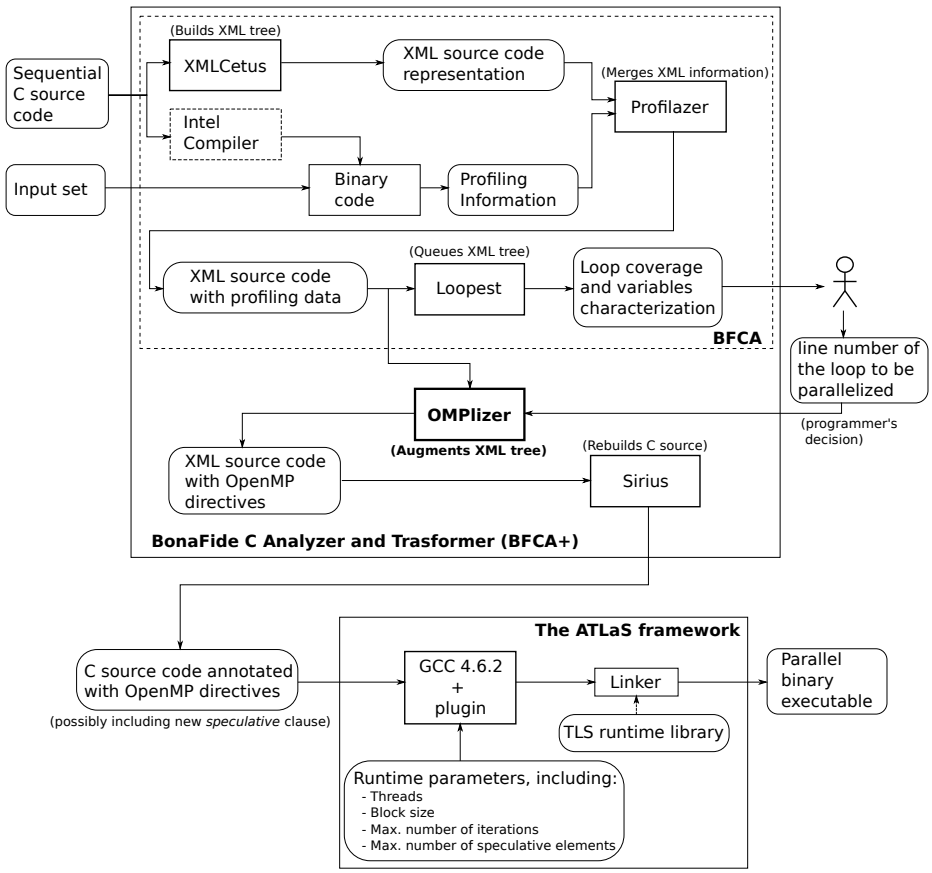

Figure 1: Overview of the BFCA+ plus ATLaS architecture, that analyzes the code, generates an OpenMP-based speculatively-parallel version of the code, and finally compiles it to create an executable that runs in parallel speculatively.

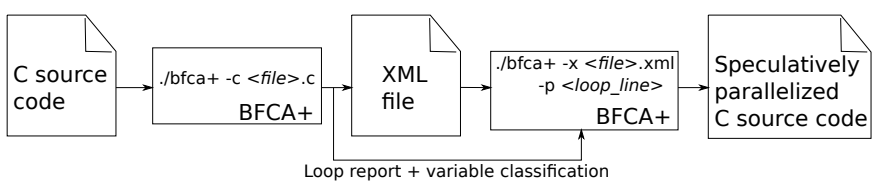

Figure 2: Overview of the process that transforms a sequential C code into a parallel one. BFCA+ automatically generates the OpenMP constructs, including the *speculative* clause.

 ISBN: 978-84-617-2230-3

```
1  #define NITER 1000000, MAX 100
2  int array[MAX];
3  ...
4
5  for ( P = 0 ; P < NITER ; P++ )
6    Q = P % (MAX) + 1;
7    aux = array[Q - 1];
8    Q = (4 * aux) % (MAX) + 1;
9    array[Q - 1] = aux;
10 }
```

Figure 3: Synthetic benchmark with a data structure (`array`) whose accesses may lead to a dependency violation.

```
$ ls
synthetic.c
$ bfca+ -c synthetic.c
...
synthetic.c:6:  Line number:  6
synthetic.c:6:  Number of lines:  5
synthetic.c:6:  Inclusive Time Percent:  34.2
synthetic.c:6:  Exclusive Time Percent:  34.2
synthetic.c:6:  It doesn't contain pointer arithmetic.
synthetic.c:6:  Number of Loop Control Variables:  1
synthetic.c:6:  Loop Control Variable:  (int) P.
synthetic.c:6:  Variables read and written:  (int) Q,
                (int) aux, (int) array[100].
synthetic.c:6:  Variables only read:  (int) P.
synthetic.c:6:  Private Variables:  (int) Q, (int) aux,
                (int) P.
synthetic.c:6:  Speculative Variables:  (int) array[100].
synthetic.c:6:  It is a well-formed FOR Loop.
...
$ bfca+ -x synthetic.profiled.xml -p 6
$ ls
synthetic.c   synthetic.profiled.spec.c
synthetic.profiled.xml   synthetic.profiled.spec.xml
$ ./atlas --threads 4 --block 100 --maxpointer 100 --maxiter 1000000
    -e synthetic.exe -c synthetic.profiled.spec.c
...
$ ls
synthetic.c   synthetic.profiled.spec.c   synthetic.exe
synthetic.profiled.xml   synthetic.profiled.spec.xml
```

Figure 4: Example of BFCA+'s run to parallelize the loop in line 5 of the *synthetic* benchmark shown in Fig. 3. It is also shown how ATLaS compiles the source code and generates the executable.

 ISBN: 978-84-617-2230-3

```
<ForLoop condition="P&lt;1000000" initial="P=0;" length="7" lineNumber="6"
         step="P ++" entryCount="1" absTime="3161466" absTimePercent="41.6"
         selfTime="3161466" selfTimePercent="41.6">
  <Statement lineNumber="-1">
    <ExpressionStatement lineNumber="-1">
      <Expression>
        <BinaryExpression lhs="P" operator="=" rhs="0">
          <AssignmentExpression>
            <Expression>
              <IDExpression>
                <Identifier array="" name="P" opUnary="" type="int"/>
              </IDExpression>
            </Expression>
            <Expression>
              <Literal>
                <IntegerLiteral value="0"/>
              </Literal>
            </Expression>
          </AssignmentExpression>
        </BinaryExpression>
      </Expression>
    </ExpressionStatement>
  </Statement>
  <Expression>
    <BinaryExpression lhs="P" operator="&lt;" rhs="1000000">
      <Expression>
        <IDExpression>
          <Identifier array="" name="P" opUnary="" type="int"/>
        </IDExpression>
      </Expression>
      <Expression>
        <Literal>
          <IntegerLiteral value="1000000"/>
        </Literal>
      </Expression>
    </BinaryExpression>
  </Expression>
  <Expression>
    <UnaryExpression expression="P" operator="post ++">
      <Expression>
        <IDExpression>
          <Identifier array="" name="P" opUnary="" type="int"/>
        </IDExpression>
      </Expression>
    </UnaryExpression>
  </Expression>
  <Statement lineNumber="6">
    <CompoundStatement>
      <Statement lineNumber="7"> ... </Statement>
      <Statement lineNumber="8"> ... </Statement/>
      <Statement lineNumber="9"> ... </Statement/>
      <Statement lineNumber="10"> ... </Statement/>
    </CompoundStatement>
  </Statement>
</ForLoop>
```

Figure 5: XML representation created by BFCA for the loop in line 5 of Fig. 4 before being parallelized.

 ISBN: 978-84-617-2230-3

```
<ForLoop annotation="OpenMP" condition="P&lt;1000000" initial="P=0;" length="5"
         lineNumber="6" step="P ++" entryCount="1"
         absTime="3161466" absTimePercent="41.6" selfTime="3161466" selfTimePercent="
             41.6">

  <Annotation annotation="#pragma omp parallel for default(none) schedule(static) \
                          speculative(array) \
                          private(Q, aux, P)"/>

  <Statement lineNumber="-1">
    <ExpressionStatement lineNumber="-1">
      <Expression>
        <BinaryExpression lhs="P" operator="=" rhs="0">
          <AssignmentExpression>
            <Expression>
              <IDExpression>
                <Identifier array="" name="P" opUnary="" type="int"/>
              </IDExpression>
            </Expression>
            ...
```

Figure 6: XML representation of Fig. 5 augmented by OMPlizer (highlighted code).

```
#define NITER 1000000, MAX 100
int array[MAX];
...
#pragma omp parallel default(none) \
 private(P, Q, aux) speculative(array)
for ( P = 0 ; P < NITER ; P++ )
  Q = P % (MAX) + 1;
  aux = array[Q - 1];
  Q = (4 * aux) % (MAX) + 1;
  array[Q - 1] = aux;
}
```

Figure 7: Loop of Fig. 3 parallelized after being processed by BFCA+.

 ISBN: 978-84-617-2230-3

programmer in order to choose which loop BFCA+ should be parallelized.

Then, the programmer runs BFCA+ again with the XML file that represents the source code (Fig. 5) and the line number of the loop to be parallelized. With these data, BFCA+ processes the XML tree to augment the target loop (Fig. 6). This loop is annotated to be later transformed, whereas a new XML node is also inserted to represent the OpenMP parallel directive. Taking the classification obtained in the previous execution of BFCA+ either only OpenMP standard clauses are added, or our proposed *speculative* clause is inserted to trigger the subsequent transformation of the loop to handle a speculatively parallel execution.

After these two runs, BFCA+ generates a source code (Fig. 7) in which the parallel loop has been annotated with the OpenMP constructs stated in the XML tree. Once the source code has been properly annotated, it can be compiled using ATLaS, creating an executable that is able to run in parallel speculatively. During this compilation, ATLaS generates new code that is necessary to handle the speculative execution.

To sum up, Figs. 3–7 show the transformation process performed by BFCA+, from a sequential source code to a parallel code with TLS capabilities. Without BFCA+, all these transformations and generated code would be manually performed by the programmers, potentially prone to errors.

5 Conclusions

This paper addresses the problem of the automatic synthesis and generation of OpenMP constructs. We propose a system, called BFCA+, that achieves this purpose by firstly performing a characterization of the loops of a source code, and classifying their variables according to their accesses. Our solution combines this information with the programmer's choice on which loop to parallelize in order to automatically instrument the loop to make it parallelizable by either the OpenMP standard or speculatively. This automatic instrumentation frees the programmer from a manual edition of the source code that requires some technical expertise and likely leads to a tedious, error-prone task.

Although BFCA+ can be easily used with other approaches, it is specially focused on TLS. BFCA+ generates a *speculative* clause [2] for those variables that could lead to any dependency violation. The classification and generation of OpenMP clauses performed by BFCA+ is essential for ATLaS the compile and runtime system that uses this new clause to compile and execute a certain loop speculatively in parallel. This loop will be guaranteed to run correctly in parallel, ensuring that its execution follows a sequential semantics. Moreover, the experimental results show that the automatic transformation leads to a faster code than the one obtained by manually replacing accesses to speculative variables with function calls.

Our solution simplifies the whole process of parallelizing a code speculatively, reducing

 ISBN: 978-84-617-2230-3

the learning curve needed to apply this parallelization technique, and relying on BFCA+ both classification of the variables, and the augmentation of the loop being parallelized.

Acknowledgments

This research is partly supported by the Castilla-Leon Regional Government (VA172A12-2), MICINN (Spain) and the European Union FEDER (MOGECOPP project TIN2011-25639, HomProg-HetSys project, TIN2014-58876-P, CAPAP-H5 network TIN2014-53522-REDT).

References

[1] Laksono Adhianto, Francois Bodin, B. Chapman, Laurent Hascoet, Aron Kneer, David Lancaster, I. Wolton, and M. Wirtz. Tools for OpenMP application development: The POST project. *Concurrency - Practice and Experience*, 12:1177–1191, 2000.

[2] Sergio Aldea, Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. A new GCC plugin-based compiler pass to add support for thread-level speculation into OpenMP. In *Euro-Par'14 Proceedings*, 2014.

[3] Sergio Aldea, Diego R. Llanos, and Arturo Gonzalez-Escribano. Support for thread-level speculation into OpenMP. In *IWOMP'12 Proceedings*, pages 275–278, June 2012.

[4] Sergio Aldea, Diego R. Llanos, and Arturo Gonzalez-Escribano. The BonaFide C analyzer: Automatic loop-level characterization and coverage measurement. *The Journal of Supercomputing*, 68(3):1378–1401, 2014.

[5] Cdric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'04 Proceedings*, pages 7–16, 2004.

[6] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI'08 Proceedings*, pages 101–113, 2008.

[7] Marcelo Cintra and Diego R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *PPoPP'03 Proceedings*, pages 13–24, June 2003.

[8] Francis H. Dang, Hao Yu, and Lawrence Rauchwerger. The R-LRPD Test: Speculative parallelization of partially parallel loops. In *IPDPS'02 Proceedings*, pages 20–29, 2002.

[9] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A Source-to-Source compiler infrastructure for multicores. *IEEE Computer*, 42(12):36–42, 2009.

 ISBN: 978-84-617-2230-3

[10] Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. New data structures to handle speculative parallelization at runtime. In *HLPP'14 Proceedings*, 2014. To appear.

[11] Manish Gupta and Rahul Nim. Techniques for speculative run-time parallelization of loops. In *SC'98 Proceedings*, pages 1–12, 1998.

[12] Cos S. Ierotheou, Haoqiang Jin, Gregory Matthews, Stephen P. Johnson, and Robert Hood. Generating OpenMP code using an interactive parallelization environment. *Parallel Computing*, 31(10–12):999–1012, October 2005.

[13] Haoqiang Jin, Gabriele Jost, Jerry Yan, Eduard Ayguade, Marc Gonzalez, and Xavier Martorell. Automatic multilevel parallelization using OpenMP. *Journal of Scientific Programming, EWOMP'11*, 11(2)(2):177–190, April 2003.

[14] Stephen Johnson, Emyr Evans, Haoqiang Jin, and Constantinos Ierotheou. The Para-Wise expert assistant - Widening accessibility to efficient and scalable tool generated OpenMP code. In *WOMPAT'04 Proceedings*, pages 67–82, 2005.

[15] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI'07 Proceedings*, pages 211–222, 2007.

[16] Chunhua Liao, Daniel J. Quinlan, Jeremiah J. Willcock, and Thomas Panas. Automatic parallelization using OpenMP based on STL semantics. *Languages and Compilers for Parallel Computing (LCPC)*, 2008.

[17] Luigi Nardi, Fouad Badran, Pierre Fortin, and Sylvie Thiria. YAO: A generator of parallel code for variational data assimilation applications. In *HPCC'12 Proceedings*, pages 224–232, June 2012.

[18] Julien Taillard, Frdric Guyomarc'h, and Jean-Luc Dekeyser. A graphical framework for high performance computing using an MDE approach. In *PDP'08 Proceedings*, pages 165–173, February 2008.

[19] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjdin, and Ramakrishna Upadrasta. Graphite two years after: First lessons learned from real-world polyhedral compilation. In *GROW'10 Proceedings*, pages 4–19, 2010.

 ISBN: 978-84-617-2230-3