J Supercomput (2017) 73:88–99
DOI 10.1007/s11227-016-1623-0

BFCA+: automatic synthesis of parallel code with TLS capabilities

Sergio Aldea[1] · **Diego R. Llanos**[1] · **Arturo Gonzalez-Escribano**[1]

Published online: 21 January 2016
© Springer Science+Business Media New York 2016

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 to 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 dependency violations. Our experimental evaluation shows that the framework not only saves development time, but also leads to a faster code than the one manually parallelized.

Keywords Automatic parallelization · Compiler framework · OpenMP · Source synthesis · Source transformation · Speculative parallelization · XML

✉ Diego R. Llanos
diego@infor.uva.es

Sergio Aldea
sergio@infor.uva.es

Arturo Gonzalez-Escribano
arturo@infor.uva.es

[1] ETS Ingeniera Informtica, Universidad de Valladolid, Paseo Beln 15, 47011 Valladolid, Spain

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 parallelizes 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 the possibility of data dependencies [1].

Thread-Level Speculation (TLS) [2,3] is a runtime technique that can be used to run loops in parallel that may present dependency violations. TLS optimistically assumes that the code can be executed in parallel, relying on a runtime monitor to ensure correctness. 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.

Until now, speculative techniques were experimental, requiring the manual intervention of expert programmers. These programmers firstly needed to extract certain information about the source code that they aim to parallelize, such as variable usages within each loop, or I/O functions that complicate, or even preclude, the parallelization. The second step was to manually add all the functions and structures needed to handle the speculative execution.

Our research group has been working several years in the problem of automatic synthesis of parallel code with TLS capabilities. Our prior work included the development of both an analysis framework, called BFCA, and a TLS runtime system, called ATLaS.

- BFCA [4,5] is an analysis framework that extracts profiling and dependency information of `for` loops in C code, in order to detect which *for* loops are the best candidates for parallelization.
- ATLaS is a TLS compile [6] and runtime system [7] that extends OpenMP functionalities with a new *speculative* clause, to allow the parallelization of `for` loops that are not guaranteed at compile time to be dependency free.[1] During parallel execution, variables that were labeled as *speculative* are monitored at runtime. If a dependency violation occur, the thread that has consumed an incorrect value is stopped and restarted, ensuring sequential semantics.

The use of both frameworks greatly simplified the development of parallel code, but the intervention of an expert programmer was still needed to transform the sequential code into a parallel version.

The framework presented in this paper closes the loop, allowing for the first time the automatic synthesis of parallel code with TLS capabilities. BFCA+ relies on the data returned by BFCA to automatically add the directives needed to run a chosen loop in parallel, either using the classic OpenMP directives if the loop does not present

[1] Only well-formed `for` loops where the number of iterations are known at the beginning of the loop can be parallelized by the ATLaS framework. See [7] for additional details.

dependencies among iterations, or using the new OpenMP *speculative* clause provided by ATLaS if the lack of dependencies cannot be guaranteed at compile time. The BFCA+ framework can also be modified to support other state-of-the-art software-based speculative frameworks, such as SpLIP [8] or MiniTLS [9].

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

Regarding parallel performance, our experimental results, obtained by the execution of four different benchmarks that present dependency violations at runtime, show both a noticeable speedup (up to 13.5× speedup with 56 cores in the presence of dependency violations) and good scalability. Interestingly, the automatic transformation of the sequential code leads to better performance than those obtained when the user manually transforms the code for the same purpose using ATLaS runtime functions directly.

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 [10] that provides a simple environment that also allows the intervention of the user. A more advanced system is ParaWise/CAPO [11–13], 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 [14], a source-to-source framework that uses the polyhedral model to optimize the code and generate OpenMP parallel code automatically.

Graphite [15] 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. This work inspired Polly [16], a similar proposal but focused on a different compiler: LLVM [17]. Other works that also use the polyhedral model such as Par4All [18] and PYPS [19] are based on PIPS [20], a source-to-source framework. Par4All uses the analysis performed by PIPS to optimize and create OpenMP (among other standards) source codes. PYPS is a Python-based programmable pass manager, that also generates OpenMP constructs.

Unlike most of the approaches, which are source-to-source parallelizers that automatically generate OpenMP directives and clauses (e.g. Liao et al. [21], Cetus [22], or several of the proposals seen above), Gaspard2 [23] 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 [24] 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 [6]. 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+ architecture and usage

As we stated in Sect. 1, our proposal has been built upon two different solutions that, until now, worked independently of each other. The first solution is BFCA [5], 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 to their usage (*shared* or *private*), but also to insert appropriate OpenMP directives to enable the automatic parallelization of the target loop (see the solid box in Fig. 1).

In certain situations, however, the parallel execution of a given loop may fail due to the possibility of runtime dependencies among iterations. 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 dependency 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 [4]. 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 [7], it is composed of 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 dependency 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 `for` loops. The user should first run the framework to see which loops can be parallelized, choose one loop, and let the automatic transformation system to generate an executable that

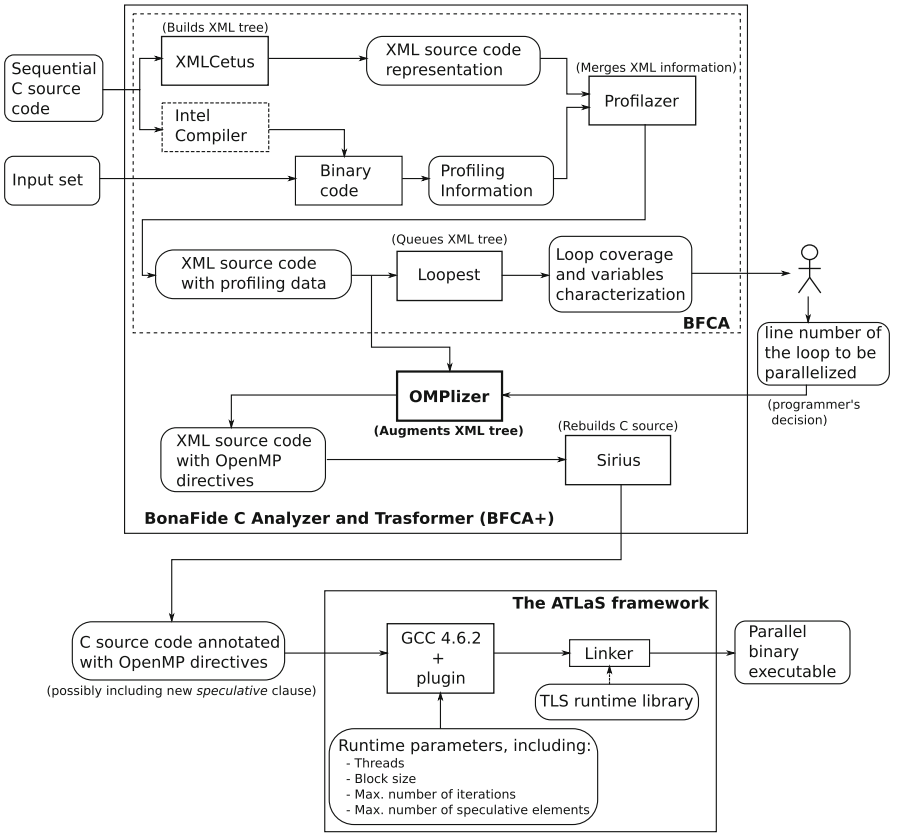

Fig. 1 Overview of the BFCA+ plus ATLaS architecture, that analyzes the code, generates an OpenMP-based speculatively parallel version of the code, and finally compiles it to create an executable that runs in parallel speculatively

is guaranteed to run correctly in parallel.[2] 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 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 [22] 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) Loopestthat exploits XML capabilities to characterize every loop in the source code

[2] The current version of BFCA+ only transforms a single loop of the application to avoid the transformation of two nested loops, a situation not allowed by the ATLaS runtime system. We expect to overcome this limitation in the near future.

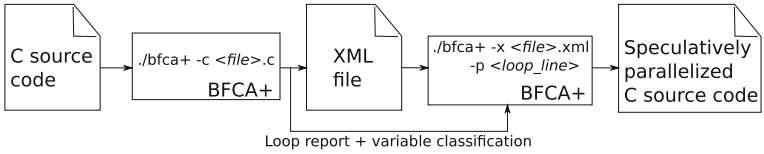

Fig. 2 Overview of the process that transforms a sequential C code into a parallel one. BFCA+ automatically generates the OpenMP constructs, including the *speculative* clause

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. 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.

Despite the complexity of the transformation process, the usage of the BFCA+ framework is simple. As an example, consider the sequential code depicted in Fig. 3. A first invocation to BFCA+ (shown in Fig. 4) analyzes the sequential code statically and dynamically, producing both a report with the characterization of the loop, and an XML file that merges this static characterization with the profiling information of the loop coverage (Fig. 5).

A second run of BFCA+, also shown in Fig. 4, allows the programmer to indicate the starting line of the loop that should be parallelized. This second invocation generates an augmented version of the XML file, called `program.spec.xml`, that includes the directives needed to parallelize the loop (Fig. 6). In addition, this second invocation transforms the augmented XML representation back to C. The result, shown in Fig. 7, is a C file containing a parallel, speculative version of a loop that cannot be parallelized by parallelizing compilers due to the possibility of dependency violations.

Finally, ATLaS is responsible of transform the parallel version of the code that contains the non-standard *speculative* clause into standard OpenMP directives plus

Fig. 3 File `program.c`: Fragment of C code that depicts a loop where the use of a data structure (`array`) may lead to a dependency violation

```
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 }
```

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

Fig. 4 Usage of BFCA+ in the parallelization of the loop in line 5 of the `program.c` file depicted in Fig. 3. The invocation to the ATLaS compiler framework is also shown

additional code. The intermediate result (before being effectively compiled) is shown in Fig. 8.

4 Evaluation

To evaluate the capabilities of BFCA+, we have used it to generate the OpenMP constructs (including our proposed *speculative* clause) for some synthetic and real-world benchmarks. The experiments have been designed to verify that the use of BFCA+ leads to executable files that are functionally equivalent to those manually parallelized by an experienced programmer.

All the benchmarks used in the experiments are not parallelizable at compile time due to several data dependencies, requiring runtime speculative parallelization. We

```
<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>
          ...
```

Fig. 5 File `program.xml`: fragment of the XML representation created by BFCA for the loop in line 5 of Fig. 4 before being parallelized

```
<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>
          ...
```

Fig. 6 File `program.spec.xml`: XML representation of Fig. 5 augmented by OMPlizer (*highlighted code*)

Fig. 7 File `program.spec.c`: C representation of `program.spec.xml` as returned by the Sirius module of BFCA+

```
#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;
}
```

```
#define NITER 1000000, MAX 100
int array[MAX];
...
specbegin(NITER)
#pragma omp parallel default(none) \
 private(P, Q, aux, ini, current, tid, retflag, value)
 shared(array, wheel_ns, wheel_ms, wheel, upper_limit, varblock)
{
#pragma omp for schedule(static) nowait
for (tid = 0; tid <= 3; tid = tid + 1) {
 ini = 0;
 current = tid;
 P = varblock[0][current]+ini;
 if (i > upper_limit - 1)
   goto labelSquash_1;
 varblock[2][current] = varblock[2][current]+1;
labelStartIteration_1:
   Q = P % (MAX) + 1;
   //aux = array[Q — 1];
   if (specload_pointer((unsigned char *) &(array[Q-1]),
       sizeof (array[Q-1]), current, (unsigned char *) &value) == -1)
     earlySquash(1);
   aux=value;
   Q = (4 * aux) % (MAX) + 1;
   //array[Q — 1] = aux;
   specstore_pointer((unsigned char *) &(array[Q-1]),
       sizeof (array[Q-1]),current,(unsigned char *) &value);
labelEndIteration_1:
 if ((P != varblock[1][current] + ini) && (i < NITER - 1)) {
   P = P + 1;
   goto labelStartIteration_1;
 }
labelSquash_1:
 retflag = threadend_pointer(&current);
 if (retflag == JOBDONE) goto labelEndLoop_1;
 if (retflag == JOBTODO) {
   i = varblock[0][current] + ini;
   varblock[2][current] = varblock[2][current] + 1;
   goto labelStartIteration_1;
 }
labelEndLoop_1:
 ;
 } // for loop
} // pragma omp parallel
```

Fig. 8 Code generated by ATLaS after processing the source code of the synthetic benchmark augmented by BFCA+ (shown in Fig. 7)

have used several real-world applications whose main loops are not parallelizable at compile time because of the possibility of dependence violations, and therefore require TLS to run in parallel. They are the two-dimensional Convex Hull problem (2D-Hull) [25], the Delaunay Triangulation using the Jump-and-Walk strategy [26], the two-dimensional Minimum Enclosing Circle (2D-MEC) problem [27], and a C implementation of TREE [28]. The loops considered in the first three applications are the main loop of the codes, that consumes almost 100 % of the execution time. For TREE, we have parallelized the ACCEL loop that consumes 95.17 % of the total execution time. 2D-Hull and 2D-MEC are executed with a ten-million-point dataset, Delaunay with a one-million-point dataset, and TREE with a dataset of 4096 nodes.

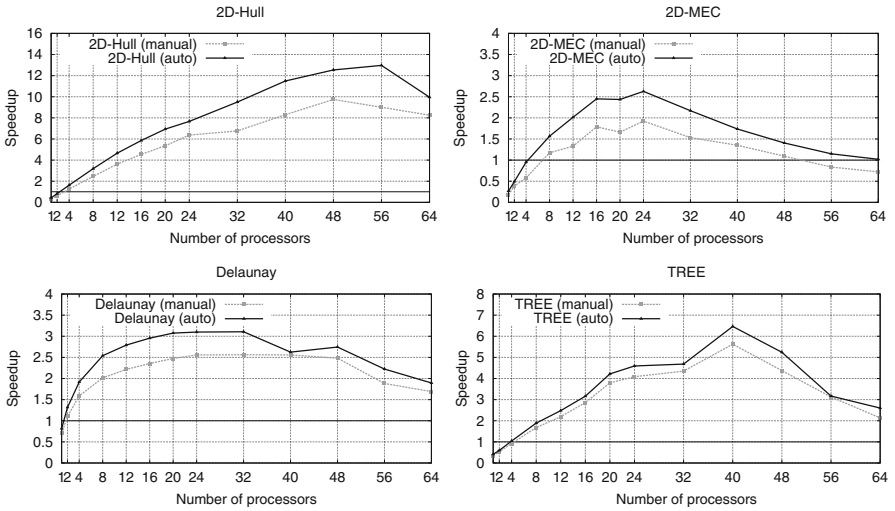

Fig. 9 Performance of the codes parallelized by using BFCA+ plus ATLaS compared with the performance achieved by the corresponding manually parallelized codes

These experiments have been run on a 64-processor server, equipped with four 16-core AMD Opteron 6376 processors at 2.3 GHz and 256 GB of RAM, which runs Ubuntu 12.04.3 LTS. All threads had exclusive access to the processors during the execution of the experiments, and we used wall clock times in our measurements.

To test the performance of the applications automatically transformed, we compare two different, parallel versions of each benchmark. The first version, called *auto* in the plots, has been obtained with the use of BFCA+, and therefore includes OpenMP parallel directives that make use of the *speculative* clause. The second one, called *manual*, has been obtained by manually augmenting the same loop with all the code needed by the runtime speculative library, a rather complex transformation procedure that transforms the original code to something similar to the code shown in Fig. 7.[3]

Figure 9 shows the relative performance between these two versions of each benchmark. We have found that the use of the OpenMP *speculative* clause inserted by BFCA+ leads to slightly better execution times than those obtained with the equivalent manual transformations. The reason is that ATLaS is not a preprocessor than first produces a code similar to the *manual* one, and later asks `gcc` to compile it. Instead, ATLaS is a GCC plugin that works directly with the GIMPLE intermediate representation, performing the transformations after several optimizations have been triggered by `gcc`. In the benchmarks considered, the *auto* versions are faster than the *manual* versions, although it is not guaranteed that this will be the case for any other application. See [6] for additional details on the behavior of the ATLaS GCC plugin.

[3] Note that the manual transformation process is included to figure out which loop would be more profitable to be parallelized and then perform an in-depth analysis of the data elements being accessed inside the loop. This is an error-prone, time-consuming process that, for the benchmarks considered, took between 10 and 30 h.

For the 2D-Hull, our solution achieves a peak speedup of 13×, with an improvement of 30 % over the manual parallelization of the code. The parallelization of the loop with our solution is straightforward, while the manual changes needed to use the TLS runtime library need more than 30 h of carefully replacing all loads and stores with function calls and changing the loop structure to support thread scheduling. In the case of the 2D-MEC, with a larger amount of dependency violations, our solution achieves minor speedups, with peaks of 2.6×. Although these are not big figures, the manual use of the TLS library to parallelize this application requires more than 10 h of a very specialized work. Moreover, our solution leads to a 39 % of performance improvement. Delaunay's execution produces a high number of dependency violations, which affects the speedup. Delaunay achieves a peak performance of 3.1× speedup with our solution, a 18 % faster than the manual approach. The programming effort to obtain a speculative version for this benchmark is very similar to the one needed for 2D-Hull. TREE obtains a peak of 6.5× speedup with our solution, improving the manual parallelization in a 12 %. This benchmark is characterized by the presence of reductions over sum and maximum operations that involve speculative variables, which avoid the parallelization by the compiler. These situations are easily resolved by our proposed clause, while handling them manually requires more than 10 h of programming effort. Finally, in all cases it can be seen that performance starts to degrade if we keep adding more threads to the work. The reason is that the TLS management cost increases with the number of threads. Please refer to [7] for a detailed analysis of the management costs and the potential performance losses of the ATLaS framework.

5 Conclusions

This paper shows a solution to the problem of the automatic synthesis and generation of OpenMP constructs, including those needed to support thread-level speculation. We propose a system, called BFCA+, that takes advantage of two existing frameworks (BFCA and ATLaS) to allow the automatic parallelization of a given loop, regardless of the presence of potential dependency violations, by simply indicating its line number. Our future work includes the automatic selection of the most promising target loop, a decision that requires the use of appropriate heuristics, and to extend BFCA+ to support other TLS frameworks such as SpLIP [8] or MiniTLS [9]. BFCA+ is freely available under request.

Acknowledgements This research has been partially supported by MICINN (Spain) and ERDF program of the European Union: HomProg-HetSys project (TIN2014-58876-P), CAPAP-H5 network (TIN2014-53522-REDT), and COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS).

References

1. Aldea S, Llanos DR, Gonzalez-Escribano A (2012) Using SPEC CPU2006 to evaluate the secuential and parallel code generated by commercial and open-source compilers. J Supercomput 59(1):486–498
2. Cintra M, Llanos DR (2003) Toward efficient and robust software speculative parallelization on multiprocessors. In: PPoPP'03 proceedings, pp 13–24

3. Dang FH, Yu H, Rauchwerger L (2002) The R-LRPD test: speculative parallelization of partially parallel loops. In: IPDPS'02 proceedings, pp 20–29
4. Aldea S, Llanos DR, Gonzalez-Escribano A (2012) Support for thread-level speculation into OpenMP. In: IWOMP'12 proceedings, pp 275–278
5. Aldea S, Llanos DR, Gonzalez-Escribano A (2014) The BonaFide C analyzer: automatic loop-level characterization and coverage measurement. J Supercomput 68(3):1378–1401
6. Aldea S, Estebanez A, Llanos DR, Gonzalez-Escribano A (2014) A new GCC plugin-based compiler pass to add support for thread-level speculation into OpenMP. In: EuroPar'14 proceedings, LNCS 8632, Springer, pp 234–245
7. Aldea S et al (2015) An OpenMP extension that supports thread-level speculation. IEEE Trans Partial Distrib Syst (to appear)
8. Oancea CE, Mycroft A, Harris T (2009) A lightweight in-place implementation for software thread-level speculation. In: SPAA 2009 proceedings, pp 223–232. ACM, New York
9. Yiapanis P et al (2013) Optimizing software runtime systems for speculative parallelization. ACM Trans Arch Code Optim (TACO) 9(4):39
10. Adhianto Laksono et al (2000) Tools for OpenMP application development: the POST project. Concurr Pract Exp 12:1177–1191
11. Ierotheou Cos S et al (2005) Generating OpenMP code using an interactive parallelization environment. Parallel Comput 31(10–12):999–1012
12. Jin Haoqiang et al (2003) Automatic multilevel parallelization using OpenMP. J Sci Program EWOMP'11 11(2):177–190 (2)
13. Johnson S et al (2005) The ParaWise expert assistant—widening accessibility to efficient and scalable tool generated OpenMP code. In: Proceedings of the WOMPAT'04, pp 67–82
14. Bondhugula, Uday et al (2008) A practical automatic polyhedral parallelizer and locality optimizer. In: PLDI'08 proceedings, pp 101–113
15. Trifunovic K et al (2010) Graphite two years after: first lessons learned from real-world polyhedral compilation. In: GROW'10 proceedings, pp 4–19
16. Grosser T et al (2011) Polly—polyhedral optimization in LLVM. In: IMPACT'11 workshop proceedings, Charmonix, France, pp 1–6
17. Lattner C, Adve V (2004) LLVM: a compilation framework for lifelong program analysis transformation. In: CGO'04 proceedings, pp 75–86 (2004)
18. Amini M et al (2012) Par4All: from convex array regions to heterogeneous computing. In: IMPACT'12 HiPEAC workshop proceedings, Paris, France, pp 1–2
19. Guelton S (2011) Building source-to-source compilers for heterogeneous targets. PhD thesis, Universit europenne de Bretagne, Rennes (2011)
20. Amini M et al (2011) PIPS is not (just) polyhedral software. In: IMPACT'11 workshop proceedings, Charmonix, France, pp 7–12
21. Liao C et al (2008) Automatic parallelization using OpenMP based on STL semantics. In: Languages and compilers for parallel computing (LCPC)
22. Dave Chirag et al (2009) Cetus: a source-to-source compiler infrastructure for multicores. IEEE Comput 42(12):36–42
23. Taillard J, Guyomarch F, Dekeyser JL (2008) A graphical framework for high performance computing using an MDE approach. In: PDP'08 proceedings, pp 165–173
24. Nardi L et al (2012) YAO: a generator of parallel code for variational data assimilation applications. In: HPCC'12 proceedings, pp 224–232
25. Clarkson KL, Mehlhorn K, Seidel R (1993) Four results on randomized incremental constructions. Comput Geom Theory Appl 3(4):185–212
26. Devroye L, Mücke EP, Zhu B (1998) A note on point location in Delaunay triangulations of random points. Algorithmica 22:477–482
27. Welzl E (1991) Smallest enclosing disks (balls and ellipsoids). In: New results and new trends in computer science. LNCS, vol 555. Springer, New York, pp 359–370
28. Barnes JE (1997) TREE. Institute for Astronomy, University of Hawaii. ftp://hubble.ifa.hawaii.edu/pub/barnes/treecode/