

Towards a compiler framework for thread-level speculation

Sergio Aldea, Diego R. Llanos, and Arturo González-Escribano
 Dpto. de Informática
 Universidad de Valladolid
 Valladolid, Spain
 saldeal@gmail.com, {diego,arturo}@infor.uva.es

Abstract—Speculative parallelization techniques allow to extract parallelism of fragments of code that can not be analyzed at compile time. However, research on software-based, thread-level speculation will greatly benefit from an appropriate compiler framework for easy prototyping and further development of new techniques. This paper presents an experimental XML-based compilation framework to handle speculative parallelization of C code. The framework extends Cetus, a source-to-source C compiler, to build an XML tree based on the Cetus Internal Representation of the source code. Other modules of our framework rely on XPath and XSLT capabilities to process the XML tree generated, to perform analysis on the use of variables and to augment the original code for software-based, speculative parallel execution. The use of the current version of our framework allows a fast prototyping of new analysis and transformation solutions, with a reduction of around 83% on the number of code lines needed with respect to the direct use of Cetus for the same purpose.

To show the possibilities of this framework, we present an automatically-generated classification of loops for several SPEC CPU2006 C benchmarks. This classification is useful to better understand the potential benefits derived from the use of speculative parallelization techniques. The development framework presented here is freely available under request.

Index Terms—speculative parallelization; compiler framework; XML; XPath; XSLT;

I. BACKGROUND

Speculative parallelization (SP), also called Thread-Level Speculation [1], [2], [3] or Optimistic Parallelization [4], [5] aims to automatically extract loop- and task-level parallelism when a compile-time dependence analysis can not guarantee that a given sequential code can be safely executed in parallel. Speculative parallelization optimistically assumes that the code can be executed in parallel, and relies on a runtime hardware or software monitor to ensure that no dependence violation is produced. In the presence of such a violation, earlier software-only speculative solutions [1], [6] interrupted the speculative execution and re-executed the loop serially. More recent approaches [2], [7], [8] rely on a monitor that stops only the offender thread and its successors, re-starting them with the correct data values. As long as not many dependence violations arise, speculative parallelization may speed up these non-analyzable fragments of code.

Speculative parallelization can be either implemented in hardware or software. While hardware mechanisms do not need changes in the code and do not add overheads to

speculative execution, they require changes in the processors and/or the cache subsystems (see e.g. [9], [10], [11], [12], [13]). Software-based speculation, on the other hand, requires to augment the original code with instructions that drive the runtime dependence analysis. Although these instructions imply a performance overhead, software-based SP can be effectively used in current shared-memory systems with no hardware changes.

To better understand how SP works, we will briefly describe the different situations that may arise when two threads access the same variable concurrently. Informally speaking and focusing on loop-based speculation, variables that are always written before being read in the context of a given iteration are called *private*. Variables that are only read and not written in the whole loop are called *read-only shared* variables. If a compiler detects that all variables inside a loop are either private or read-only shared, then the loop can be parallelized safely¹. Unfortunately, most loops have variables whose values might be written in a particular iteration and later be read in a subsequent iteration. Sequential semantics impose a total order for both operations, and if these two operations are done out-of-order by different threads a dependence violation occurs. In this case, the results generated by the thread that consumed the outdated value of such *speculative* variable should be discarded, together with all the results generated by its successors. This is called a *squash* operation.

Despite the fact that SP effectively helps in the extraction of parallelism for non-analyzable code, until now these techniques have had a limited impact in the compiler community. There are two basic reasons for this. First, depending on the number of dependence violations that arise at runtime, the speculative overhead may not compensate for the performance gain obtained. Second, the lack of a SP compiler framework makes difficult to measure the impact of SP on widely-used sequential benchmarks, since both the classification of variable usage and the source code modifications needed should be done manually, a tedious and error-prone task. The programmer should first classify the variable usage into private, read-only shared and speculative categories, a daunting task if the code has more than a few dozens of lines. After

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

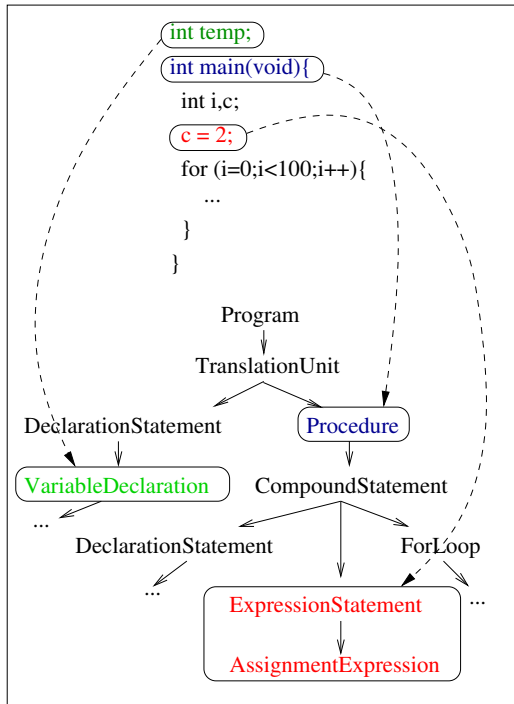


Figure 1: IR Tree Structure Example.

that, the code should be modified to insert all the speculative parallelization calls, requiring an in-depth knowledge of the speculative parallelization scheme being used.

In this paper we address this second problem. We present a source-to-source compiler framework designed to classify variable usage for a given loop and to augment the code to benefit from software-based speculative parallelization. Our framework is partially based on the Cetus source-to-source C compiler. Cetus has been modified to generate an XML tree based on the Internal Representation (IR) of the source code. A second tool, Loopest, relies on XPath capabilities to perform analysis on variables usage and loops. This information can be used automatically to augment the XML tree with the code needed for speculative execution. Finally, a third tool called Sirius translates the resulting XML tree back to C code. In this paper we describe the current state of our experimental compiler framework in detail, together with a summary of results obtained when using its analysis capabilities with several C benchmarks of the SPEC2006 compiler suite.

II. UNDERSTANDING CETUS

Cetus [14] is a compiler infrastructure written in Java for source-to-source transformation of C programs developed by Purdue University. Cetus provides several functions, such as auto-parallelization of loops through private- and shared-variables analysis, and automatic insertion of OpenMP directives [15].

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 their class member functions.

In Cetus, the concept of statements and expressions are closely related to the syntax of the C language, making the source-to-source translation process easy. However, there are some disadvantages: an increasing complexity for pass writers (since they should think in terms of C syntax) and limited extensibility to process additional languages. Fortunately, this problem is mitigated by the provision of several abstract classes, which represent generic control constructs. Thus, generic passes can be written using the abstract interface, while more language-specific passes can use the derived classes.

Figure 1 shows an example of Cetus IR from a C source code. In Cetus terminology, a “TranslationUnit” is a file containing source code. The syntax tree and the class hierarchy are not equivalent. For example, in the syntax tree, the parent of a TranslationUnit is a Program, however neither TranslationUnit nor Program have a parent in the class hierarchy.

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 for developing our compiler framework, we modify it to build an XML representation of its Intermediate Representation, and we use XML standard tools to perform queries and modifications to the structure. The next section describes the framework in more detail.

III. COMPILER FRAMEWORK ARCHITECTURE

Figure 2 shows our source-to-source compiler framework architecture. Solid lines represent the data flow currently implemented; dashed lines represent work still in progress. The input of this system is the original C file. A modified version of Cetus, called XMLCetus, uses Cetus’ IR to build an XML tree representing the original C code, containing all the information needed to analyze and rebuild the source files.

A second tool, called Loopest, receives this input file and uses XPath to query the XML DOM tree. Queries currently implemented perform a dependence analysis of scalar variables and looks for other constructs (like memory management, I/O function calls and pointer arithmetic) in the context of every single *for* loop. With this information, Loopest generates an analysis report. As an example, in Sect. IV we will use this framework to process the SPEC CPU2006 benchmark.

Loopest is currently being extended in two ways: to also analyze the dependence pattern of array structures and dynamic memory, needed for the application of speculative parallelization, and to transform the XML tree representing the original C file into a speculative, parallel version (dashed arrow labeled “modified XML”).

We have also developed a third tool, called Sirius, that uses XSLT to transform any XML representation of a C file back into C code. For legibility, we use the GNU tool *indent* to format the result generated by Sirius (not shown in the figure).

After briefly reviewing the overall framework architecture, we will now discuss XMLCetus, Loopest and Sirius in more

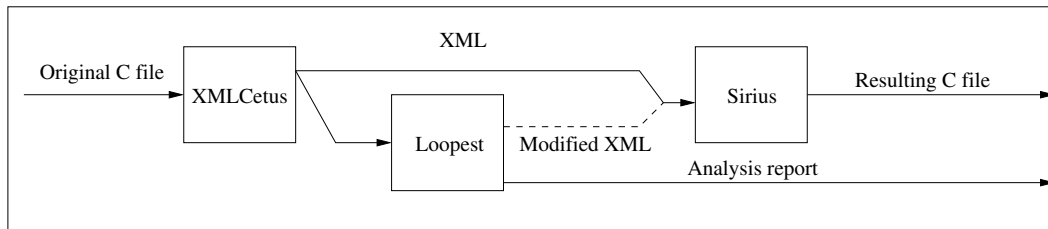


Figure 2: Compiler framework architecture.

detail.

A. Converting Cetus IR to XML

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, a new function called *createDomTree()* is invoked with the *Program* node (the first node of the tree) as input parameter. Every node of the IR will have a corresponding representation in the XML DOM tree, thus preserving the original structure of the Cetus IR. The following steps explain the transformation procedure from Cetus IR to XML DOM tree.

- Beginning with the first node, *Program*, XML Cetus recursively descends all its *TranslationUnit* children nodes, representing the source code files passed to Cetus. The tree is traversed in preorder, depth-first search. A *getChilden()* function gets the children of each node as a list of nodes. Through a casting operation, the list is transformed into a *Traversable*-type objects list. *Traversable* is the type defined by Cetus as the generic class, representing any kind of node.
- Next, the type of the children nodes are checked. If one node is an instance of both a given class and its parent class, two DOM elements that represent both nodes are created to reflect the original structure. This is just an example of the several situations that may arise when creating DOM nodes and their relationships.
- When the instance is checked as belonging to a particular class, an object instance of this class is created. Now, it is possible to obtain relevant information from the node and create a new DOM element with attributes that reflects this information.
- Finally, after the creation of the element with its attributes, the DOM element is appended to its corresponding parent.

This procedure generates an XML document which represents the DOM tree, and can be printed using the *printDomTree()* function.

B. Querying and Modifying the XML tree

Loopest is an experimental Java tool that provides two functionalities: (a) generation of reports on the use of variables in the context of every single *for* loop present in the source code, and (b) modification of the XML tree to insert directives

and functions that allow the speculative parallelization of the original source code.

Loopest relies on XPath capabilities to perform queries and generate reports based on the XML tree generated by XMLCetus. XPath syntax is easy to learn and provides enough functionality for this purpose. With XPath, it is possible to build complex queries with few words or lines. The result of these queries may be new node-sets that can be combined to search for new results. One of the main advantages of using such a tool is that its functionality can be modified easily, detecting new language constructions by adding or modifying queries, thus allowing fast prototyping of new solutions.

XPath queries work in a similar way than recursive searches in a directory-based filesystem structure, allowing to select nodes or set of nodes in an XML document. We have found that such queries are much simpler to develop than to directly modify the Java code that manages the IR structure in Cetus.

In order to classify variable usage inside *for* loops, Loopest executes a set of XPath queries which determine the variables being read, being written, being read-and-written, and some other queries useful to find private, read-only shared and speculative variables in a loop, including detection of private variables that are used after the end of the loop. These queries process all *for* loops present in the code, regardless of their depth level.

Loopest has two different functionalities. The first one is to generate a report with the classification of variable usage (using the *ListUtils* package, provided by Apache Commons [16]), while the second one is to augment a given loop with all the instructions needed to execute that loop speculatively in parallel. To do so, the current version of Loopest needs the line number where the target *for* loop starts. We are currently working in other alternatives, such as using profile-based analysis and different heuristics to choose the target loop automatically.

C. Regenerating the C code

After building, analyzing and modifying the XML tree, the last part of the process is to convert this representation back into C code. To do so, we have developed a Java tool 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 capabilities, and uses template rules to translate the XML output document again to C. To apply the XSLT

transformation rules, we chose the Saxon tool [17], due to its open-source nature and because it implements XPath and XSLT 2.0.

The structure of the XSLT program developed consists of a set of template rules, one for each element of the DOM tree that should be transformed into C language elements. These template rules generate the C code that corresponds to the DOM element identified, and indicate the application of another rule where necessary. In another cases, Sirius defines modes for the template rules in order to apply them depending on the context. This differentiation is necessary to correctly rewrite the code. As a result of the transformation made by Sirius, a C source file is generated. At this point, we use the GNU tool *indent* to format the output file and make it more pleasant for the programmer. With this transformation Sirius finishes the compilation framework process.

D. Using Cetus for the same purpose

It is important to highlight the differences between our framework and a similar system based exclusively on Cetus. The main differences between both approaches are simplicity and extensibility. Detection of private and read-only shared variables is also within Cetus capabilities, but the code required to implement this functionality is much longer and complex than Loopest's code. Modifying Cetus requires a deep knowledge of Java, Cetus Intermediate Representation, and its associated data structures, while adding new functionalities in our system can be done simply adding new XPath queries, that 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 Cetus at least eight Java classes take part directly to locate the private variables of a given loop, consuming 2573 lines of code (calculated with SLOCCount [18]). However, Loopest only needs 425 lines of lower-complexity code to carry out the same task, an 83% reduction in the number of code lines needed.

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 speculative parallelization, just modifying or creating new XPath queries or XSLT transformations.

IV. LOOP CLASSIFICATION OF SOME SPEC CPU2006 BENCHMARKS

As an example of the capabilities of our compiler framework, Table I shows a loop classification generated automatically with our tools, for several applications in the SPEC CPU2006 benchmark [19]. This list includes all C applications except 400.perlbench and 403.gcc. This table does not pretend to show an in-depth loop characterization of SPEC applications, since such a study requires additional information about

the loop coverage in terms of sequential execution. However, we believe that this example is useful to show the capabilities of the compiler framework developed.

Regarding the two C benchmarks that have not been included in this study, we do not show their loop characterization because their source code is fragmented in many C files (53 for 400.perlbench and 155 for 403.gcc) with many conditional compilation flags, and the current version of Cetus is not able to build a single IR for them. We are currently transforming both benchmarks into a single C file to overcome this limitation, but the number of code lines for both of them (around 116000 for 400.perlbench and 484000 for 403.gcc) is hindering this process.

A detailed description of each column of Table I follows, together with some comments regarding the possibility of using speculative parallelization techniques with these loops. For each benchmark considered, the table shows the following information:

- The number of *for* loops present in the benchmark.
- Percentage of "classical" *for* loops. These are loops with a single control variable, with all three fields of the FOR structure (initialization, conditional evaluation, and increment) being used, and with no changes in the control variable inside the loop body. Current implementation of the speculative engine used can only deal with this kind of loops, accounting for more than 85% of the total number of loops on average.
- Percentage of loops that contain uses of pointer variables. The average value (80%) reflects the importance of supporting pointer arithmetic in speculative parallelization schemes, a problem that is not yet solved in the general case.
- Percentage of loops that perform calls to memory management functions, such as *malloc()* or *free()*. On average, only 1.17% of loops make use of dynamic memory capabilities. Although this datum should be complemented with the actual coverage of this 1.17% of loops in terms of sequential execution time, this preliminary result suggests that dynamic memory management might not be a priority in the list of problems that speculative parallelization techniques should solve to speed-up these benchmarks.
- Percentage of loops that contain I/O function calls. Current speculative parallelization implementations are not able to handle I/O speculatively, so these loops can not be parallelized.
- Percentage of loops that only hold private and read-only shared variables, and therefore may be parallelized at compile time. Although our study indicates that roughly half of the loops fall into this category, the reader should take into account that in most cases it may not be profitable to parallelize these loops because thread-management overheads may lead to slowdowns.

This is just an example of the studies that can be conducted with our framework. The flexibility provided by XML tools

Application	Lines of code	Number of FOR Loops	Classical Loops (%)	Loops with pointers (%)	Loops with memory management (%)	Loops with I/O activity	Parallelizable loops (%)
401.bzip2	7 292	120	93.33	88.33	0.83	7.5	46.67
429.mcf	2 044	33	63.64	96.97	0	12.12	30.3
433.milc	12 837	418	64.11	87.56	2.39	19.86	33.25
456.hmmer	33 210	739	88.23	95.81	3.52	13.67	46.55
458.sjeng	13 291	216	91.67	10.19	0	4.17	51.39
462.libquantum	3 454	89	92.13	87.64	0	7.87	48.31
464.h264ref	46 142	1 792	95.31	88.23	3.23	0.89	57.59
470.lbm	875	23	100	78.26	0	43.48	69.57
482.sphinx3	18 280	556	80.4	93.53	0.54	14.2	46.04
Average	15 270	443	85.42	80.72	1.17	13.75	47.74

Table I: Loop characterization results for some C applications of the SPEC CPU2006 benchmark.

makes easy to modify XPath queries to further investigate the possibility of using speculative parallelization techniques in a different context, such as task-based speculative parallelism.

The development framework presented here is freely available under request.

V. SUMMARY AND FUTURE WORK

This paper presents a source-to-source compiler infrastructure based on Cetus that takes advantage of XML representation and its associated analysis and transformation tools. This infrastructure is primarily intended to be used in the context of speculative parallelization studies, but can be easily modified for other purposes. 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. Regarding code modification, our system currently relies on the user's choice of the loop to be modified in order to augment it with speculative parallelization capabilities. We are working in different mechanisms to choose these loops automatically.

It is important to highlight that this compiler infrastructure is not intended to be directly inserted into a production compiler. Instead, we have developed a flexible and robust tool that allows fast prototyping of new solutions regarding code analysis and modification. Our current and future work include the use of this tool with two purposes. First, to discover parallelization niches in widely-used benchmarks that may benefit from speculative parallelization. Second, to use this information to decide what limitations of current speculative parallelization schemes ought to be solved first. We hope that this process will let speculative parallelization technology be mature enough to be included in mainstream compilers.

ACKNOWLEDGEMENTS

This research is partly supported by the Ministerio de Educación y Ciencia, Spain (TIN2007-62302), Ministerio de Industria, Spain (TSI-020302-2008-89, CENIT MARTA, CENIT OASIS, CENIT OCEANLIDER), Junta de Castilla y León, Spain (VA094A08), and the HPC-EUROPA2 project (project number: 228398) with the support of the European Commission - Capacities Area - Research Infrastructures Initiative.

REFERENCES

- [1] M. Gupta and R. Nim, "Techniques for speculative run-time parallelization of loops," in *Proc. 1998 ACM/IEEE Conf. on Supercomputing*, (San Jose, CA), pp. 1–12, IEEE, 1998.
- [2] F. Dang, H. Yu, and L. Rauchwerger, "The R-LRPD test: Speculative parallelization of partially parallel loops," in *IPDPS'02*, Apr. 2002.
- [3] M. Cintra and D. R. Llanos, "Toward efficient and robust software speculative parallelization on multiprocessors," in *ACM PPOPP'03*, (San Diego, California, USA), pp. 13–24, ACM, 2003.
- [4] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *ACM PLDI'07*, (San Diego, California, USA), pp. 211–222, ACM, 2007.
- [5] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew, "Optimistic parallelism benefits from data partitioning," in *ASPLOS'08*, (Seattle, WA, USA), pp. 233–243, ACM, 2008.
- [6] L. Rauchwerger and D. Padua, "The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization," in *ACM PLDI'95*, (La Jolla, CA, USA), pp. 218–232, ACM, 1995.
- [7] M. Cintra and D. R. Llanos, "Design space exploration of a software speculative parallelization scheme," *IEEE Tran. on Par. and Distr. Systems*, vol. 16, no. 6, pp. 562–576, 2005.
- [8] P. Rundberg and P. Stenström, "Low-Cost Thread-Level data dependence speculation on multiprocessors," in *Workshop on Scalable Shared Memory Multiprocessors*, June 2000.
- [9] L. Hammond, M. Willey, and K. Olukotun, "Data speculation support for a chip multiprocessor," in *ASPLOS'98*, (San Jose, CA, USA), pp. 58–69, ACM, 1998.
- [10] P. Marcuello and A. González, "Clustered speculative multithreaded processors," in *Proc. of the 13th Intl. Conf. on Supercomputing*, (Rhodes, Greece), pp. 365–372, ACM, 1999.
- [11] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," in *ISCA'00*, (Vancouver, British Columbia, Canada), pp. 1–12, ACM, 2000.
- [12] M. Cintra, J. F. Martínez, and J. Torrellas, "Architectural support for scalable speculative parallelization in shared-memory multiprocessors," in *ISCA'00*, (Vancouver, British Columbia, Canada), pp. 13–24, ACM, 2000.
- [13] L. R. Y. Zhan and J. Torrellas, "Hardware for speculative Run-Time parallelization in distributed Shared-Memory multiprocessors," in *HPCA'98*, p. 162, IEEE Computer Society, 1998.
- [14] C. Dave, H. Bae, S. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A Source-to-Source compiler infrastructure for multicores," *Computer*, vol. 42, no. 12, pp. 36–42, 2009.
- [15] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald, *Parallel Programming in OpenMP*. Morgan Kaufmann, 1 ed., Oct. 2000.
- [16] W. Iverson, *Apache Jakarta Commons: Reusable Java(TM) Components*. Prentice Hall PTR, Feb. 2005.
- [17] D. Tidwell, *XSLT, 2nd Edition*. O'Reilly Media, 2 ed., June 2008.
- [18] D. A. Wheeler, "Sloccount: Counting source lines of code." <http://www.dwheeler.com/sloccount/>.
- [19] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.