Universidad de Valladolid

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

DEPARTAMENTO DE INFORMÁTICA

TESIS DOCTORAL:

Compile-Time Support for Thread-Level Speculation

Presentada por **D. Sergio Aldea López**
para optar al grado de
doctor por la Universidad de Valladolid

Dirigida por:

Dr. Diego R. Llanos Ferraris

Valladolid,
Julio 2014

Resumen

Durante la última década, el desarrollo de procesadores multinúcleo ha incrementado significativamente las capacidades paralelas de los sistemas de cómputo. Sin embargo, la gran cantidad de código secuencial ya existente no puede aprovechar de forma directa estas capacidades, lo que obliga al programador a paralelizar sus aplicaciones. Esta es una tarea que requiere un amplio conocimiento tanto de la aplicación como de la arquitectura subyacente. Técnicas como la paralelización especulativa facilitan esta tarea, al ocuparse de garantizar que el código paralelo cumple con la semántica secuencial. Sin embargo, estas técnicas también requieren la intervención manual de programadores experimentados.

Esta Tesis aborda este problema proponiendo una nueva cláusula OpenMP, `speculative`, que define aquellas variables que pueden provocar una violación de dependencia, y un sistema en tiempo de compilación que transforma esta nueva cláusula en llamadas a una librería en tiempo de ejecución paralela especulativa. Estas propuestas garantizan que todos los accesos a variables especulativas se realizan respetando la semántica secuencial del programa, y evita obligar al programador a que modifique manualmente el código para soportar su paralelización especulativa.

Antes de instrumentar un bucle con construcciones OpenMP, incluida nuestra cláusula propuesta, el programador necesita extraer cierta información del código a paralelizar. Sin herramienta automáticas, el programador tiene que extraer esta información manualmente. Esta Tesis aborda también el problema de la caracterización automática de bucles secuenciales con el objetivo de encontrar nichos de paralelización en *benchmarks* conocidos que puedan beneficiarse de la paralelización especulativa basada en software. Para ello, proponemos un sistema que aprovecha una representación intermedia XML para realizar un análisis estático del código fuente y lo combina con la información de su perfil de ejecución, obteniendo la caracterización del código. Además, esta Tesis propone un sistema que aprovecha esta información, sintetizando y generando automáticamente las directivas y cláusulas OpenMP necesarias para paralelizar un código especulativamente.

La incorporación de nuestra cláusula propuesta, `speculative`, en la implementación de OpenMP de un compilador tan extendido como GCC, junto con la automatización de todo el proceso de paralelización, ayudará a que la paralelización especulativa basada en software esté lo suficientemente madura como para ser utilizada en producción.

Palabras clave

Paralelización especulativa, TLS, XML, XPath, análisis de código, perfiles de ejecución, representación de código fuente, transformación de código, GCC, plugin, OpenMP, generación automática de código.

Abstract

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 architectures 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. Parallelization techniques such as Thread-Level Speculation (TLS) eases this task, ensuring that the parallel code satisfies the sequential semantics. Nevertheless, this technique also requires manual and tough intervention by expert programmers.

This Ph.D. thesis addresses this problem defining a new OpenMP clause, called `speculative`, which points out those variables that may lead to dependency violation, and a compile-time system that seamlessly translates this new OpenMP clause into calls to our TLS runtime system. This ensures that all accesses to these speculative variables will be carried out according to sequential semantics, and frees programmers from the manual augmentation of the source code required by the speculative parallelization.

Before instrumenting a loop with OpenMP constructs, including our proposed `speculative` clause, programmers firstly need to extract certain information about the source code that they aim to parallelize. Without automatic tools, programmers have to manually extract the information. This Ph.D. thesis also addresses the problem of automatic characterization and coverage of sequential loops, with the aim of finding parallelization niches in widely-used benchmarks that may benefit from software-based speculative parallelization. To do this, we have proposed a system that takes advantage of an XML-based representation of the source and combines profiling information to extract all this information. Besides, we have also proposed a system that leverages this information, automatically synthesizing and generating the OpenMP constructs needed to parallelize the source code speculatively.

We believe that the implementation of the new OpenMP clause in a mainstream compiler, together with the automation of the whole process of the parallelization, will help thread-level speculation to be mature enough for its production use.

Keywords

Speculative parallelization, Thread-Level-Speculation, TLS, XML, XPath, profiling information, code analysis, compiler framework, source code representation, source code transformation, GCC, plugin, OpenMP, automatic code synthesis.

A mis padres y Beatriz

Agradecimientos

Esta Tesis no se habría hecho realidad sin el consejo y la guía de mi tutor, Diego. Su infatigable trabajo y su exigente motivación me han permitido seguir adelante, no bajar los brazos nunca, y finalizar orgullosamente esta Tesis. Trabajando con Diego desde el proyecto de la Ingeniería Técnica, han sido siete años que han marcado mi forma de ser, de trabajar, y de ver la informática. Espero que esta Tesis haga justicia a su trabajo.

A lo largo de estos años también ha sido muy importante la figura del Profesor Arturo González Escribano. Sus comentarios siempre me han hecho reflexionar, han aportado una visión diferente y enriquecedora a mi investigación, y sin duda han elevado la calidad de la misma.

Quiero dar las gracias a mis compañeros en esta aventura: Yuri, Javier, Héctor, Álvaro y Ana. Sus risas, bromas, y compañía han generado un ambiente de trabajo que echaré de menos y será díficil de repetir. Las horas de trabajo en el laboratorio se pasarían sin duda mucho más despacio sin su compañía.

También quiero agradecer a mis amigos los buenos ratos, las noches de olvido, y los findes de camadería. Sin ellos, hoy estaría un poco menos cuerdo y bastante más lejos de terminar la Tesis.

Dejo para el final las personas más importantes de mi vida; mis padres, que me enseñaron que no hay nada que no pueda hacer, y cuya sabiduría me ha llevado hasta obtener la máxima formación académica; y Beatriz, que me demuestra que no hay nada que no podamos hacer, y cuya sonrisa sacó adelante esta Tesis en muchos momentos.

GRACIAS.

Sergio Aldea López

“I don’t know where I’m going, but I’m on my way”.

Carl Sandburg, Incidentals (1904)

Contents

List of Figures

List of Tables

Resumen de la tesis

Una de las principales preocupaciones de las ciencias de la computación es el estudio de las capacidades paralelas tanto de programas como de los procesadores que los ejecutan. Existen varias razones que hacen muy deseable el desarrollo de técnicas que paralelicen automáticamente el código. Entre ellas se encuentran el inmenso número de programas secuenciales existentes ya escritos, la complejidad de los lenguajes de programación paralelos, y los conocimientos que se requieren para paralelizar un código. Sin embargo, los actuales mecanismos de paralelización automática implementados en los compiladores comerciales no son capaces de paralelizar la mayoría de los bucles en un código, debido a la dependencias de datos que existen entre ellos. Por lo tanto, se hace necesaria la búsqueda de nuevas técnicas que saquen beneficio de las potenciales capacidades paralelas del hardware y arquitecturas multiprocesador actuales. Estas técnicas requieren la intervención manual de programadores experimentados, y la paralelización especulativa no es una excepción.

Esta Tesis aborda este problema definiendo una nueva cláusula OpenMP, llamada `speculative`, que permite señalar qué variables pueden llevar a una violación de dependencia. Además, esta Tesis también propone un sistema en tiempo de compilación que, usando la información sobre los accesos a las variables que proporcionan las cláusulas OpenMP, añade automáticamente todo el código necesario para gestionar la ejecución especulativa de un programa. Esto libera al programador de modificar el código manualmente, evitando posibles errores y una tediosa tarea.

Antes de instrumentar un bucle con directivas y cláusulas OpenMP, incluyendo nuestra propuesta de cláusula `speculative`, los programadores deben extraer cierta información sobre el código fuente que quieren paralelizar. Sin herramientas automáticas, los programadores tienen que extraer manualmente esta información. Esto incluye el uso de las variables dentro de cada bucle; funciones de entrada y salida que puedan complicar o incluso impedir la paralelización; y aún más importante, determinar si merece la pena paralelizar un bucle, o si la sobrecarga necesaria para gestionar los diferentes hilos será mayor que el beneficio obtenido de la paralelización.

Esta Tesis aborda también el problema de la caracterización automática de bu-

cles secuenciales con el objetivo de encontrar nichos de paralelización en *benchmarks* conocidos que puedan beneficiarse de la paralelización especulativa basada en software. Para hacer esto, proponemos un sistema que aprovecha una representación intermedia XML para realizar un análisis estático del código fuente y lo combina con la información de su perfil de ejecución, obteniendo la caracterización del código,

Finalmente, esta Tesis propone un sistema que aprovecha esta información, sintetizando y generando automáticamente las directivas y cláusulas OpenMP necesarias para paralelizar un código especulativamente. La incorporación de nuestra cláusula `speculative` en la implementación de OpenMP de un compilador tan extendido como GCC, junto con la automatización de todo el proceso de paralelización, ayudará a que la paralelización especulativa basada en software esté lo suficientemente madura como para ser utilizada en producción.

R.1 Metodología de investigación

La metodología de investigación seguida por esta Tesis para alcanzar los objetivos propuestos es definida por el método de investigación para ingeniera del software descrita en [2]. Este método establece cuatro etapas que el proceso de investigación tiene que seguir. Estas etapas pueden repetirse cíclicamente hasta refinar las soluciones propuestas.

1. *Observar las soluciones existentes*. Esta etapa tiene el propósito de detectar problemas que serán abordados durante el proceso de investigación, comenzando por las soluciones existentes. Esto conlleva un completo estudio del estado del arte con el objetivo de encontrar trabajos relacionados con nuestra investigación. Este estudio se encuentra dividido en diferentes secciones que se presentan dentro de cada capítulo de la Tesis.

2. *Proponer mejores soluciones*. En esta etapa se propone una solución que aborde las limitaciones encontradas en la fase anterior. Como mostraremos a lo largo de este documento, existe una carencia de sistemas que combinan paralelización automática con técnicas de paralelización especulativa basada en software, es decir, soluciones que no requieren la intervención de un programador para añadir el código extra necesario para gestionar la ejecución especulativa. Además, el motor de ejecución especulativa en el que se basa esta Tesis requiere OpenMP para clasificar las variables de los bucles a paralelizar, y por tanto, es necesario añadir una nueva cláusula que permita identificar aquellas variables que lleven a violaciones de dependencia. Hasta ahora, el motor clasificaba estas variables como *shared*, forzando al programador a añadir manualmente el código necesario para gestionar la ejecución especulativa. Este trabajo puede

evitarse definiendo una nueva categoría: *speculative*, y su cláusula asociada. Haciendo esto, un sistema puede procesar en tiempo de compilación esta nueva cláusula y añadir automáticamente el código necesario para gestionar la ejecución especulativa. El problema de cómo obtener la información necesaria para clasificar las variables en función de sus accesos también es abordado por esta Tesis.

3. *Construir o desarrollar la solución*. La solución propuesta en la etapa anterior es implementada en esta etapa. Para ello, hemos desarrollado un prototipo de sistema en tiempo de compilación que cubre las necesidades encontradas en la etapa anterior.

4. *Medir y analizar la nueva solución*. Finalmente, este método ingenieril establece que la solución propuesta tiene que resolver los problemas descubiertos en la primera etapa. Para ello, hemos usado pruebas de regresión durante el desarrollo y validación de la corrección del sistema, mientras que también hemos usado benchmarks sintéticos y aplicaciones reales para evaluar el sistema.

R.2 Metas y contribuciones

De acuerdo a los problemas identificados en la sección anterior, la pregunta de investigación de esta Tesis es la siguiente:

> *Es posible desarrollar un mecanismo en tiempo de compilación capaz de (1) detectar nichos susceptibles de ser paralelizados especulativamente, (2) evaluar su impacto en términos de tiempo de ejecución en paralelo, y (3) transformar automáticamente código secuencial en código especulativamente paralelo?*

Para responder estas preguntas de investigación, se han cumplido los siguientes objetivos intermedios.

R.2.1 Evaluación de las capacidades de los compiladores

Hemos medido las capacidades de paralelización de los compiladores comerciales, exponiendo las limitaciones de los mecanismos de paralelización automática que implementan. El estudio revela que estos mecanismos de paralelización automática sólo alcanzan un 19% de speedup en promedio para los benchmarks del SPEC CPU2006.

1. Using SPEC CPU2006 to Evaluate the Secuential and Parallel Code Generated by Commercial and Open-source Compilers. Sergio Aldea, Diego R. Llanos,

Arturo Gonzalez-Escribano. *The Journal of Supercomputing, 59(1)*, January 2012, pages 486-498.

2. Evaluación de compiladores comerciales usando SPEC CPU2006. Sergio Aldea, Diego R. Llanos, Arturo Gonzalez–Escribano. *Actas XIX Jornadas de Paralelismo*, Castellón, Spain, September 17–19, 2008.

R.2.2 Propuesta y definición de una cláusula OpenMP: `speculative`

Hemos añadido soporte para TLS en OpenMP. Para ello, hemos propuesto una nueva cláusula OpenMP que permita controlar aquellas variables que son susceptibles de provocar una violación de dependencia. Esta cláusula recibe el nombre de `speculative`, y permite la ejecución en paralelo de cualquier bucle cuyo análisis de dependencia no puede hacerse en tiempo de compilación.

3. Support for thread-level speculation into OpenMP. Sergio Aldea, Diego R. Llanos, and Arturo Gonzalez-Escribano. *Proceedings of the 8th international conference on OpenMP in a Heterogeneous World (IWOMP'12)*, Barbara M. Chapman, Federico Massaioli, Matthias S. Müller, and Marco Rorro (Eds.). Springer-Verlag, Berlin, Heidelberg, 2012. pages 275-278.

R.2.3 Diseño, implementación, y evaluación de la cláusula OpenMP `speculative`

Hemos desarrollado un plugin GCC que añade soporte en el compilador para la cláusula `speculative` propuesta en el punto anterior. Este plugin transforma el bucle que ha sido marcado con la directiva `omp parallel for` y la cláusula `speculative`, en un bucle especulativamente paralelo. Esta transformación supone la inserción de todas las llamadas a la librería de ejecución TLS que son necesarias para (a) distribuir bloques de iteraciones entre los diferentes procesadores, (b) sustituir las lecturas y escrituras sobre variables especulativas por sus correspondientes versiones especulativas, y (c) realizar las copias parciales de los valores de las variables especulativas al final de la ejecución de cada bloque de iteraciones. La librería de ejecución paralela especulativa usada en esta Tesis es la desarrollada por Estebanez, García-Yágüez, Llanos, y Gonzalez-Escribano [62, 69].

También hemos mejorado la documentación existente sobre los plugins GCC. Para ello, hemos descrito cómo programar, enlazar con GCC y ejecutar un plugin, así como la estructura interna de un plugin.

Finalmente, hemos evaluado la cláusula propuesta y el plugin desarrollado. El código generado automáticamente por el sistema no sólo obtiene speedup en aplicaciones que no son paralelizables en tiempo de compilación por mecanismos automáti-

cos convencionales, sino que también obtiene un menor rendimiento que el código paralelizado manualmente. Los speedups obtenidos mediante el sistema desarrollado son alrededor del 20% mejor que los speedups obtenidos mediante la alternativa manual. Por lo tanto, con la cláusula OpenMP `speculative`, los programadores pueden paralelizar aplicaciones evitando todas los inconvenientes y dificultades asociados a la paralelización especulativa manual.

4. A New GCC Plugin-Based Compiler Pass to Add Support for Thread-Level Speculation into OpenMP. Sergio Aldea, Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. Aceptado en Euro-Par 2014. Volumen 8632 de Lecture Notes of Computer Science. Pendiente de publicación.

5. An OpenMP extension to support Thread-Level Speculation. Sergio Aldea, Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. Submitted.

6. Una extensión para OpenMP que soporta paralelización especulativa. Sergio Aldea, Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. *Actas XXV Jornadas de Paralelismo*, Valladolid, Spain, September, 17-19, 2014.

R.2.4 Detección de nichos de paralelización especulativa y clasificación de variables

Hemos propuesto un sistema basado en XML, llamado *BonaFide C Analyzer* (BFCA), que combina el análisis estático del código fuente con la información de su perfil de ejecución con el objetivo de generar informes completos de todos los bucles de un programa. Estos informes incluyen el porcentaje de tiempo de ejecución del bucle respecto del total del programa, si un bucle es adecuado para la paralelización, una clasificación de todas las variables dentro del bucle en función de cómo son accedidas, así como otros inconvenientes que impiden el paralelismo. Toda esta información permite analizar cómo determinadas construcciones del código fuente son usadas en aplicaciones reales, ayudando al programador a paralelizar el código. Esta información puede ser procesada automáticamente para definir cláusulas OpenMP, incluyendo también nuestra cláusula propuesta para definir las variables especulativas.

BFCA nos ha permitido hacer un extenso estudio sobre las aplicaciones del SPEC CPU2006 [82]. Este estudio no sólo caracteriza cuantitativa y cualitativamente los bucles de esas aplicaciones en función de su idoneidad para ser paralelizados, sino que también informa hasta qué punto el uso de técnicas de paralelización automática puede ayudar a reducir el tiempo de ejecución. Este estudio también clasifica todos los bucles en estos benchmarks de acuerdo a diferentes dificultades que pueden afectar la paralelización, como el uso de aritmética de punteros, llamadas a funciones de entrada/salida y de gestión de memoria, y dependencias de variables globales y

estáticas. Este tipo de información es extremadamente difícil de obtener de forma manual, y puede ser usada para guiar el desarrollo de futuros proyectos en el área de la paralelización automática.

Finalmente, nuestra estudio muestra que el 47,72% de los bucles presentes en las aplicaciones analizadas son potencialmente paralelizables mediante modelos de programación paralela como OpenMP, mientras que el 37,7% únicamente pueden ser ejecutados en paralelo con la ayuda de técnicas de ejecución paralela especulativa.

7. The BonaFide C Analyzer: Automatic Loop-level Characterization and Coverage Measurement. Sergio Aldea, Diego R. Llanos, and Arturo Gonzalez-Escribano. *The Journal of Supercomputing*, 2014. Online, DOI:10.1007/s11227-014-1091-3.

8. Towards a compiler framework for thread-level speculation. Sergio Aldea, Diego R. Llanos, Arturo Gonzalez-Escribano. *Proceedings of the 19th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP 2011)*, Ayia Napa, Cyprus, February 9-11, 2011. pages 267–271.

9. Extending a source−to−source compiler with XML capabilities. Sergio Aldea, Diego R. Llanos, Arturo Gonzalez−Escribano. *Actas XXI Jornadas de Paralelismo*, Valencia, Spain, September 7−10, 2010.

10. XMLCetus y Sirius: análisis y traducción de código C utilizando herramientas XML. Sergio Aldea, Diego R. Llanos, Arturo Gonzales-Escribano. *Technical Report IT-DI-2010-001*, Departamento de Informática, Universidad de Valladolid, 2010.

R.2.5 Síntesis automática de código especulativo

Hemos propuesto una solución que aprovecha la clasificación de las variables que realiza BFCA para generar automáticamente una versión paralela del código analizado, insertando directivas y cláusulas OpenMP, incluyendo nuestra cláusula `speculative`. Esta propuesta libera al programador de modificar e instrumentar manualmente el código fuente con esas construcciones OpenMP, lo que generalmente es una tarea tediosa y propensa a errores.

Como resultado, el bucle instrumentado es ejecutado correctamente en paralelo, con un rendimiento que dependerá del número que violaciones de dependencia que se produzcan en tiempo de ejecución. Esta solución también puede ser utilizada para instrumentar un bucle paralelo con OpenMP estándar.

Las conclusiones de esta contribución serán enviadas para su publicación durante el año 2014.

R.3 Conclusiones

Esta Tesis ha respondido a las preguntas de investigación planteadas inicialmente. Las dos primeras cuestiones se han probado mediante un experimento en el que los bucles *FOR* de las aplicaciones en C de la suite de benchmarks SPEC CPU2006 han sido caracterizados y clasificados. Se ha obtenido que, en promedio, el 37,7% de los bucles *FOR* son potencialmente paralelizables mediante técnicas especulativas. Además, se ha obtenido que esos bucles representan el 28,34% del tiempo de ejecución de los benchmarks, siendo este porcentaje un valor significativo. La extracción de estos dos números demuestra que (1) es posible detectar nichos susceptibles de ser paralelizados especulativamente, y (2) es posible evaluar su impacto en términos de tiempo de ejecución. Determinar si un bucle es potencialmente paralelizable es una tarea compleja debido al gran número de variables que están involucradas. Esto demuestra el valor que tiene un sistema como el desarrollado, que permite realizar esta tarea automáticamente.

Para responder a la tercera pregunta, hemos desarrollado un sistema en tiempo de compilación que recibe el nombre de ATLaS. Este sistema demuestra que es posible transformar automáticamente un código secuencial en uno especulativamente paralelo. Esta transformación ha sido realizado en varios pasos. Primero, mediante la caracterización de los bucles *FOR*, BFCA añade automáticamente en el código las directivas OpenMP necesarias para paralelizarlo, incluyendo nuestra cláusula propuesta, `speculative`. Esta cláusula es entonces analizada por ATLaS para generar todo el código necesario para gestionar la ejecución especulativamente paralelo del código. Este proceso no requiere la intervención manual del programador, salvo la mera selección del bucle a ser paralelizado. Por lo tanto, la transformación de una versión secuencial del código en una versión paralela se lleva a cabo de manera automática, incluyendo la instrumentación del código con las construcciones OpenMP, y la generación del código adicional para gestionar la paralelización especulativa.

Esta Tesis no cubre ciertos cuestiones que nos gustaría abordar, y que definen el trabajo futuro para esta investigación. En primer lugar, es necesaria definir una serie de heurísticas que permitan escoger automáticamente los bucles a paralelizar. Con estas heurísticas, los programadores podrían evitar decidir qué bucle es más idóneo o más beneficiosa es su paralelización en términos de tiempo de ejecución. En segunda lugar, la solución propuesta por esta Tesis está enfocada en bucles *FOR*. Como trabajo futuro queda actualizar esta solución para que sea capaz de detectar, analizar, y paralelizar cualquier región del código fuente que sea susceptible de ser paralelizada. Por otro lado, sería conveniente el desarrollado de un compilador fuente-a-fuente para analizar sintácticamente el código y generar directamente una representación XML, evitando la dependencia con sistemas de terceros como Cetus. Finalmente, con to-

dos los puntos anteriores resueltos, el siguiente paso sería el desarrollo de un compilador que automáticamente paralelizara bucles o cualquier región del código fuente, que sería escogida siguiente las heurísticas propuestas. Este compilador generaría el código binario ejecutable, que dependiendo de la región paralela sería ejecutado en paralelo especulativamente, o usando OpenMP estándar.

R.4 Agradecimientos

Esta Tesis está parcialmente financiado por: Junta de Castilla y León (VA172A12-2, PIRTU), Ministerio de Industria (CENIT OCEANLIDER), Ministerio de Ciencia e Innovación y el Fondo Europeo de Desarrollo Regional (MOGECOPP project TIN2011-25639, CAPAP-H3 network TIN2010-12011-E, CAPAP-H4 network TIN2011-15734-E), y el proyecto HPC-EUROPA2 (número de proyecto: 228398, financiado por la Comisión Europea dentro del VII Programa Marco de Investigación y Desarrollo)

Chapter 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 knowledges required to parallelize source codes, 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, the search of new techniques that obtain a profit from potential parallel capabilities of current hardware and CMP (Chips MultiProcessor) architectures is still necessary.

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 want 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 Ph.D. thesis arises in this context. First, with a framework proposal that automatically extracts such information from source codes; and second, with a compile-time system proposal that automatically adds all the extra code lines to handle the speculative execution of a program.

1.1 Motivation

Most of the parallel programming models require 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* [96]. 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 (that is, 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]. 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 *dependency 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 Ph.D. thesis we will address the problem of obtaining the characterization and coverage of target loops automatically. By using an XML-based representation of the source code, we will determine the usage of the variables in the context of loops, and classify them in three categories: *private*, *shared*, and a third one, called *speculative*, which identifies variables that may lead to dependency violations. Besides this classification, our proposal also extracts other meaningful information from the code, such as pointer usage, memory management, I/O activity or usage of static variables inside loops. This information is relevant from the point of view of a programmer that wants to parallelize these loops.

Since executing a loop in parallel usually implies an overhead to start up the mechanisms of parallelization, execution times of loops are very important to determine whether it is worthwhile to parallelize them. A short loop (in execution time) is not worth being parallelized. It will probably take less time to execute it sequentially, because thread-management overheads associated to parallelization may ruin potential improvements of a parallel execution. However, a long loop could recover this

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

overhead when it is executed in parallel. Therefore, loops must be correlated with their execution profile to determine the relative amount of time that is spent in them. Otherwise, if all of these loops are located in cold code, the compiler could spend a great deal of effort speculatively parallelizing cold loop nests which would not result in any performance improvement. Our proposal collects the profiling information in order to better characterize the loops, and properly identify which of them are more profitable to be parallelized.

Once a loop is identified as suitable to be parallelized, programmers need to manually parallelize it using speculative techniques. However, this task is usually tedious and mistakes are easily committed. In this Ph.D. thesis, we address this problem proposing a prototype to automatically perform all the transformation needed in the source code in order to speculatively parallelize an application. Our final goal is to generate speculative parallel source code automatically, being the resultant code functionally equivalent to the original sequential code. In order to achieve this goal, we will implement these transformations into a mainstream compiler such as GCC.

1.2 Objectives of the dissertation

According to the identified problems described in the previous section, the research question to be solved in this Ph.D thesis is the following:

> *Is it possible to develop a compile-time mechanism able to (1) detect susceptible niches for speculative parallelization, (2) evaluate their impact in terms of parallel execution time, and (3) automatically transform such sequential source code into a parallel speculative version?*

1.2.1 Milestones

In order to answer this research question, we need to accomplish some intermediate, more specific objectives:

Goal 1: Evaluation of compilers parallelization capabilities

> *To assess the limitations of the automatic parallelization approaches provided by commercial compilers.*

Speculative parallelization is an alternative to automatic parallelization performed by compilers. Therefore, it is necessary to demonstrate why it is so important to study this alternative instead of simply settling for a compiler-guided automatic parallelization. In order to provide numerical data about the performance of automatic

parallelization performed by compiler, it is necessary to carry out an analysis of the parallelization capabilities of some commercial and well-known compilers, such as the Intel C/C++/Fortran compiler, Sun Studio compiler, and GCC.

Goal 2: OpenMP `speculative` clause proposal and definition

> *To propose and define a new OpenMP clause to point out the variables that could lead to a dependency violation.*

This kind of variables can be called *speculative*. This Ph.D. thesis takes as its reference the software-based speculative parallelization runtime system developed by Cintra and Llanos [36, 37, 110]. This system classifies the variables in terms of their accesses, and uses OpenMP to handle the execution of the threads. Until then, these variables are classified using the *private* and *shared* OpenMP clauses. These clauses do not distinguish the kind of variables that may lead to a dependency violation, which are precisely the kind of variables that speculative parallelization deals with. Operating in this way requires the manual intervention of programmers to point out the statements that contains accesses (writings or readings) to those variables, and to modify these statements to add the necessary code to handle the speculative execution controlling the possible dependency violations. This work can be avoided by labeling these variables with a new specific OpenMP clause, letting a compile-time system to process them and to automatically add the extra code necessary to handle the speculative execution.

Goal 3: OpenMP `speculative` clause design, implementation and evaluation

> *To design and implement the new OpenMP clause into a mainstream compiler, as well as evaluating the proposed clause with real-world applications, comparing the performance of the generated code with respect to the speculative execution of manually-instrumented versions of the same benchmarks.*

Due to its open-source character, the strong community behind it, and above all, the plugin mechanism to add new functionalities, GCC [72] is the compiler chosen to implement our OpenMP extension. GCC plugins [71] provide extra features to the compiler, enabling to modify GCC by adding, replacing, monitoring, or even removing passes from the compiler, without touching the GCC source code. Using the plugin mechanism, we will develop a compile-time system that automatically adds the code needed to handle the speculatively parallel execution of a loop, and uses the proposed OpenMP clause to manage the *speculative* variables. The additional code

inserted in this way will be needed by the runtime speculative system. The variable classification that this system needs is provided by Goal 4.

With the new OpenMP `speculative` clause, programmers will be able not only to parallelize some applications that are not parallelized at compile-time by conventional automatic schemes, but also they will be able to parallelize those applications without all the hurdles involved in the manual-speculative parallelization. In order to evaluate the capabilities of the proposed clause and the compile-time system, which perform all the changes needed in a source code to parallelize it speculatively, we will test them using synthetic benchmarks, and also some real-world applications. Besides of observing the speedups that our proposal can achieve, these experiments will also allow us to compare the performance of the automatic-generated code in relation with the same applications modified by hand to use the same speculative runtime system.

Goal 4: Speculative parallelization niches detection and variable classification

> *To merge static analysis and profiling information to extract some features of source code, and detect and quantify the hurdles that may affect the parallelization.*

Parallelization 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. The combination of static analysis of source code with profiling data will provide useful information regarding all loops in the target application, including loop coverage, loop suitability for parallelization, a classification of all variables inside loops based on their accesses, and other hurdles that restrict parallelization. This information allows analyzing how particular language constructs are used in real-world applications, and helps programmers to find candidate loops to be parallelized. Moreover, this information includes a classification of all variables used inside each loop.

Goal 5: Automatic synthesis of speculative code

> *To generate an OpenMP-based version of the source code analyzed that uses the new* `speculative` *clause to speculatively parallelize a target loop.*

The classification of all variables used inside each loop can be used to automatically generate an OpenMP-based parallel version of the loop, using the *shared*, *private* and the proposed *speculative* clause to declare the uses of each variable inside the loop. As a result, the target loop will be guaranteed to correctly run in parallel, with a

parallel performance that will depend on the actual number of dependency violations that will arise at runtime.

1.3 Research methodology

In order to accomplish the objectives proposed in this Ph.D. thesis, we have followed a research methodology defined by the research method for software engineering described in [2]. This method establishes four phases that the research process has to follow. This phases can be cyclically repeated with the aim of refining the solutions proposed.

1. *Observe existing solutions.* This phase has the purpose of detecting the problems that will be addressed during the research process, starting with the existing solutions. A complete study of the literature should be carried out to find works related with our research. This study is split into the different *state-of-the-art* sections presented in each chapter of this document.

2. *Propose better solutions.* In this phase, a solution is proposed a solution to overcome the limitations found in the previous step. As we will see, there is a lack of works proposing systems that merge automatic parallelization with thread level speculation, i.e. solutions that require no intervention of the programmers to add the necessary extra code to handle the speculative execution. Moreover, the software-based speculative runtime system used in this research requires OpenMP clauses to classify variables, and hence, it seems necessary to add a new clause to point out those variables that may lead to dependency violations. Until now, OpenMP classifies them simply as *shared*, forcing programmers to manually add the code that handle such situations. This work can be avoided by defining a new *speculative* category, and its associated clause. Doing so, a compile-time system can process it automatically adding the extra code necessary to handle the speculative execution. The information needed to classify variables according to their accesses is also a need that this dissertation will cover.

3. *Build or develop the solution.* The solution proposed in the previous phase is implemented in this step. We will develop a prototype of a compile-time system that covers the needs found in the previous phase.

4. *Measure and analyze the new solution.* Finally, the engineering method establishes that the proposed solution has to solve the problems discovered in the first phase of the research methodology. We will use regression tests during the

development, in order to validate the correction of the system, whereas we will evaluate the system using synthetic benchmarks and some real-world applications.

1.4 Document structure

This document is structured as follows. Chapter 2 uncovers the limitations of the automatic parallelization approaches, which are not able to parallelize many codes on the basis of a compile-time analysis. This study provides a result that lead us to other alternatives, such as speculative parallelism. Chapter 3 defines a new OpenMP clause, with the aim of classifying the variables that may lead to dependency violations. This clause provides an opportunity to automatize the speculative parallelization of source code, which will be implemented in Chap. 4. Chapter 4 also evaluates the capabilities of the proposed compile-time system, measuring the speedup achieved by parallelizing some real-world applications, and also comparing the performance obtained by the automatic parallelization of those codes in relation with the manually-modified versions of the same codes that take advantage of the same runtime parallelization system. Chapter. 5 proposes a solution to extract all the information required in the process of parallelizing source codes, helping not only with the decision about which loop is more profitable to be parallelized, but also providing meaningful information in order to parallelize a loop properly. This information is very valuable not only for speculative schemes, but also with other kind of parallelization techniques. Finally, Chap. 6 proposes a system that uses the reports generated by the solution proposed in the previous chapter to automatically generate an OpenMP-based version of the analyzed source code that uses the proposed OpenMP clause. As a result, the source code is parallelized speculatively.

To sum up, Chap. 7 covers the contributions of this Ph.D. thesis, as well as the conclusions, potential future work, and improvements over the developed system. This chapter also aims to answer our research question, and collects the different publications associated to each partial goal.

Finally, several appendices are included. Appendix A covers some extended details related with the technologies involved in the development of the prototypes proposed. Appendix B describes in-depth and helpful details of GCC for the development of plugins. Appendix C contains installation and user's manuals for the prototypes developed and presented in this document. and finally, App. D describes the content of the digital support attached to this document.

Chapter 2

Evaluation of Compilers Parallelization Capabilities

As a runtime technique, it is expected that speculative parallelization extract more parallelism than pure compile-time techniques, such as the ones implemented in parallelizing compilers. To measure how effective are current compilers in parallelizing source code, in this chapter we will carry out an analysis of the parallelization capabilities of some commercial and well-known compilers, such as the Intel C/C++/Fortran compiler, Sun Studio compiler, and GCC. These capabilities will be measured in terms of the parallel performance of the code generated. Since each compiler has its own characteristics, and applies its own optimizations, a study of the sequential performance will be also carried out, with the aim of determining if there is a correlation between both sequential and parallel performances.

2.1 Problem description

Compilers are a critical part of any computing environment, allowing programmers to better exploit the hardware capabilities of their systems, minimizing the sequential execution time and, in some cases, offering the possibility of parallelizing part of the code automatically. However, in many cases this approach is limited mainly due to two reasons: Insufficient performance improvement, and inability to parallelize the code effectively. In the first case, although the compiler is capable of parallelizing the code, the resulting program may not produce noticeable speedups. In the second case, compilers perform static analysis to determine whether a code is parallelizable, being rather cautious in the decision. A compiler usually does not parallelize part of a code

if it is not completely sure that this part is parallelizable. Since correctness is their main concern, compilers do not assume risks.

As a result, many codes that have an inherent parallelism are not parallelized by compilers, losing the opportunities that current multicore architectures offer. In order to effectively measure the limitations of automatic parallelization techniques, this chapter relies on the SPEC CPU2006 v1.1 benchmark suite to evaluate the parallel performance of the code generated by three widely-used compilers (Intel C++/Fortran Compiler, Sun Studio, and GCC). As we will see, this evaluation will give support to our aim of improving and making more accessible other alternatives such as the speculative parallelization, which may obtain better performances in some codes, or even parallelize codes that are impossible of parallelizing by others means.

The rest of the chapter is structured as follows. Section 2.2 introduces the concept of benchmarking and gathers information on the particular benchmarks suite that we have used in the study: SPEC CPU2006. Section 2.3 is split into two parts. The first part discusses in detail the performance of the sequential code generated by the compiler suites using the two benchmark sets that are part of the SPEC CPU2006 suite: CINT2006 and CFP2006, whereas the second one describes the effect in terms of performance of the auto-parallelization flags available in Sun and Intel compilers. Finally, Sect. 2.4 summarizes the results, providing overall ratings and describing the main conclusions of the study. Moreover, it summarizes our contributions to the literature.

2.2 State of the art

In order to evaluate the performance of a compiler it is not appropriate to use arbitrary source codes. This section shows the reason to use benchmarks in order to obtain meaningful results that are reproducible by other research groups. Besides, benchmarks are the perfect choice to compare performances between compilers, and the SPEC CPU2006 benchmarks suite is one of the most widespread alternatives used by researchers in the field of high performance computing.

2.2.1 Benchmarking Overview

Benchmarking is a concept transversal to many fields, and particularly to business. One of the main concerns of businesses, no matter the sector in which they are, is the incessant evolution in order to improve themselves and overcome their competence. In this race, it is important the development of increasingly efficient products, with higher quality and cheaper costs. Businesses should not only overcome rivals and win the battle of market, but also learn from them their mistakes and successes, to see

which strategies are used to reach those results.

In business terms, benchmarking allows measuring differences between enterprises, to observe results from the strongest competitor, and to understand the strategies that this competitor has followed to reach its position. Thus, benchmarking is not only a measuring instrument for a particular product or set of products, but also and more important, it is a tool that allows comparing some products with others, observing improvements that the business can adopt, and integrating strategies that lead to the development of higher-quality and higher-performance products. Thereby benchmarking is observed as a learning process, and in this Ph.D. thesis it is used to evaluate compiler's performance, to compare between them, and finally, to learn about their parallelization capabilities. This will lead us to seek other alternatives such as speculative parallelization.

Before that, in order to understand the importance of benchmarking it is useful to know its history.

History of benchmarking

In 1979 [24, 25], Xerox was in a critical situation. It was losing its position in a market created by itself and dominated along the 60's with more than the 80% world participation. This situation was due to the good work of its Japanese competitors. In 1979, companies as Canon, Minolta or Ricoh began to sell their photocopiers in the USA, with lower prices than Xerox's. Japanese's photocopiers had a sale price equal to the production costs of Xerox's photocopiers.

With the purpose of recovering its leading position, Xerox sent a crew to Fuji-Xerox, its Japanese affiliate, to study in detail processes and materials used in that country. Xerox was looking for a reference (a benchmark) to change its strategy and understand what it could do better. The result of this study was discouraging. Results uncovered the reason for such a big difference in prices:

- Xerox had nine times more suppliers and twice more employees.
- Delivery times of final products was doubled.
- Production lines had ten times more defective components.
- Final products had seven times more manufacturing defects.

At sight of these results, Xerox concluded that Japanese competitiveness was not due to a cheaper labor, neither government subsidies, but details in the manufacturing process. This first study performed by Xerox was the first benchmarking. As result, Xerox's products improved in quality (from 91 defects in 100 machines to 14 defects), production costs (reduced by 50%), and development time (reduced by 60%).

In 1981, in evidence of these improvements, Xerox's directives ordered to adopt benchmarking studies in all their business, and they embraced a new philosophy: To reach higher quality in their products and processes, both employees and benchmarking are essential. As a consequence, annual increases in productivity went from 2-3% to over 10%. These results led other competitors to accept this practice along the 80's, being later adopted in other sectors.

Before 1981, most of the studies were performed using comparisons between products of the same company. Benchmarking changed how companies were working, and it led to a new way of working in which products and processes from other companies were the focus point to grow up as business.

Formal definition of benchmarking

Despite the work of many experts in the field, there does not exist an unique definition of benchmarking. From the business context we can obtain two classical definitions. Robert C. Camp, benchmarking pioneer, introducer of this technique in Xerox, and now the president of Global Benchmarking Network, defines benchmarking as "*the activity of learning, exchanging, and adapting best practices to your organization*" [26]. Michael J. Spendolini, a reputed expert in benchmarking, slightly extends Camp's definition: "*Benchmarking is a continuous and systematic process to evaluate products, services and processes against competitors, or renowned organizations considered world leaders in their field*" [148].

Both definitions are related to the search for industrial best practices that lead to superior performance. These definitions are straightforwardly applicable to business, but benchmarking is also very important in computing. From the computing point of view, benchmarking is related with the execution of a computer program, or set of programs, with the purpose of measuring the performance of an entire computer, some hardware components, or another software programs as compilers or database management systems. A benchmark provides a way to compare the performance of different subsystems between them and it frequently allows a rapid technical progress and a community building [147] since each group has a reference to compare its developments.

Types of benchmarks in computing

Once we have seen two definitions of benchmarking, we will focus on the computing field. Computing benchmarks can be classified in two ways [14]: high-level vs. low-level benchmarks, and synthetic vs. application benchmarks.

High-level vs. low-level benchmarks

Depending on the level of the measured components, benchmarks are high-level or low-level. Low-level benchmarks intend to measure the performance of the individual hardware components: CPU clock, memory and cache cycle times, hard disk average access time, etc. These tests can be useful to check whether the performance of a particular component is accordant with the expected performance, or simply to know characteristics of an unknown system.

High-level benchmarks measure the performance of the hardware/driver/OS combination for a specific aspect of a system, e.g. file I/O performance, or for a specific hardware/driver/OS/application performance, e.g. a benchmark to measure the performance of the Apache Web Server on different systems.

Due to their characteristics, low-level benchmarks are all synthetic benchmarks, while high-level benchmarks may be synthetic or application benchmarks.

Synthetic vs. application benchmarks

Benchmark can also be classified as synthetic or application benchmarks, depending on their operation and design benchmarks. Synthetic benchmarks are designed to measure the performance of individual components, generally taking them to their maximum capacity. However, results of these benchmarks are controversial, since they do not reflect the performance of the components in real-life situations. Synthetic benchmarks use instructions typical of real programs, and measure the speed in which they are processed. These benchmarks are a kind of "real-life application simulators", and they could not represent the real performance of the component that they measure.

On the other hand, application benchmarks or "real-world benchmarks" are formed by or are based on real applications, which simulate real-situation workloads to measure the performance of the whole system, or only a subsystem. This is the case of the SPEC CPU benchmarks suites used in this document, and the reason to choose them. These benchmarks provide a more reliable measurement of the performance of a system than synthetic benchmarks, since the latter only choose a set of representative instructions and their results must be interpreted under these conditions.

SPEC CPU Benchmarks suites

The SPEC CPU2006 benchmark suite will be used in this chapter to check the parallelization capabilities of commercial compilers. Moreover, these benchmarks will be also used in following chapters to obtain different results, including the performances of the prototypes proposed in this Ph.D. thesis. Testing with synthetic applications could be simpler, but not realistic. With the aim of obtaining results in which we can

trust, it is necessary to use "real" applications, such as the ones included in the SPEC CPU benchmarks suites.

SPEC (Standard Performance Evaluation Corporation) [150] is a non-profit consortium formed by several software companies, hardware manufacturers, etc. This consortium was founded in 1988 by a small group of computer sellers with the aim of obtaining a performance standard test that would offer some realistic and comparable results. Over time, SPEC has grown to become one of the more successful performance standardization bodies, with more than 60 member companies.

SPEC seeks to simulate real situations, and because of that, it gets real-life applications from various fields of science and engineering (maths, physics, chemistry, etc.), using them as benchmarks with different workloads in order to obtain performance evaluations. The first benchmark suite was released by SPEC in October 1989 [135], called "*SPEC Benchmark Suite for Unix Systems version 1.0*". This suite was formed by just ten benchmarks, and it allowed to produce three metrics: "*Integer SPECmark*" for the four integer benchmarks, "*floating-point SPECmark*" for the six floating-point benchmarks, and "*overall SPECmark*", which was a result that includes the execution of all the benchmarks without type distinctions. In December 1991 [136], SPEC renamed its integer and floating point benchmark metrics as "*SPECint*" and "*SPECfp*". The following month, SPEC released "*SPECint92*" and "*SPECfp92*", and hence the previous versions were renamed as "*SPECint89*" and "*SPECfp89*". However, this new version did not allow getting an overall result of both metrics.

SPEC updates periodically the suite due to the evolution of computers and the growth and increasing complexity of application programs. In this way, SPEC released new versions in 1995, 2000, and 2006, each one with more applications, more complex codes, and larger workload sets. Following subsections slightly describe the last two versions used in this Ph.D. thesis: SPEC CPU2006 and SPEC CPU2000.

SPEC CPU2006

SPEC CPU2006 is the last benchmark suite developed by SPEC, whose first version was released on August 24, 2006. Current version of SPEC CPU2006 is 1.2 [51], released in September 2011. The reason of this update is the rapid evolution of the computers performance in the first decade of the 21st century. In summer of 2006, many of the CPU2000 benchmarks executed in contemporary computers ended their executions in less than a minute. Hence, little changes in the system state or measurement conditions could affect final results significantly. Other factors also influenced in this update. Programs became more complex and larger since the release of CPU2000. Therefore, in SPEC CPU2006, the consortium decided to update the benchmarks included in the suite for new versions of the same programs, and also new relevant

programs of other fields, as voice recognition or video compression.

SPEC CPU2006 includes 29 benchmarks, whose descriptions can be found in [82]. They are encompassed in two groups: "*CINT2006*", which contains applications that only work with integer numbers, and "*CFP2006*", which contains applications that work with floating-point numbers. SPEC supplies the benchmarks in the form of source code, which testers are not allowed to modify except under very restricted circumstances.

To execute each benchmark, SPEC CPU2006 has three different workload sets, ranked in order of increasing workload:

- **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.
- **Reference**, which is the workload set used to obtain execution times and hence, the final performance results.

SPEC establishes strict rules to run the benchmarks and to report the results, with the aim of ensuring that the observed level of performance can be obtained by other researchers. The benchmark also includes a tool to run and score benchmarks automatically [149].

For CPU2006, SPEC defines two type of metrics. *Base* metrics are reproduced compiling and running the entire benchmark using the same compiler flags in the same order for a given language. These metrics are required for all reported results, and provide a consistent baseline for comparing performance. The *peak* metrics are optional and have less strict requirements, allowing to obtain better results using a tailored set of flags for each benchmark, while the *base* metrics have stricter guidelines for compilation.

SPEC uses a reference machine to normalize the performance metrics used in the CPU2006 suites. Each benchmark was run and evaluated on this system to establish a reference time for that benchmark. These times are then used in the SPEC calculations. SPEC uses a Sun system, the "Ultra Enterprise 2" introduced in 1997, as the reference machine. This system has a 296 MHz UltraSPARC II processor with two cores and two GB RAM. Table 2.1 summarizes its characteristics.

After running the benchmark, the SPEC CPU2006 suite generates a report with the relative performance of the System-Under-Test (SUT) compared with the reference machine. To consider a report valid, it should be generated executing each benchmark three times with the test and train workloads and then three times with the reference workload. The execution times of the latter provides the final results. After the entire benchmark suite is run on the SUT, a ratio for each benchmark is calculated using

CPU model	UltraSPARC II processor
CPU Characteristics	296 Mhz, 2 cores, 1 chip
L1 Cache Size	16 KB I + 16 KB D
L2 Cache Size	2 MB I+D
RAM Memory	2 GB
Disk Subsystem	2x36 GB 10000 RPM SCSI
Operating System	Solaris 10 3/05
Kernel Version	2.6.19
File System Type	ufs
Compiler	Sun Studio 11

Table 2.1: Characteristics of SPEC CPU2006 reference system. According to the specifications, one of the disks is dedicated to the operating system and the other to the code and data set of the benchmark suite.

the wall-clock time spent on the SUT and the time spent by the reference system, as provided by the suite in [126]. From these ratios, the suite calculates the geometric mean of 12 normalized ratios, one for each integer benchmark, and the geometric mean of 17 normalized ratios, one for each floating-point benchmark.

SPEC CPU2000

The SPEC CPU2000 benchmark suite was released at December 30, 1999, and it was updated until the version 1.3, released at November 1, 2005 [50], almost 6 years later. Finally, CPU2000 was officially retired in February 2007 [49]. Despite CPU2000 is not the last version released by SPEC, at present it is widely used by researchers because CPU2006 is complex and tricky.

SPEC CPU2000 includes 26 benchmarks [81] that are divided in two groups, "*CINT2000*" and "*CFP2000*", following the same criteria than in SPEC CPU2006. CPU2000 also defines three workload sets to execute its benchmarks: *test*, *train*, and *reference*, ranked in order of increasing workload. Execution rules applied in CPU2006 are also applied here, and CPU2000 defines the same two type of metrics: *base* and *peak*. Overall, most of what was said for the CPU2006 benchmark suite applies here. The reference machine used in SPEC CPU2000 is even the same than the machine used in SPEC CPU2006, but with less memory (256 MB) and slower caches.

2.2.2 Automatic parallelization and SPEC CPU2006

Surprisingly, there is a lack of comparisons about the automatic parallelization capabilities of modern compilers, using either SPEC CPU2006 or other benchmarks. However, some research has been carried out on the characterization of SPEC CPU2006

benchmark on sequential architectures. Li *et al.* [107] characterize the performance of SPEC CPU2006 both on Intel and AMD architectures, using GCC 4.1.2 with the `-O3` flag. Their study includes an evaluation of the level of instruction-level parallelism found. Kejariwal *et al.* [98] compares the behaviour of both SPEC CPU2000 and SPEC CPU2006 on an Intel platform, using the Intel Optimizing Compiler, giving some hints that may guide the design of future microprocessors. Ye *et al.* [172] characterize the performance differences between 32- and 64-bit versions of the SPEC CINT2006 on an Intel x86-64 platform, using GCC 4.1.1. Their work shows that several X86 integer applications runs slower in 64-bit mode than in 32-bit mode, a result consistent with our study, as we will see in the following section. This explains the effect based on the different instruction and data cache lines requested when using these different address spaces.

2.3 Evaluation of the parallelization capabilities of commercial compilers

Speculative parallelization is an alternative to automatic parallelization performed by compilers. Therefore, it is necessary to demonstrate why it is so important to study this alternative instead of simply relying on automatic parallelization.

This section aims to measure how much performance can be obtained by automatic parallelization, evaluating three popular widespread compilers for Intel-based architectures: Intel C++/Fortran Optimizing Compiler version 11.0.074 [88, 145], Sun Studio 12 [155] and GCC 4.3.2 [73]. Parallelization capabilities of these compilers will be measured in terms of the parallel performance of the code generated. Since each compiler has its own characteristics, and applies its own optimizations, we will also perform a study of the sequential performance of the code generated, with the aim of determining if there is a correlation between both sequential and parallel performances. In this way, we can find if the parallelization capabilities of a compiler are related to the sequential performance that the same compiler is able to extract from a source code.

Therefore, both sequential and parallel performance have been evaluated. The evaluation has been made compiling and running the SPEC CPU2006 v1.1 suite on both 32-bits and 64-bits systems, comparing the performance of these compilers with different benchmarks and hardware configurations. The study also includes a detailed description of the different problems that arise while compiling SPEC CPU2006 benchmarks with these tools, an information difficult to obtain elsewhere.

To sum up, the goal of this study is threefold: (1) To provide results that lead to a better understanding of compiler technology and use, (2) to give an insight into SPEC CPU2006 benchmark suite, describing the main problems encountered while

CPU model	Intel® Core™ 2 CPU E6300
CPU Characteristics	1.86 Ghz, 1066 MHz bus. 2 cores, 1 chip
L1 Cache Size	32 KB I + 32 KB D on chip per core
L2 Cache Size	2 MB I+D on chip per core
RAM Memory	3 GB (2x512 MB + 2x1 GB DDR2 667 MHz)
Operating System	Mandriva Linux Release 2007.1, 32 bits (`SUT1`)
	Mandriva Linux Release 2007.1, 64 bits (`SUT2`)
Kernel Version	2.6.17-13
File System Type	ext3.
Compilers	GCC 4.3.2
	Intel C++/Fortran 11.0.074 Professional Edition
	Sun Studio 12 (Sun C/C++ 5.9, Sun Fortran 95 8.3)

Table 2.2: Characteristics of the Systems-Under-Test SUT1 and SUT2.

CPU model	Dual Core AMD Opteron® Processor 270
CPU characteristics	1.93 Ghz, 1066 MHz bus. 4 cores, 2 chips
L1 Cache Size	64 KB I + 64 KB D on chip per core
L2 Cache Size	1 MB I+D on chip per core
RAM Memory	4 GB
Operating System	Gentoo Base System release 1.12.9
Kernel Version	2.6.19
File System Type	ext3
Compilers	GCC 4.3.2
	Intel® C++/Fortran 11.0.074 Professional Edition
	Sun Studio 12 (Sun C/C++ 5.9, Sun Fortran 95 8.3)

Table 2.3: Characteristics of the System-Under-Test SUT3.

using different compiler suites, and finally (3) to show the limit of the parallelization capabilities of these compilers that encourages us to seek other alternatives such as speculative parallelization.

Systems under test

To perform the study, two different hardware configurations were used, and for one of them it was used 32- and 64-bits versions of the same operating system. This leads to three different Systems-Under-Test (SUT). The first System-Under-Test (SUT1) is based on an Intel® Core™ 2 CPU E6300 processor, running a 32-bits version of the GNU-Linux operating system. SUT2 is based on the same hardware as SUT1, but it runs 64-bits version of GNU-Linux. Table 2.2 summarizes their characteristics. Finally, SUT3 is based on a Dual Core AMD Opteron® Processor 270 (see Table 2.3).

Described systems were evaluated with the entire SPEC CPU2006 v1.1 bench-

mark suite. The chosen performance metric was the *base speed* provided by the benchmark. A base metric was chosen instead of a peak metric since the former has more strict guidelines for compilation and forces to use the same compiler flags for all benchmarks, avoiding the use of tailored optimizations.

As it has been previously described in Sect. 2.2.1, SPEC CPU2006 provides three workload sets for each benchmark: *test*, *train* and *reference*. These sets are ranked in order of increasing workload. The first two are workloads that intend to check the correctness of the compilation and execution process, while the third one is used to evaluate performance. After running the benchmark, the SPEC CPU2006 suite generates a report with the relative performance of the SUT compared with the reference machine. Recall that, to consider a report valid, it should have been generated executing each benchmark three times with the test and train workloads and then three times with the reference workload. Execution times of the latter provides the final results. Each execution time is then compared with the execution time of the same benchmark in the reference system [126], defining a ratio. From these ratios, the suite calculates the geometric mean of 12 integer benchmark normalized ratios, called *SPECint2006*, and the geometric mean of 17 floating-point benchmark normalized ratios, called *SPECfp2006*.

Compiler flags

In order to compile and run the benchmark suite, each compiler needs some particular flags. Unfortunately, SPEC CPU2006 does not suggest a minimum set of flags. This makes the search for adequate flags a trial-and-error process. Moreover, as Chan *et al.* pointed out in 1994 [32], it is difficult to ensure that none of the flags chosen is offered by the compiler just to optimize some SPEC program, a situation not allowed by any released SPEC benchmark. Besides this, the *base speed* metric forces to use the same flags to compile the entire benchmark. After an extensive experimentation, the final flags used in the study are shown in Table 2.4.

We found that that many additional optimizations available did not work equally well or, even worse, could not be applied to all benchmarks. The particular reasons are described below. We found this information extremely useful for anyone interested in running these benchmarks.

GCC compiler, 410.bwaves, 32-bits versions (SUT1) In this system, when executing 410.bwaves compiled with the `-O3` flag, the "train" input set leads to incorrect results. `-O2` flag should be used instead.

GCC compiler, 64-bits version (SUT2 and SUT3) The `-O3` flag includes by default `-finline-functions`, which expands functions during compilation. This flag

GCC	-O3 -funroll-loops -fno-inline-functions ftree-vectorize
INTEL	**sequential 32-bit flags:** -O3 -ipo -xT -axT -no-prec-div -funroll-all-loops -no-for-main (C and Fortran at once) **sequential 64-bit flags:** -O3 -ipo -xW -axW -no-prec-div -funroll-all-loops -no-for-main (C and Fortran at once) **parallel flags added:** -parallel
SUN	**sequential 32-bit flags:** -fast -xarch=sse3 -library=stlport4 (C++ Benchmarks except 453.povray) **sequential 64-bit flags:** -fast -xarch=sse3 -m64 -library=stlport4 (C++ Benchmarks except 453.povray) **parallel flags added:** -xautopar -xreduction

Table 2.4: Compiler and linker flags used to obtain the SPEC CPU2006 *base speed* metric used in this study. 64-bit binaries was generated for 64-bit SUTs.

makes some benchmarks fail in 64-bits systems. Therefore, we used the flag `-fno-inline-functions` to cancel the optimization.

GCC compiler, 400.perlbench The `-fno-string-aliasing` flag is needed to run this benchmark in all SUTs, so it should be included among the flags needed to obtain the *base speed* metric.

GCC compiler, auto-parallelization options The GCC compiler is unable to compile some benchmarks when adding the `-ftree -parallelizing-loops=n` flag. This fact makes impossible to compare the effect of this feature with the corresponding features of Sun and Intel compilers, since in order to run the benchmark all applications must compile and run correctly.

Intel compiler Different problems were detected in the compilation of all the benchmarks when the `-fast` flags is used. This is because the `-static` flag is automatically included when using `-fast`. We have replaced the use of `-fast` with the use of all flags it includes, except `-static`.

Intel compiler, mixed Fortran-C applications Intel Fortran compiler does not correctly compile benchmarks that are written with a Fortran-C combination, such as 435.gromacs, 436.cactusADM and 454.calculix. The flag `-nofor-main` for the Fortran compiler solves this problem. This option specifies that the main program is not written in Fortran, so it prevents the compiler to link `for_-main.o` in the applications.

Intel compiler, 64-bits systems For these systems the `-xT` flag generates specialized code, enabling vectorization. In particular, according to Intel Reference

Manual [88], `-xT` may generate SSE instructions for the Intel® Core™ 2 Duo Processor family. However, we have found that the use of `-xT` flag produces invalid executions. The flag `-xW`, that optimizes for Pentium™4 processor, was used instead. This problem does not happen with option `-axT`, that also generates non-processor specific code, although we change it to `-axW` to follow Intel recommendations.

Sun compiler, 32-bits versions (SUT1), 400.perlbench and 416.gamess To compile these benchmarks, the `-xautopar` and `-xreduction` should not be used. This only happens with the v1.1 version of the SPEC CPU2006 benchmark, while the v1.0 version runs perfectly well with these flags.

Sun compiler, 64-bits versions (SUT2 and SUT3), 400.perlbench To compile this benchmarks, the `-xautopar` and `-xreduction` should not be used. As the with preceding issue, this only happens with the v1.1 version of the SPEC CPU2006 benchmark, while the v1.0 version runs well with these flags.

Sun compiler, C++ benchmarks In these cases, SUT1 displays the following linker error with the 64-bits configuration:

```
/usr/lib64/libm.so: file not recognized: File format not recognized
```

This error is due to a problem with the Sun linker, that is not able to use the `libm` library in 64-bits operating systems. We simply replace Sun's linker with GNU's, through a symbolic link.

Sun compiler, C++ benchmarks It was necessary to use the STLport implementation of C++ standard library (`-library=stlport4`), instead of using the library by default (`libCstd`). This change solves compilation errors in C++ benchmarks. However, in the particular case of 453.povray, this option had to be removed, using the library by default, because the STLport library causes the following error:

```
octree.cpp, line 755: Error: The function copysign must have a prototype.
```

This change in the library used does not invalide the *base speed* metric obtained.

Sun compiler, auto-parallelization options When using these options with Sun compilers, it is necessary to set the `OMP_NUM_THREADS` environment variable to match the number of threads to use during the parallel execution, and the `PARALLEL` variable, to set the number of available processors. In our case, we set these variables to four.

410.bwaves, 483.xalancbmk, 447.dealIII There is a run-time issue to be considered when running these benchmarks. Due to a problem with the system stack size,

these benchmarks show the following error message:
`410.bwaves: copy #0 non-zero return code (rc=0, signal=11)`
The solution is to increase the stack size before running the benchmark suite, through the command `ulimit -s unlimited`.

2.3.1 Sequential performance

SPEC CPU2006 benchmarks are divided in two sets: integer benchmarks and floating-point benchmarks. Since differences between these two types of benchmarks are significant, we will analyze their results separately.

Sequential performance of integer benchmarks

To evaluate the relative performance of the code generated by each compiler suite, we ran the entire SPEC CPU2006 test with each compiler and each SUT considered. We start our study evaluating the relative performance of the SPEC CPU2006 integer applications. Figure 2.1 shows the performance of the code generated for each compiler suite for all three SUTs considered. Figure 2.1(a) shows the results for SUT1, in which Intel's performance is much higher than the performance offered by GCC and SUN. The reason is, in part, the availability of the autovectorization flag in Intel, which is capable of vectorizing five loops in the 462.libquantum benchmark, while GCC is only able to do it in a single loop. Sun performance is clearly lower than Intel's, both for the geometric mean and for most of the benchmarks executed.

Figure 2.1(b) shows the results for the 64-bits configuration (SUT2). In this case, Sun achieves a better performance, although the differences among the compilers are not so high.

The numerical differences in terms of performance are lower in SUT3 (64-bits), and the overall performance of the machine is similar for all three compiler suites (Fig. 2.1(c)). It is interesting to note the lower performance of 462.libquantum for the Intel compiler, while for SUT1 and SUT2 the same benchmark runs much faster. We have found no explanation for this effect, since we have used the same compilation flags as in SUT2. Finally, it is worth remarking the high performance obtained by the code generated with GCC in 464.h264ref benchmark, improving by 18% the results obtained using Intel compilers.

We conclude that, in the set of integer benchmarks of SPEC CPU2006, all three compilers perform equally well in the 64-bits environment with the optimization flags considered. Regarding the 32-bits environment, Intel shows an average improvement of 30.7% over the performance of the code generated by GCC compiler and 24.7% over Sun compiler. We remind that performance is a moving target in the field of

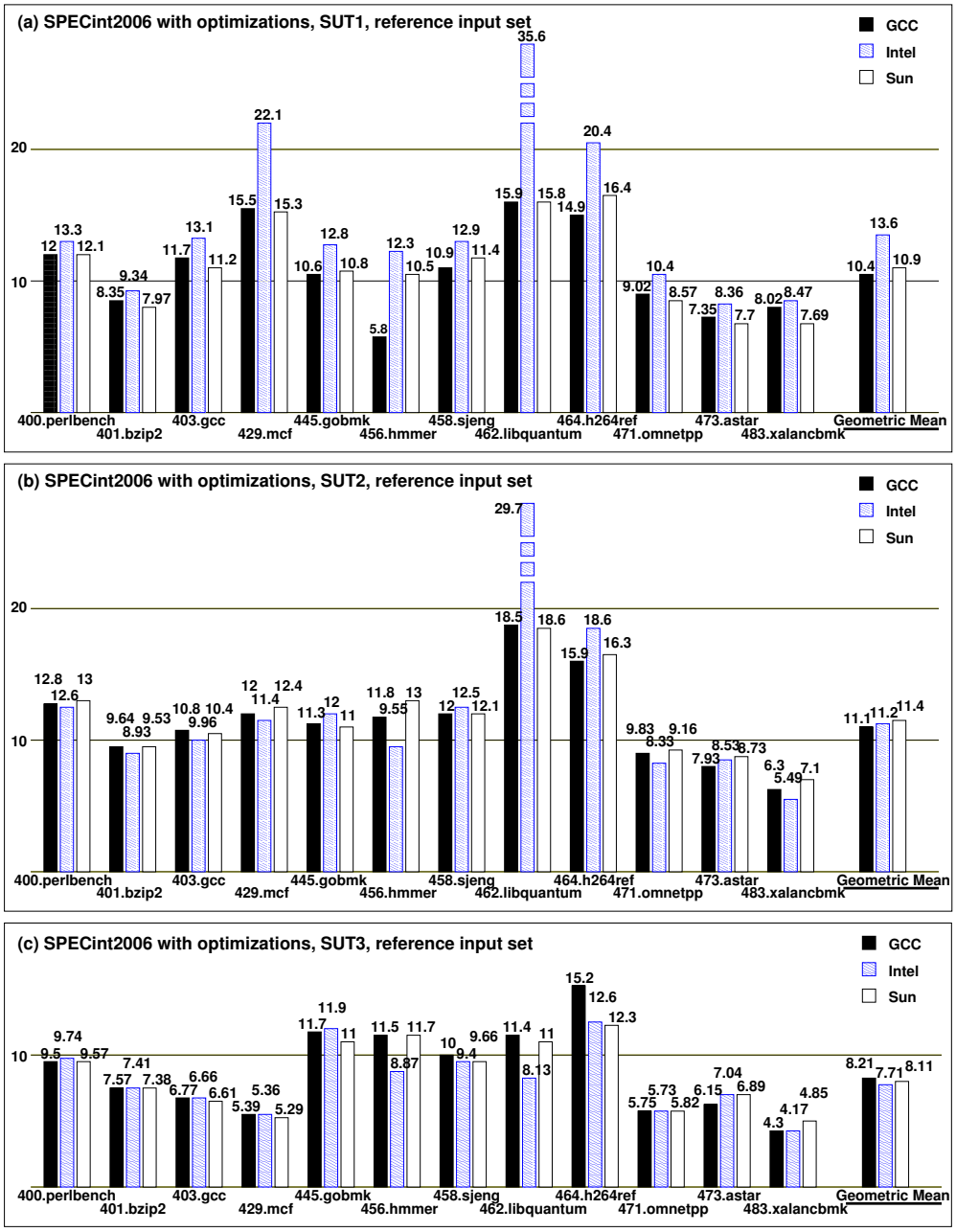

Figure 2.1: Sequential performance of SPEC CPU2006 INT benchmarks.

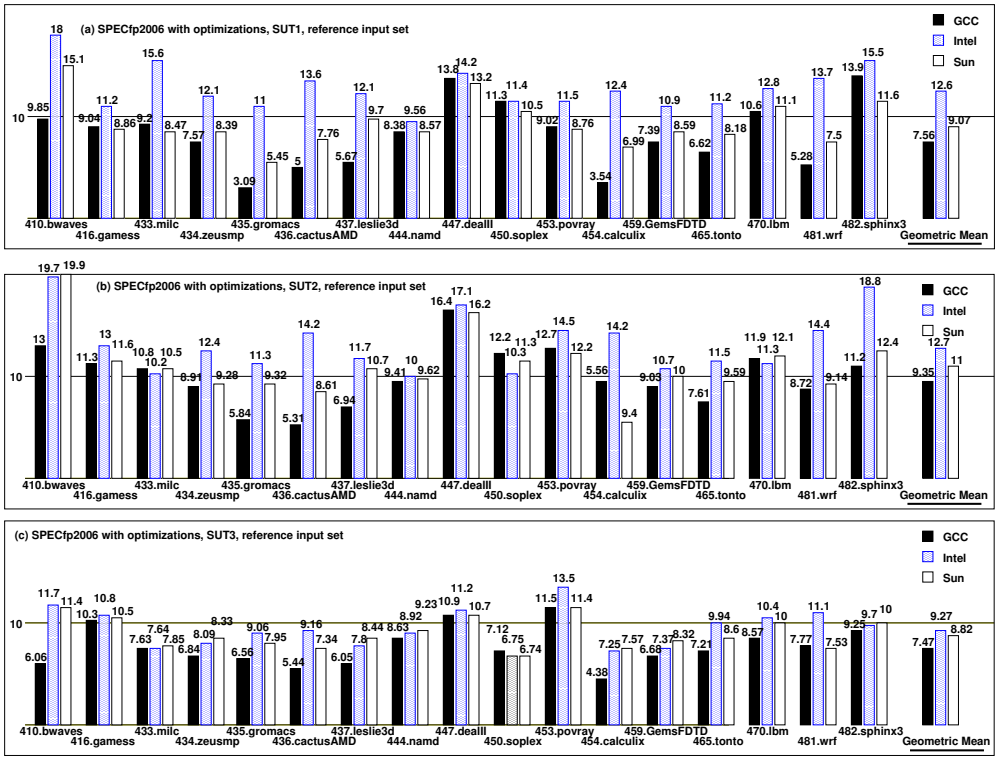

Figure 2.2: Sequential performance of SPEC CPU2006 FP benchmarks.

compilers, and these results may experience variations with other SUT configurations, compiler versions or different optimizations.

Sequential performance of floating-point benchmarks

Performance differences are bigger for SPEC CPU2006 floating-point benchmarks. Figure 2.2 shows the relative performance of the execution of all 17 floating-point applications using the different compiler suites. Figure 2.2(a) shows the results obtained in SUT1. For this environment, Intel gets the best results in every single benchmark, surpassing the Sun compiler for more than 101% in 435.gromacs, 84% in 433.milc, or 75% in 439.cactusAMD. On the other hand, GCC obtains the worst results in 11 benchmarks, with a global performance of 7.48 points, 38% lower than Intel and 18% lower than Sun compilers.

Figure 2.2(b) shows the results for SUT2, that is, the same hardware configuration than SUT1 but with a 64-bits operating system. Differences among compilers remain

quite large. Intel obtains the best performance on 14 out of 17 benchmarks, with an average performance that is 35.8% better than the performance obtained with GCC code and 15.4% better than Sun's.

If we compare the results for SUT1 and SUT2, we observe that all three compilers improve their results when running in the 64-bits environment, while the improvement is much smaller when dealing with integer-based applications. This result is consistent with the work by Ye *et al.* [172], that reports a performance gain of 7% on average when running in a 64-bit address space. In the floating-point case, Intel compilers achieve an average improvement of about 7%, Sun improves by 21% and GCC improves by 25%, although the performance of the latter is well below its competitors.

Finally, Fig. 2.2(c) shows the relative performance of SUT3, a 64-bits system (recall Table 2.3). Differences are not so high in this case, although Intel performs consistently better than Sun and GCC compilers, with better results in roughly half of the benchmarks. As it happened in other cases, with rare exceptions, the use of GCC compilers lead to worse results than the use of Intel and Sun compilers, although differences are minor: 20% of slowdown with respect to the code generated by Intel and 16% with respect to Sun's.

We conclude that, in the case of the SPEC CPU2006 floating-point applications, the use of the Intel compiler suite considered leads to better performance results with the optimization flags considered, particularly when the code runs on Intel-based architectures.

2.3.2 Parallel performance

The following lines summarize the performance of the parallel code generated by Intel and Sun compilers. Although GCC 4.3.2 offers the possibility of exploiting loop-based parallelism through the use of the `-ftree-parallelizing-loops` flag, its use leads to compilation errors in several of the benchmarks of the SPEC CPU2006 suite. The strict rules of use of the benchmark suite prevent us to compile and evaluate separately each benchmark, so the performance of parallel GCC applications will not be considered here.

Regarding integer benchmarks, the performance results of the parallel versions automatically generated by Intel and Sun compilers are pretty much the same that for their sequential counterparts. These results, that are not shown here, makes clear that the autoparallelization capabilities offered by Intel and Sun compilers are not enough to extract any parallelism of the considered integer benchmarks. These benchmarks are in fact hardly parallelizable even by hand, and the cost of thread management is usually higher than the benefits obtained.

While differences in integer-based benchmarks are minimal, the performance gain obtained with autoparallelization mechanisms is much higher for floating-point-based

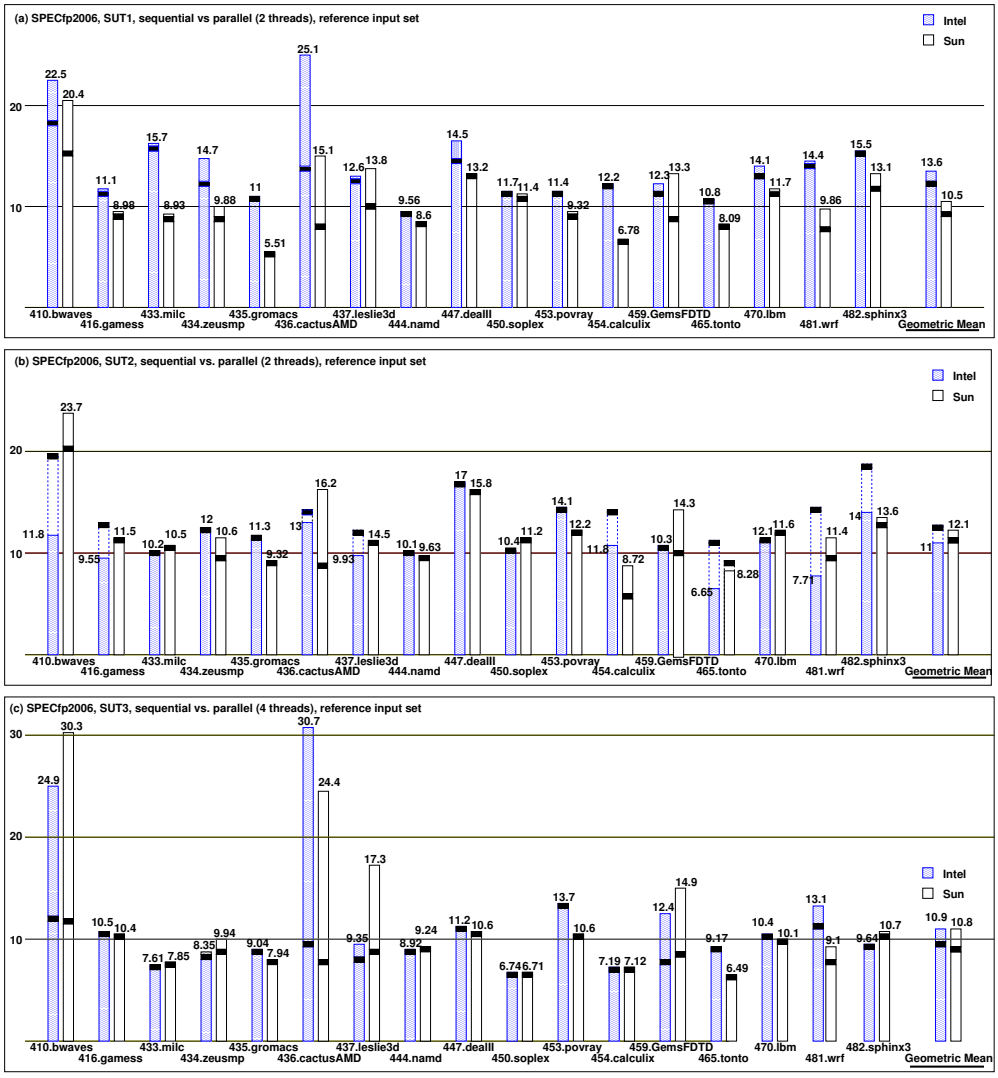

Figure 2.3: Parallel performance of SPEC CPU2006 FP benchmarks. The black box in each bar represents the performance of the sequential version of the application in the same SUT.

applications. Figure 2.3 shows the results.

The relative performance of the parallel code running in SUT1 is shown in Fig. 2.3(a). The black lines in each bar represent the performance of the sequential execution of the same benchmark in this architecture. We can see that both Intel and Sun compilers present a performance gain in almost all benchmarks, with no significant slowdown in

any case. The average performance obtained according to SPEC specifications shows an improvement of about 15% for Intel code and 16% for the code generated by Sun compilers. We can consider these values as acceptable, taking into account that they have been obtained automatically and the parallel system is composed of just two threads. It is interesting to highlight the behavior of 436.cactusAMD, with a speedup of about $2\times$ with two cores.

The situation changes drastically when evaluating the performance of the parallel code in SUT2, that runs a 64-bits environment. Figure 2.3(b) shows the results. It is surprising to discover that the Intel-compiled version of several applications run *slower* in this parallel environment, while the performance of the code generated by Sun compilers is similar to SUT1. Since the purpose of this study is to evaluating the capabilities of different compilers while running SPEC benchmark code, no effort was done in migrating the applications to a 64-bits environment other than adjusting compiler flags. For this environment, Sun compilers obtains a performance gain of about 10% with respect to the sequential evaluations on SUT2.

Finally, Fig. 2.3(c) shows the performance results when running the benchmarks in a four-threads environment. In this case, Intel achieves a 17% improvement average over the sequential code, while Sun gets about 22% improvement. Efficiency, on the other hand, drops to 29% for Intel and 30% for Sun code.

Making the average of the results for the floating-point applications of SPEC 2006, the use of auto-parallelization flags allows obtaining a significant performance gain, of about 17% on average with Intel compilers and 21% with Sun compilers.

2.3.3 Conclusions of the study

This study aims to help users to better understand the wide spectrum of compiler technology, evaluating three of the most popular compilers using SPEC2006 benchmarks. The chapter evaluates the performance of the generated code for both integer and floating-point benchmarks, not only in terms of sequential performance of the generated code, but also in terms of the availability of auto-parallelization mechanisms.

Our results show that, with respect to integer benchmarks, all three compilers perform equally well, with small differences in terms of performance that are too small to be considered significant. In the set of floating point benchmarks, differences between sequential performance of the code generated by the three compilers are more relevant. The code generated by the Intel compiler used is 39.8% faster than the code generated by GCC and 16.4% faster than the code generated by Sun compiler. Due to the constant improvements on compiler technologies, these performance differences may change in other compiler versions and/or using a different set of optimization flags.

With respect to the possibility of auto-parallelizing the code, our study concludes

that this option is not useful when processing SPEC 2006 integer applications. However, in the case of the floating-point applications of SPEC 2006, the use of auto-parallelization flags allows obtaining a significant performance gain, of about 17% on average with Intel compilers and 21% with Sun compilers. Although Sun obtains a better performance gain, it might be due to its lesser performance in the sequential tests. We lack the implementation details that explain the performance gap between these two compilers, but the study arises that their parallelization capabilities are quite similar, and differ from one SUT to another.

Considering not only overall results, it is worth noting that the performance gains are not uniform. For example, Intel presents a low performance in 462.libquantum benchmark in 64-bits systems, while GCC compiler generates fast code for 464.h264ref with SUT2. This prevents us to not generalize these conclusions to all applications written in a given programming language. Moreover, a tailored collection of flags for a given application may improve performance results in particular combinations of applications, compilers and platforms. Therefore, a developer who wants to choose between one of these compilers, either to run a sequential application or parallelize a code automatically, should evaluate their performance for his/her application and underlying architecture.

2.4 The problem of extracting more parallelism

Having in mind the risks of generalizing performance results while talking about code generated by different compilers, we can conclude from these results that parallelization capabilities of both Intel and Sun compilers are an interesting feature that is mature enough to consistently produce a performance improvement at no cost in terms of developing effort. This fact makes these options a good starting point to better exploit the capabilities of modern multicore systems.

However, and here it is the main problem and concern, relying on these features to generate a parallel version for a many-cores architecture may not be enough to fully exploit hardware capabilities of the system, due to the reduced performance gain obtained. We find that searching for alternatives, as speculative parallelization, that could exploit better CPU's processing capabilities, and the intrinsic parallelism presented in many of the source codes, is quite necessary.

Some studies suggests that the performance of thread-level speculation is very limited [97], obtaining just an 1% of improvement over SPEC CPU2006 benchmarks [96]. However, these studies and same-style others have been severely criticized in [89], because they only consider parallelism at the innermost loop level, not considering outer loops, which imply to discard a significant source of speedup. They do not either take into account the effect of compiler optimizations that could

improve the performance of speculative parallelization. Trying to correct these mistakes, Packirisamy *et al* [134] perform their own study and obtain very different results. Overall, with an optimal loop selection in SPEC CPU2006 benchmarks, they achieve a speedup of 60% on four cores, which is quite higher than the 17% obtained by Intel compiler using auto-parallelization in a four cores system in our study.

These results encourage us to seriously consider speculative parallelization, and to build procedures and tools to do this parallelization as easy as possible. A way to achieve this is taking advantage of the information present in the source code, and combining it with profile information. This path is explored in the following chapter.

The work and the conclusions described in this chapter has been published in the following papers:

- Using SPEC CPU2006 to Evaluate the Secuential and Parallel Code Generated by Commercial and Open-source Compilers. Sergio Aldea, Diego R. Llanos, Arturo Gonzalez-Escribano. *The Journal of Supercomputing, 59(1)*, January 2012, pages 486-498.

- Evaluación de compiladores comerciales usando SPEC CPU2006. Sergio Aldea, Diego R. Llanos, Arturo Gonzalez–Escribano. *Actas XIX Jornadas de Paralelismo*, Castellón, Spain, September 17–19, 2008.

Chapter 3

OpenMP `speculative` Clause Proposal and Definition

Software-based, thread-level speculation (TLS) systems allow the parallel execution of loops that cannot be analyzed at compile time. TLS systems optimistically assume that the loop is parallelizable, and augment the original code with functions that check the consistency of the parallel execution. If a dependency violation is detected, offending threads are restarted, making them to consume correct values. Although many TLS implementations have been developed so far, robustness issues and changes required to existent compiler technology prevent them to reach the mainstream. In this chapter we propose a different approach: To add TLS support to OpenMP. A new OpenMP *speculative* clause would allow executing in parallel loops whose dependency analysis cannot be done at compile time.

3.1 Problem description

The availability of multicore architectures allows users not only to run several applications at the same time, but also to run parallel code. However, the manual development of parallel versions of existent, sequential applications is an extremely difficult task because it needs (a) an in-depth knowledge of the problem to be solved, (b) understanding of the underlying architecture, and (c) knowledge on the parallel programming model to be used. Meanwhile, automatic parallelization offered by compilers only extracts parallelism from loops when the compiler can assure that there is no risk of a dependency violation at runtime. Only a small fraction of loops falls into this category, leaving many potentially-parallel loops unexploited. Alternative, recently-

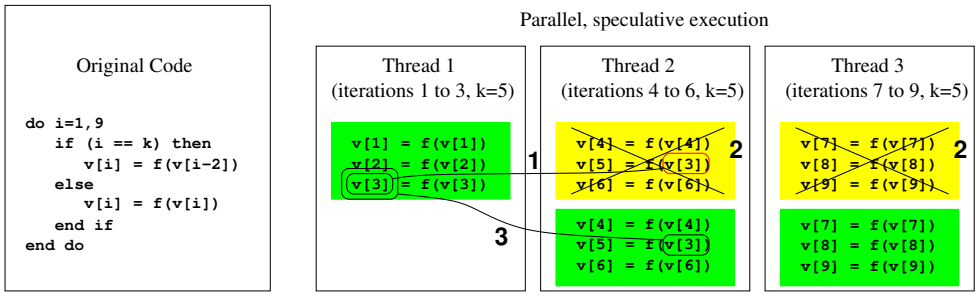

Figure 3.1: Speculative parallelization starts the parallel execution of the loop, while a monitor tracks the execution to detect cross-thread dependency violations. If such a violation occurs, (1) speculative parallelization stops the consumer thread and all threads that execute subsequent blocks, (2) discards its partial results, and (3) restarts the threads to consume the correct values.

proposed concurrency primitives, such as transactional memory, suffer from other problems such as immaturity and a lack of scalability [60].

The most promising runtime technique to extract parallelism from fragments of code that cannot be analyzed at compile time is called software-based Speculative Parallelization (SP). This technique, also called Thread-Level Speculation (TLS) [36, 53, 78] or even Optimistic Parallelization [102, 103] aims to automatically extract loop- and task-level parallelism when a compile-time dependency analysis cannot 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.

There exists a trade-off between the potential performance gain and the overheads due to the use of additional functions to monitor the consistency of the parallel execution. However, as long as not many dependency violations arise, TLS can speed up these fragments of code, effectively augmenting the percentage of sequential code that can be executed in parallel.

To better understand how SP works, we will briefly describe the different situations that may arise when two threads access the same variable concurrently, as we can see in Fig. 3.1. Informally speaking, and focusing on loop-based speculation, variables that are always written before being read in the context of a given iteration

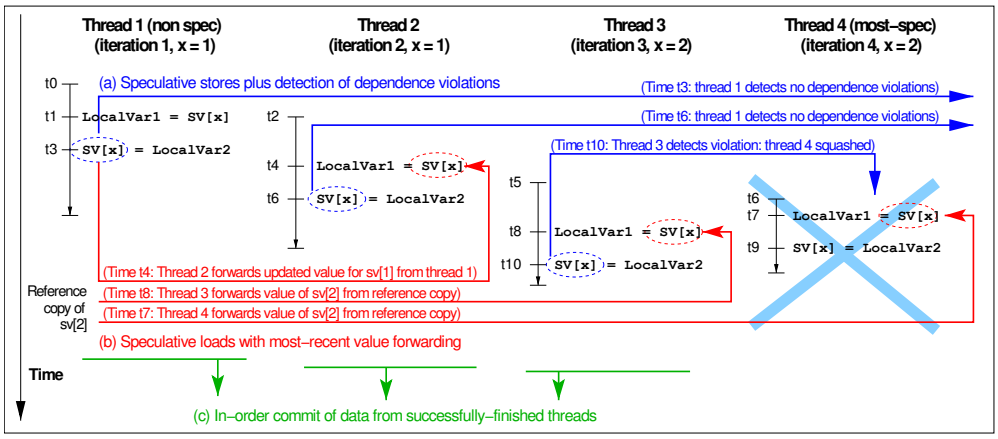

Figure 3.2: Example of speculative execution of a loop and summary of operations carried out by a runtime TLS library.

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 safely parallelized[1]. 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 dependency violation occurs. In this case, the results generated by the thread that consume 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.

Figure 3.2 shows a more detailed example of thread-level speculation. The figure represents four threads executing four consecutive iterations, and the sequence of events when the loop is executed in parallel. The value of `x` was not known at compile time, so the compiler was not able to ensure that accesses to the `SV` structure do not lead to dependency violations when executing them in parallel. Note that, at runtime, the actual indexes of `SV[x]` are known.

A handful of TLS solutions has been proposed during the last years by several research groups [36, 56, 103, 171]. However, none of these solutions are mature yet to be integrated into production compilers and runtime libraries. The limitations suffered by these systems are motivated by the lack of a compile-time framework to search for

[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<MAX; i++) {
   b = func(i);
   v[i] = b * a[i];
}
```

(a)

```
#pragma openmp parallel for \
 private (i,b) shared (a,v)
for (i=0; i<MAX; i++) {
   b = func(i);
   v[i] = b * a[i];
}
```

(b)

Figure 3.3: Example of loop parallelization with OpenMP.

code fragments suitable for speculative execution, and the difficult of automatically adding the speculative directives needed for runtime parallelization. Most TLS solutions so far relies on human intervention to transform the original, sequential code into a version that can be speculatively executed in parallel. Some proposals manually analyze and transform the original code, while others need the programmer to manually mark the potentially parallel regions with compiler directives. None of these solutions are desirable in production environments, since they require a deep understanding of both the original code and the speculative parallelization method used.

The purpose of this chapter is to discuss how to add TLS support into OpenMP [132], one of the most successful parallel programming models available. Parallel applications written with OpenMP should explicitly declare parallel regions of code. In the case of parallel loops, the programmer should classify all variables used inside the loop according to their use. Figure 3.3 shows an example of (a) a sequential C loop, and (b) its parallelization with OpenMP directives. As can be seen, all variables inside the loop body should be classified as "private" or "shared". Informally speaking, "private" variables are always defined in a given iteration before their use, being their data not propagated across iterations. On the other hand, "shared" variables have values that are visible by all threads executing the loop in parallel, i.e. their data should be available across different iterations. In our example, `a[]` is a read-only shared vector, while `v[]` is a shared vector whose elements are modified by each iteration.

Being OpenMP a simple and powerful mechanism for code parallelization, its use has several limitations. First, the classification of all variables inside of the critical region according to their use is a time-consuming, error-prone task. Second, OpenMP does not ensure the execution of the code according to sequential semantics, being the programmer responsible for such a task. In our example, in Fig. 3.3, the programmer is responsible to ensure that each thread modifies a different element of `v[]`. Third, in many cases potentially-parallel regions cannot be safely parallelized because their control flow depend on runtime data. Consider the code depicted in Fig.3.4. Suppose that the value of `k` is not known at compile time. Assuming `b>0`, if the parallel execution of the loop calculates iteration `i` *before* iteration `i-b`, access to `v[i-b]` may

```
for (i=0; i<MAX; i++) {
   b = func(i);
   if (b==k)
      v[i] = v[i-b];
   else
      v[i] = b * a[i];
}
```

(a)

```
#pragma openmp parallel for \
 private (i,b) shared (a,k) \
 speculative(v)
for (i=0; i<MAX; i++) {
   b = func(i);
   if (b==k)
      v[i] = v[i-b];
   else
      v[i] = b * a[i];
}
```

(b)

Figure 3.4: A loop that cannot be safely parallelized with current OpenMP clauses (a), and its parallelization with our new `speculative` clause (b).

return an outdated value, breaking sequential semantics. The only way to guarantee a correct behavior would be to serialize the execution of iterations $i - b$ and i, a difficult task in the general case.

Taking all this into consideration, our proposal consists in augmenting OpenMP with software-based TLS techniques, to ensure that definitions and uses of `shared` variables are carried out according to sequential semantics. To do so, we propose a new `speculative` clause. This clause would allow the programmer to handle variables whose use can potentially lead to a dependency violation, and therefore should be monitored at runtime in order to obtain correct results. Note that the use of such a category effectively frees the programmer from the task of deciding whether a particular variable is private or shared. To the best of our knowledge, no production-state parallel programming model incorporates support for software-based TLS.

In addition to the usefulness of this new clause for any programmer, the clause may also have a great impact in the usability of the software-based speculative parallelization runtime system developed by Cintra and Llanos [36, 37, 110], which this Ph.D. thesis takes as its reference. This system uses OpenMP to handle the execution of the threads, and to classify the variables in terms of their accesses. With the definition of a new *speculative* clause, variables that might lead to a dependency violation could be annotated by the programmer, and automatically instrumented at compile time.

The rest of the chapter is organized as follows. Section 3.2 describes some of the properties of OpenMP. Section 3.3 gathers the different approaches to the problem of adding support for speculative parallelism into OpenMP. Section 3.4 describes the new OpenMP clause that we propose, while Sect. 3.5 describes the changes that the proposed clause has to perform in the loop being parallelized. Finally, Sect. 3.6 sum-

marizes the conclusions of this chapter, as well as our contributions to the literature.

3.2 Analysis of OpenMP capabilities

This section explores some of the features of OpenMP that make it a good alternative to parallelize source codes, analyzing some of its limitations that lead us to propose the definition of a new clause.

OpenMP [33, 52] is an API (Application Programming Interface) developed to specify parallel computation in shared memory systems, for C, C++, and Fortran. In a shared memory system, all processors share the same memory; namely, data written by a processor is visible by others processors. Therefore, OpenMP, as a shared memory programming language, communicates by manipulating shared memory variables.

In simplified terms, OpenMP is an abstraction layer that hides most of the tricky details of parallel programming in shared memory systems. OpenMP is appropriated when the programmer knows that the code can be run in parallel, but the data dependency analysis is not able to determine whether it is safe to parallelize or not, or when the compiler lacks information to parallelize the code. The way of using OpenMP is through the addition of compiler directives in the original sequential code, annotating and identifying parallel loops, without no bigger changes.

However, OpenMP does not provide the capability of processing variables that may lead to a dependency violation. It only provides clauses, which affect the directives, to point out which variables are shared by all threads, and which variables are private to each thread. Programmers are responsible for detecting those situations in which a dependency violation may occur, and manually rewrite the code using some of the OpenMP directives. However, some situations are impossible to solve without breaking the parallelism. For these cases, and also for those situations when a manual rewriting of the code is not straightforward, it becomes greatly useful the definition of a new clause that allows handling such conflictive variables, and triggers all the changes needed in the code automatically.

To sum up, some of the OpenMP advantages and drawbacks are the following:

OpenMP advantages

- Good performance and scalability.
- De-facto and mature standard.
- Portability. OpenMP is supported by several compilers[2] as GCC, Intel's, IBM's

[2] see `http://openmp.org/wp/openmp-compilers/`

or Cray's compilers.

- OpenMP requires a little effort to parallelize since it appears as a abstract layer. With simple compiler directives it is possible to declare which variables are private or shared. OpenMP makes the rest.
- OpenMP allows incremental parallelism, working on one region at one time. Thus, substantial changes in several sections at once are not needed.
- In general, original sequential source code does not need to be changed. This reduces the chance of introducing errors.
- Using a shared memory model avoids to explicitly specify data distribution.

OpenMP disadvantages

- OpenMP does not give the programmer any hint about the feasibility of running the code in parallel. The programmer uses OpenMP at his/her own risk, and usually needs to perform some changes manually in order to parallelize codes which have likely dependency violations. The new clause proposed in this chapter aims to solve this issue.
- Even if a code is feasible to be run in parallel, OpenMP does not ensure an improved performance.
- It requires a compiler that supports OpenMP.
- OpenMP overheads are not worthy when the size of the loop is too small.

Defining a new clause to point out variables that might lead to dependency violations is the first step in order to achieve an automatic transformation of the code to handle this dependencies. In the next chapter, we will focus on how we combine the proposed clause with a TLS system in order to parallelize a code.

3.3 OpenMP and TLS: State of the art

As far as we know, there are not proposals to extend OpenMP to support software-based TLS. Instead, in the literature there are some approaches that extend OpenMP to support hardware speculation. Early works such as [117] propose the use of pragma directives to enable speculative parallelism at a hardware level, being the details of the implementation transparent to the programmer. In a similar way, [133] exposes the advantages of using OpenMP to explicit hints to the compiler and the underlying

hardware in order to extract speculative parallelism. In contrast to our proposal, these works do not define any new OpenMP directive.

More recently, proposals are focused on Transactional Memory (TM) [83, 105], an alternative that implements speculative parallelism but differs from TLS in how data dependencies are handled. TM uses transactions rather than locks to synchronize critical sections in parallel applications, ensuring that these transactions are executed atomically in isolation, similarly to the transactions of a database system. In the other hand, TLS spawns new threads from the one-thread sequential application, which is split in tasks or chunks of iterations that are speculatively executed by each thread.

Proposals such as [13, 123, 169] extend OpenMP to support TM, providing new directives and clauses in order to mark and wrap critical sections. OpenTM [13], from Stanford, is an Application Programming Interface (API) which extends OpenMP to support speculative parallelization by using TM. OpenTM defines new constructs which enable programmers to wrap critical sections within transactions, and also extends the runtime system of OpenMP, including new runtime functions, to allow the transactional execution. They implement OpenTM extending the GNU OpenMP (GOMP) implementation [130] for GCC 4.3.0. A similar proposal comes from the Barcelona Supercomputing Center. They also propose new directives or clauses to integrate OpenMP with TM [123] and present a runtime environment [122] to support their proposal. Instead modifying GCC, they use a source-to-source translator to transform the code within a transaction, and generate the proper function calls to the runtime library routines. Another proposal to include transactions in OpenMP is the IBM XL STM compiler [169]. They suggest a simple directive to wrap the structured block as a transactional region, and provide a runtime system to execute the code region. Focusing on the IBM XL compiler, there are also a proposal of directives to support hardware transactional memory [20], and an another to support hardware-based TLS [86]. Note that these two proposals require the use of the IBM Blue Gene hardware, and hence, unlike our system, they cannot be used in any architecture.

Similar to these proposal, but focused on embedded systems, Soc-TM [63] is a framework that proposes another extension to OpenMP, defining a set of compiler directives to provide hardware and software support for TM programming on embedded systems. This allows the extraction of the speculative parallelization inherent in some applications, without dealing with the low-level details of the transactional programming. Programmers only need to outline transactions in the application code, and the system transforms and execute the resulting code.

Although some of these proposals implement the code generation required, as far as we know, there are not any specific work that proposes or implements OpenMP extensions to support software-based Thread-Level Speculation. This gap is what we aim to fill with the OpenMP clause proposed in this chapter. In the next chapter we

```
#pragma omp parallel for speculative (list)
for-loop
```

Figure 3.5: New OpenMP `speculative` clause proposed.

will see how the support for this clause is implemented as a plugin-based compiler pass that supports the TLS runtime library based on the technique that Cintra and Llanos' speculative runtime system [36, 37] implements.

3.4 Solution proposed: A new OpenMP `speculative` clause

The problem of adding speculative parallelization support to OpenMP can be handled from two points of view. The first one requires the addition of a new directive, for example `pragma omp speculative for`. However, this option is more demanding, because there are many OpenMP related components that should be modified. We believe that it is preferable to use a different approach, which is the proposal of a new clause for the OpenMP "parallel loop" construct. This new clause would enable the programmer to enumerate which variables should be updated speculatively. As it is stated in Sect. 3.2, the definition of a OpenMP clause to point out conflictive variables that may lead to a dependency violation is rather useful, not only from the point of view of a programmer facing a source code with this possible issue, but also from the perspective of the TLS system, because this clause is the first step to transform the code automatically, and handle the dependencies.

Taking this into consideration, we propose a new clause for OpenMP to provide support for speculative parallelization of *FOR* loops, using OpenMP as API, and the speculative system described in Section 4.2.1 as runtime TLS library, although details on their connection will be treated in the next chapter.

The new OpenMP clause is called *speculative*, and it needs to be used as clause of a *parallel for* directive. This new clause is shown in Figure 3.5, where `list` contains variables that may lead to any dependency violation.

With this extension, programmers write OpenMP programs as usual, but now with the possibility of annotating as `speculative` those variables that could lead to a dependency violation. With this method, programmers do not have to take care of handling these potential violations, being the speculative runtime system the responsible of such task. Once a programmer annotates each variable to its type, a compiler plugin augments the code to integrate the TLS runtime library.

Figure 3.6 shows an example of the use of the proposed clause. Variable `i` is

```
1  #pragma omp parallel for default(none)
2    private (i, Q, aux)
3    speculative (a)
4  for (i = 0; i < MAX; i++) {
5    Q = i % (MAX) + 1;
6    aux = a[Q-1];
7
8    Q = (4 * aux) % (MAX) + 1;
9    a[Q-1] = aux;
10 }
```

Figure 3.6: Example of *FOR* loop annotated with the `speculative` clause.

private, since it is the variable that controls the iterations of the *FOR* loop. Variables `Q` and `aux` are private, because they are always written before being read in the context of an iteration. And finally, variable `a` is speculative, because accesses to this variable can lead to dependency violations. During parallel execution, a particular iteration may read from `a` a non-updated value and therefore the execution will be incorrect. As we will see in Sect. 4.2.1, a speculative management of `a` allows the parallel execution of this loop.

3.5 `speculative` clause internals

From the point of view of a programmer, the structure of a loop being speculatively parallelized due to the proposed clause is not so different from a loop parallelized with regular OpenMP directives. Current OpenMP parallel constructs force the programmer to explicitly declare the variables used into the parallel region according to their use, which can be an extremely hard and error-prone task if the loop has more than a few dozen lines. In this way, if the programmer is unsure about the use of a certain variable or structure, he could simply label it as speculative.

However, the use of the new clause forces the compiler plugin to perform several changes into the source code. Since the clause points out those variables which may lead to a dependency violation, the compiler has to rewrite part of the loop in order to handle possible violations, ensuring a correct parallelization of the loop.

Under speculative execution, each thread maintains a version copy of the data structure that is accessed speculatively (in Fig 3.2, the `SV` vector). At compile time, the original code is augmented to perform speculative stores, speculative loads, and in-order commits. In addition, the loop structure is rearranged in order to allow the re-execution of squashed iterations. The following paragraphs describe these operations in detail.

Speculative stores At compile time, all write operations to the data structure being speculatively accessed should be replaced with calls to a *speculative store* function. This function writes the datum in the version copy of the current thread, and ensures that no thread executing a subsequent iteration has already consumed an outdated value for this structure element, a situation called "dependency violation". If such a violation is detected, the offending thread and its successors are stopped and restarted. In the example depicted in Fig. 3.2, the checks for dependency violations performed by Threads 1 and 2 do not find any successor that has consumed an outdated value for `SV[1]`. However, at time t_{10}, Thread 3 discovers that Thread 4 has consumed in t_7 an outdated value for `SV[2]`, so a dependency violation has been found. Therefore, Thread 4 should be stopped and restarted, in a so-called *squash* operation. When Thread 4 is restarted, it will forward the updated value for `SV[2]` from Thread 3, thus successfully continuing the execution of its chunk of iterations.

Speculative loads At compile time, all reads to the speculative data structure should be replaced with calls to a function that performs a *speculative load*. This function obtains the most up-to-date value of the element being accessed. This operation is called *forwarding*. If a predecessor (that is, a thread executing an earlier iteration) has already defined or used that element, the value is *forwarded* (as Thread 2 does in Fig. 3.2). If not, then the function obtains the value from the reference copy of the data structure (as Thread 3 does in the figure).

Commit operation If no dependency violation arises during the execution of a given thread, its changes to the speculative data structure should be *committed* to the reference copy of the data structure. Note that commits should be done in order, to ensure that the most up-to-date values are stored. After performing the commit operation, a thread can receive a new iteration or block of iterations to continue the parallel work.

Scheduling iterations under TLS Finally, the original loop to be speculatively parallelized should be augmented with a scheduling method that assigns to each free thread the following chunk of iterations to be executed. If a thread has successfully finished a chunk, it will receive a brand new chunk not executed yet. Otherwise, the scheduling method may assign to that thread the same chunk whose execution had failed, in order to improve locality and cache reutilization.

Therefore, the *speculative* clause triggers significant changes into the code. Read and write operations to speculative variables have to be replaced at compile time with

function calls of "loading" and "storing" that handle these operations. Read operations have to be changed for function calls that obtain the most up-to-date value of the element being accessed. Write operations have to be changed for function calls that write the data in the version copy of the current processor, ensuring that no thread executing a subsequent iteration has already consumed an outdated value for this structure element, a situation called "dependency violation". If such a violation is detected, the offending thread and its successors has to be stopped and restarted.

Reduction operations such as sum and maximum reductions applied on speculative variables have to be detected too. Although this type of accesses could be replaced with regular loading and storing operations, replacing them with the appropriate function calls that handle them efficiently leads to faster performance.

The clause also implies changes further than speculative loads and stores. These changes depend on the underlying TLS system. In our case, the loop annotated with the *speculative* clause requires to be transformed into a loop with as many iterations as available threads. As we saw above, at the beginning of the loop body, a scheduling method assigns to the current thread the block of iterations to be executed.

Once a thread has finished the execution of the assigned chunk of iterations, a function has to be called in order to check the correct the execution of these iterations. If the execution was successful, the version copy of the data is committed to the main copy; otherwise, version data is discarded.

To sum up, tasks that should be carried out when the `speculative` clause is used includes (1) replacing the original definition of the loop, (2) changing all accesses to speculative variables for the corresponding "loading" and "storing" function calls, and (3) adding the corresponding function call that is invoked once each thread has finished its chunk of iterations, to either commit the results, or restart the execution if the thread has been squashed.

3.6 Conclusions

Adding speculative support to OpenMP would increase the number of loops that could be parallelized with this programming model. The proposed OpenMP clause, called `speculative`, would allow executing in parallel loops whose dependency analysis cannot be done at compile time. The programmer may label some of the variables involved as `private` or `shared`, using `speculative` for the rest. Note that our proposal would let to transform *any* loop into a parallel loop, although the parallel performance will depend of the actual number of dependency violations being triggered at runtime.

Details of how the clause are implemented, and a description of the specific changes required in the code in order to link with our reference TLS library, will

be given in the following chapter.

The work and the conclusions described in this chapter has been published in the following paper:

- Support for thread-level speculation into OpenMP. Sergio Aldea, Diego R. Llanos, and Arturo Gonzalez-Escribano. *Proceedings of the 8th international conference on OpenMP in a Heterogeneous World (IWOMP'12)*, Barbara M. Chapman, Federico Massaioli, Matthias S. Müller, and Marco Rorro (Eds.). Springer-Verlag, Berlin, Heidelberg, 2012. pages 275-278.

Chapter 4

OpenMP `speculative` Clause Design, Implementation and Evaluation

In this chapter we propose a compile-time system that adds support for Thread-Level Speculation into OpenMP. Our solution augments the original user code with calls to a TLS library that handles the speculative parallel execution of a given loop, with the help of the new OpenMP `speculative` clause, which was proposed in the previous chapter, for variable usage classification. To support it, we have developed a plugin-based compiler pass for GCC that augments the code of the loop. With this approach, we only need one additional code line to speculatively parallelize the code, compared with the tens or hundreds of changes needed (depending on the number of accesses to speculative variables) to manually apply the required transformations.

4.1 Problem description

As we have stated in previous chapters, development of parallel versions of existent, sequential applications is an extremely difficult task because it needs (a) an in-depth knowledge of the problem to be solved, (b) understanding of the underlying architecture, and (c) knowledge of the parallel programming model to be used. Many parallel languages and parallel extensions to sequential languages have been proposed to exploit the capabilities of modern multicore system. The most successful proposal in the domain of shared memory system is OpenMP [33], a directive-based parallel extension to sequential languages as Fortran, C, or C++, that allows the parallelization of

user-defined code regions. As we have stated in the previous chapter, OpenMP does not ensure the correct execution of the code according to sequential semantics, making the programmer responsible for such tasks. Possible dependency violations between iterations that may occur during execution need to be addressed by the programmers.

As we saw in Chap. 2, automatic parallelization offered by compilers only extracts parallelism from loops when the compiler can assure that there is no risk of a dependency violation at runtime. Only a small fraction of loops falls into this category, leaving many potentially parallel loops unexploited. Thread-Level Speculation (TLS) techniques allow extracting parallelism from fragments of code that cannot be analyzed at compile time, namely, the compiler cannot ensure that the loop can be safely run in parallel. TLS can deal with those situations in which dependency violations may occur, leading the parallel loop to finalize its execution correctly. The main problem of these techniques is that the code needs to be manually augmented in order to handle the speculative execution and monitor the possible dependencies. Programmers have to modify those accesses to variables that may lead to a dependency violation, also known as *speculative* variables.

In Chap. 3 we have proposed to extend OpenMP to allow the user to mark variables as speculative, being such variables later handled by a compile-time system that enables the automatic transformation of the code to support its execution by a TLS runtime library. As we will see, the transformations proposed are transparent to programmers, who do not need to know anything about the TLS parallel model. These key aspects of our proposal solves the problems stated above. Programmers only have to classify variables depending on their accesses, letting our solution perform all the changes needed in the source code. To do so, in the previous chapter we have defined a new OpenMP clause (`speculative`) that handles those variables whose use may lead to any dependency violation. See Chap.3 for more details concerning this new clause and its semantic.

In this chapter, we present the development of a GCC plugin-based compiler pass to support the new clause `speculative` into GCC OpenMP implementation. This pass transforms the loop with the corresponding `omp parallel for` directive, inserting the runtime TLS calls needed to (a) distribute blocks of iterations among processors, (b) perform speculative loads and stores of `speculative` variables (pointed out using the new clause), and (c) perform partial commits of the correct results calculated so far. The TLS runtime library used is based on the same design principles as the speculative parallelization library developed by Cintra and Llanos [36, 37].

This chapter also aims to contribute to the scarce documentation about GCC plugins. We explain the plugin internal structure, together with the numerous issues that appear in the development of a plugin-based, GCC compiler pass. As far as we know, most of the information provided here is not available elsewhere, unless looking into

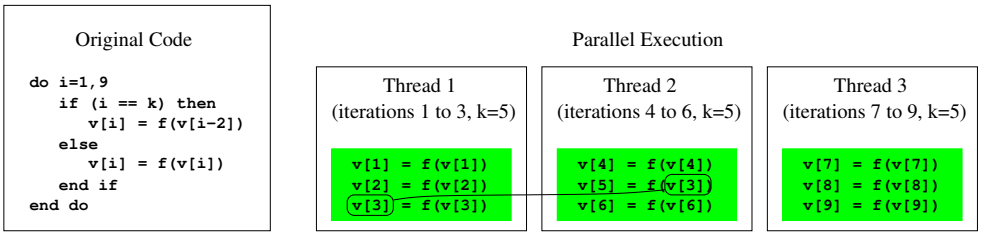

Figure 4.1: Value of k is not known at compile time, and thus the loop is not parallelizable. If $k = 5$, thread 1 may not have computed the value of $v[3]$ in time for use by thread 2. Note that if the dependency did not cross thread boundaries (for example, with $k = 6$), the compiler could parallelize the loop.

the GCC source code itself. We also describe how to build, link to GCC, and execute a plugin. In App. C we give additional details.

The rest of the chapter is organized as follows. Section 4.2 introduces some TLS key concepts, and some related work. Section 4.3 explains how the TLS runtime library used by the proposed solution works, as well as some of the transformations that need to be applied into the source code to speculatively parallelize an application. Section 4.4 describes the GCC architecture and its plugin mechanism. Section 4.5 presents the solution proposed to add support for TLS into OpenMP. Section 4.6 evaluates our OpenMP `speculative` clause with some synthetic and real-world benchmarks. Finally, Sect. 4.7 summarizes the conclusions of this chapter, as well as our contributions to the literature.

4.2 State of the art

In order to achieve the goal of implementing the new OpenMP clause, to add support for TLS, we have used the plugin mechanism of GCC. This section first describes the state of the art of TLS systems. Then, we will also introduce GCC and its plugin mechanism, with the reasons that leads to use it. Moreover, we will briefly describe some other GCC plugins that can be found in the literature.

4.2.1 Thread-level speculation systems

As it is stated in Sect. 2.1, automatic parallelization is only possible when there are not any data dependencies. If any dependency may appear in the loop, the compiler refuses to parallelize it (Figure 4.1 shows an example of a *RAW* dependency.) However, in many cases the compiler is not able to determine whether two sets of iterations

are independent. In these cases it adopts a conservative position. When this situation occurs, the compiler generates the code assuming these iterations are dependent, even if they are really not. In this context raises TLS, a technique that goes beyond the automatic parallelization and try to skip its limitations.

Speculative parallelization (SP), also called Thread-Level Speculation (TLS) or Optimistic Parallelization [102, 103] assumes that sequential code can be optimistically executed in parallel, and relies on a runtime monitor to ensure that no dependency violations are produced. A dependency violation appears when a given thread generates a datum that has already been consumed by a successor in the original sequential order. In this case, the results calculated so far by the successor (called the offending thread) are not valid and should be discarded. Early proposals [78, 142] stop the parallel execution and restart the loop serially. Other proposals stop the offending thread and all its successors, re-executing them in parallel [36, 53, 66, 171]. A recent paper by our research group [68] only restarts the offending thread and all subsequent threads that have consumed any value from it, leading to a noticeable performance improvement.

Speculative parallelization can be either implemented in hardware or software. While hardware mechanisms do not require 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. [38, 80, 116, 153, 173]). Software-based speculation, on the other hand, requires to augment the original code with instructions that drive the runtime dependency analysis. Although these instructions imply a performance overhead, software-based SP can be effectively used in current shared-memory systems with no hardware changes. This chapter proposes a software-based solution that greatly simplifies the transformation process.

As we have seen in 3.5, software-based TLS requires that the original code be augmented at compile time to perform speculative loads, speculative stores, and in-order commits. In addition, it also requires that the loop structure be rearranged in order to follow the re-execution of squashed operations. Without computational support, this is a task that programmers have to carry out manually. Our plugin solves this limitation, automatically performing all these changes required by the TLS runtime library that gives support. Programmers just need to use the new OpenMP clause we have proposed in Chap. 3 to point out which variables may lead to a dependency violation.

4.2.2 GCC: The GNU Compiler Collection

The GNU Compiler Collection (GCC) [72] is one of the most important compilers nowadays. It supports several languages, as C, C++, Objective-C, Java, Fortran (F77,F90,F95) or Ada, and more than 30 target architectures. The official meaning for GCC is “GCC Compiler Collection”, which refers to the complete suite of tools

that it includes. However, unofficially, GCC is known as the "GNU C Compiler", emphasizing its more extended use: To compile C programs. This name was the original name in its early days.

The development of GCC is supervised by the Free Software Foundation (FSF), and is the responsible organism to distribute GCC. One of the most important features of GCC is its distribution under the GNU General Public License (GNU GPL).[1]. This license allows the user not only to use and modify GCC, but also it allows the possibility of distributing these modifications, always under the GPL license, or a derived license in which the requirements for distributing the whole work cannot be any greater than the requirements that are in the GPL. This requirement is known as *copyleft*, and describes the practice of using copyright law to offer the right to distribute original or modified copies of the software preserving the same rights in these modified versions.

The GNU GPL license is one the main reasons for the popularity of GCC. But this is not the only one. GCC is written in C, paying an exceptional attention to portability. It can run on most platforms available today, and produce output for many different architectures, including cross-compiling. Of course, this feature has an obvious advantage: GCC can compile software for embedded systems that are not capable of running an entire compiler. The second argument is its modularity. Any programmer can modify GCC, adding a new front end for a new language, support for a new architecture, or adding new functionalities, thanks to its modular design. This feature, along with the GNU Generic Public License, shapes the collaborative character of the GCC project. Anyone can improve and share with others his modifications, seeing how his own contributions take part of future versions of GCC. This is one of our aims that lead us to implement the new `speculative` OpenMP clause into GCC through the plugin mechanism. We have chosen to modify GCC because it is a mainstream mature compiler, and we expect that extending GCC functionalities will have a higher impact. Moreover, as long as GCC supports more than 30 architectures, this increases the compatibility of our proposal.

Since version 4.5, GCC can be extended by plugins. Although they cannot extend the parsed language, plugins do provide extra features to the compiler, enabling to modify GCC by adding, replacing, monitoring, or even removing passes from the compiler, being unnecessary the modification of the GCC source code. The use of plugins provides several advantages, such as faster building of prototypes, easier modifications and contributions, and the use of GCC as a research compiler. Until version 4.5, extending GCC's operation required the modification of several GCC files, and a more in-depth knowledge about it was needed. Using plugins, programmers can load external shared modules, which are inserted as new passes into the compiler. We

[1] For details see the license file '`COPYING`' distributed with GCC source code.

will take advantage of this feature to develop our plugin and add support to TLS into OpenMP.

Despite being the most popular compiler, knowledge about its internal working is confusing for most of people. There are several reasons for this complexity. One of them is the huge size of its source code, more than 4.3 millions of lines of code in its version 4.6.2[2]. It accepts many different source languages, and targets a high amount of different architectures and operating systems. It also performs a lot of optimizations, most of them target-independent. Moreover, part of the code is generated at building time, adding a new level of complexity. All of these properties, together with the fact that GCC source has been coded by many different programmers, without a single architect, and split into hundreds of files, make GCC an extremely complex software.

In addition to GCC's complexity, most of the online documentation is not complete, being much information only available in the source code directly. Therefore, in this chapter, we also strive to gather and explain part of these information, mainly focusing on the plugin mechanism, which we use to implement the new OpenMP clause.

4.2.3 Other GCC Plugins

Several research groups have experimented with the GCC plugin mechanism. Among them, some plugins are designed to make the development of GCC plugins easier than with the standard procedure, such as GCC Melt [151], MilePost GCC [65], or a GCC Python plugin [113]. We decided to develop our transformation system as a GCC plugin using the standard procedure, in order to avoid dependencies to third-party, not-so-mature systems.

Besides these plugins, there are also others with different purposes, including formal verification of sequential C programs [59]), instrumentation of the compiler's intermediate representation [146]), extension of the C++ template meta-programming mechanism [47], generic static analysis that exposes different GCC intermediate representations to JavaScript [70], or program performance improvement [65]. This diversity of applications shows the wide spread of uses that GCC plugins offer.

4.3 Cintra and Llanos' TLS runtime library

In order to have a better understanding of the transformations applied by the solution proposed in this chapter, this section describes some of the most important details of the TLS runtime library which guides the development.

[2]Obtained using David A. Wheeler's 'SLOCCount'[168].

As it has been stated, the library follows the design principles of the speculative parallelization library developed by Cintra and Llanos [36, 37]. Cintra and Llanos developed a runtime library that uses a sliding window mechanism that allows the parallel execution of W consecutive chunks of iterations. Each time the non-speculative thread finishes, a partial commit takes place; the thread executing the following chunk becomes the new, non-speculative thread; and the window advances, allowing the execution of new chunks of iterations. Despite its good performance figures, the runtime library developed by Cintra and Llanos suffers from severe limitations. First, their library requires that all speculative variables were packed in a single, one-dimensional vector before the start of the speculative loop. Second, all speculative variables should share a single data type. Third, speculative variables can only be accessed by name inside the loop (no references by addresses or pointers were allowed). Finally, this runtime library creates W version copies of the entire speculative data structure, being W the size of the sliding window being used, instead of just keeping version copies of the data elements actually accessed. These limitations prevent the use of this runtime library to support a `speculative` clause, where variables and data structures labeled as `speculative` may be of different data types, can be accessed by name or address, and where speculative data structures can be of any size.

The TLS runtime library used in this Ph.D. thesis, developed by Estebanez, García-Yágüez, Llanos, and Gonzalez-Escribano [62, 69], overcomes all these limitations. It allows the speculative access to variables of any data type, both by name or by address, and managing the space needed for version copies on demand. In this section, to better understand the changes applied by the solution into the source code, we will briefly explain such transformations from the point of view of a programmer that rewrites the code.

4.3.1 Loop transformation for speculative execution

Figure 4.2 briefly shows the transformation of a parallel loop for speculative execution. This transformation is automatically carried out by our compiler plug-in. The changes are briefly described below:

Lines 2-3: Additional, internal variables are defined.

Line 4: Before the loop, the `omp_get_num_threads()` function is called to obtain the number of available threads.

Line 5: A `specbegin()` function is called to initialize the execution of the following parallel loop. If it is the first loop being parallelized, this function also initializes the runtime speculative library.

```
 1:  char a; float b;
 6:  #pragma openmp parallel for \
        private (i) \
        speculative (a, b)
 7:  for (i=0; i<MAX; i++) {
       Original loop code, part 1
10:    a = f(b);
       Original loop code, part 2
17:  }
```

(a)

```
 1:  char a; float b;
 2:  char temp; float value;
 3:  int tid, threads; ...
 4:  threads = omp_get_num_threads();
 5:  specbegin(MAX);
 6:  #pragma openmp parallel for \
        private (i, tid, temp, value,...)  \
        shared (a, b, threads,...)  \
 7:  for (tid=0; tid<threads; tid++) {
 8:   while(true) {
 9:    i = assign_following_chunk(tid,MAX,...);
       Original loop code, part 1
10:    specload(&b, sizeof(b),..., &value);
11:    temp = f(value);
12:    specstore(&a, sizeof(a),..., &temp);
       Original loop code, part 2
13:    commit_or_discard_data(tid,...);
14:    if(no_chunks_left(tid, MAX,...))
15:      break;
16:   }
17:  }
```

(b)

Figure 4.2: Loop transformation to allow its speculative execution: Original (a) and transformed (b) code.

Line 6: All variables labeled as `speculative` are automatically reclassified as `shared`. Besides this change, all reads and stores inside the *loop body* on that `speculative` variables (not shown in the figure) are replaced with calls to `specload()` and `specstore()` functions, in order to keep sequential consistency, as described in Sect. 4.2.1. Our compiler plug-in also labels other internal variables needed by the runtime systems as `private` and `shared`, such as `tid` and `threads` in our example.

Line 7: The original loop structure is replaced with a parallel `FOR` loop with just "`threads`" iterations. This launches the number of desired threads.

Line 8: A `while(true)` loop ensures that each thread repeteadly requires a chunk of iterations from the original loop to be processed. If no chunks are left, a `break` statement exits this loop and the end of the thread is reached (see line 14).

Line 9: Inside the loop, each thread receives the index of the first iteration of its assigned chunk and proceeds with the original loop body.

Lines 10-12: The read of `b` variable in line 9 of Fig. 4.2(a) is replaced with a call to the `specload()` function, that recovers the most up-to-date value for this variable. The exact behavior of `specload()` is described later in this section. The value is stored in a private, temporal location.Line 10 of Fig. 4.2(a) also performs a write on `a`. This write is replaced with a call to `specstore()` (line 9), that first stores the value in a local version copy and then checks whether a successor has already consumed an outdated value of `a`. If so, the offending thread and some or all of its successors (depending on the squash policy being defined [68]) are squashed.

It is important to highlight that only the lines of the original loop body that involve speculative variables are changed in this way: the remaining code is left with no changes.

Line 13: Once finished the original loop body, a call to `commit_or_discard_data()` checks whether the thread has been squashed or not. If a squash operation was issued by a predecessor, local copies of speculative data will be discarded. If the thread has not been squashed and it is the not-spec one, a partial commit will occur.

Line 14: After finishing their tasks related to the current chunk, all threads check whether there are no pending chunks to be executed. If there is no pending work, threads leave the `while` loop.

When all threads have exited the `while(true)` loop, the end of the parallel section has been reached and (despite the number of needed attempts) all chunks of iterations have been successfully executed, and their results committed to the `speculative` variables.

4.4 GCC and the plugin mechanism

Before describing the implementation of the new `speculative` clause, it is necessary to introduce the GCC architecture, and the plugin mechanism to add new passes and functionality to GCC operation. Moreover, we strive to clarify some aspects of the plugins, including some key points about how implementing them correctly.

4.4.1 GCC Architecture

Figure 4.3 shows the scheme of the GCC architecture [40, 71, 128, 129, 131]. In basic terms, GCC is a big pipeline that converts one program representation into another, through different stages. Each stage generates a lower-level representation, until the assembly code is generated in the last stage.

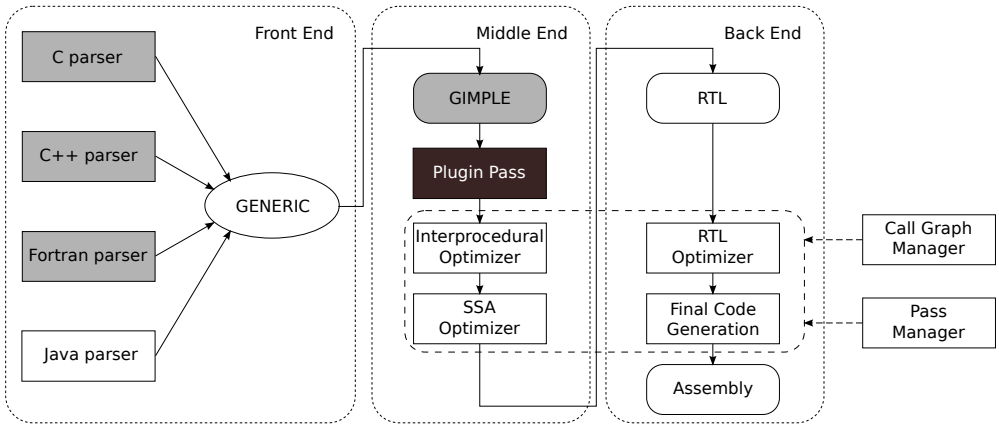

Figure 4.3: GCC Compiler Architecture. The main OpenMP related components, highlighted in grey, are the C, C++ and Fortran parsers, and the GIMPLE IR level. Highlighted in black is the location of our plugin pass.

GCC architecture has three clearly different blocks: *Front End*, *Middle End* and *Back End*. There is one front end for each programming language. The parser of each language converts source files into an unified tree form, called GENERIC, that is a high-level tree representation. When it finishes, the Front End emits a GENERIC representation of the code, that serves as the interface between the front end and the rest of the compiler.

The Middle End works on GIMPLE, which is a 3-address language with no high-level control flow structures. In GIMPLE, each statement does not contain more than 3 operands (except function calls), control flow structures are combinations of conditional statements and goto operators, and there is a single scope for variables. This kind of representation is convenient to optimize the source code, and thus, we have added our plugin in this point.

The transformation process from GENERIC into GIMPLE is performed in two phases. First, a High GIMPLE representation is created, and then, it is transformed to a Low GIMPLE representation. Figure 4.4 shows the whole process, called *gimplification*. Figure 4.5 shows the *gimplification* process that transforms the code shown in Fig. 3.6. Once the source code is in GIMPLE form, an *interprocedural optimizer* is called, where *inlining* operations, *constant propagation*, or *static variable analysis* are performed.

The following step is the transformation from GIMPLE into SSA (Static Single Assignment) representation. Figure 4.6 shows an example of this transformation. In SSA form, each variable is assigned or written only once, creating new versions for

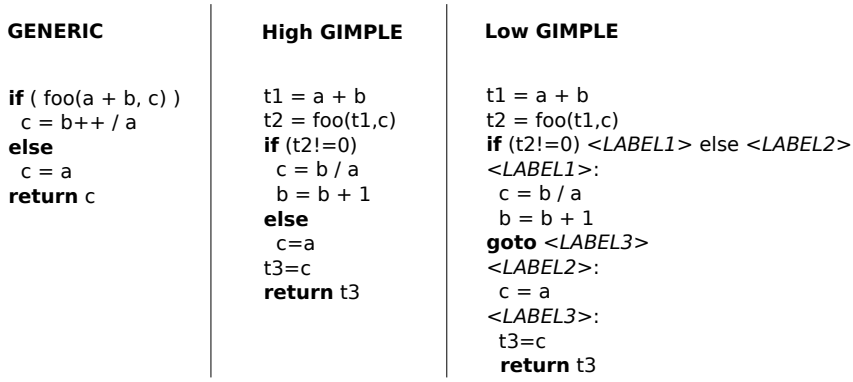

Figure 4.4: Transformation example from GENERIC into a Low GIMPLE representation.

each assignment of the same variable, that can be read many times. When different versions of the same variable are written into both branches of a conditional expression, a ϕ-function is added just after the conditional block, allowing the selection of the correct version of the variable, depending on the branch executed. SSA representation is used for several optimizations as forward expression substitution, loop interchange, vectorization or parallelization, among others. These optimizations are performed in around 100 passes.

After these optimizations, the SSA representation is converted back to the GIMPLE form, which is transformed into a *register-transfer language* (RTL) form, in which the Back End works on. RTL was the original primary intermediate representation used by GCC. It is a hardware-based representation that corresponds to an abstract machine with an infinite number of registers. GCC also uses this form to perform several optimizations, such as branch prediction or register renaming, in around 70 passes. Finally, the *Final Code Generation* step of the Back End creates the assembly code for the target architecture (x86, mips, etc.) from the RTL representation.

Transactions between the different phases are sequenced by the *Call Graph* and the *Pass Manager*. The Call Graph Manager generates a call graph for the compilation unit, decides in which order the functions are optimized, and drives the interprocedural analysis. The Pass Manager sequences individual transformations and handles pre- and post-cleanup actions as needed by each pass. All passes executed by the Pass Manager are launched in `init_optimization_passes()` in `passes.c`. Since the version 4.5 it is possible to add new passes to this compile process through the plugin mechanism.

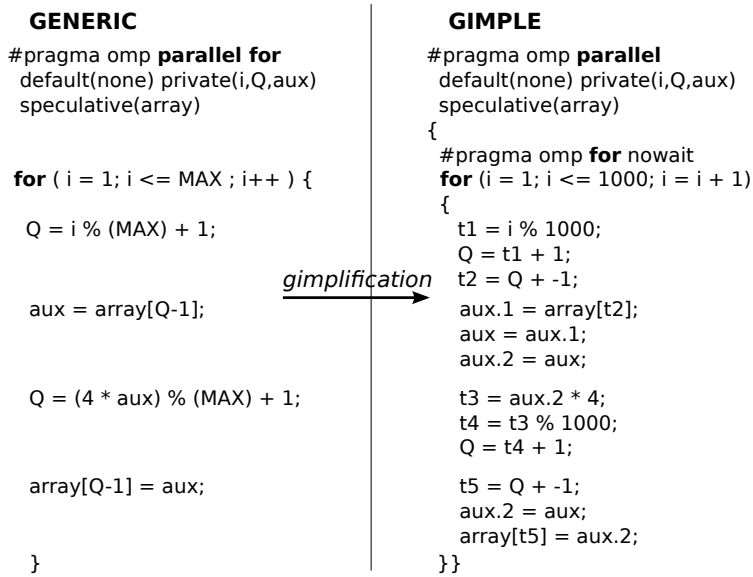

Figure 4.5: Example of transformation of the code in Fig. 3.6 from GENERIC into a GIMPLE representation.

4.4.2 GCC plugins

Since the version 4.5, GCC can be extended by plugins. Plugins provide extra features to the compiler, but cannot extend the parsed language, except using events for `#pragma`-s or `__attribute__`, among others. Plugins allow the modification of GCC, by adding, replacing, monitoring, or even removing passes from the compiler, but without touching the GCC source code. This has several advantages:

- Faster building of prototypes, since it is not necessary anymore to modify and recompile GCC.
- Easier modifications and contributions. Some internal APIs has been wrapped for external use.
- Extensibility, enabling to write specific passes to process source code, which do not have place inside GCC since they are tied to a particular domain, corporation, community or software.
- Use of GCC as a research compiler, allowing to share your modifications with the community in an easier way since you only have to distribute the plugin code, and not each GCC file modified.

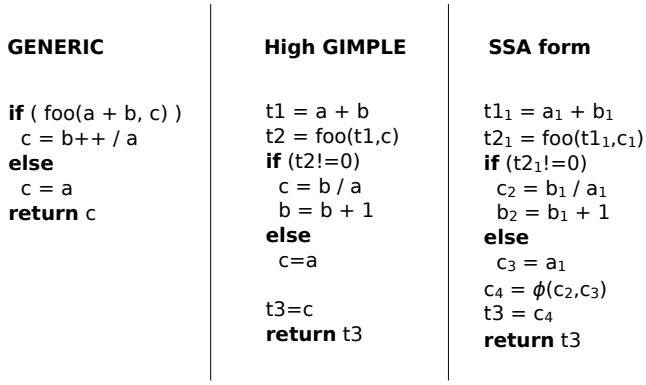

Figure 4.6: Transformation example from original GIMPLE to SSA.

Before the advent of GCC plugins, the only way to add a new pass into GCC was modifying several GCC files. In a simplified way, the steps that were necessary to add a new pass are the following [131]:

- The new pass can be implemented in a new file, which have to be added to `gcc` directory, or by editing an existing pass.
- The file `Makefile.in` controls the compilation process, so it was necessary to add a new target rule to this file.
- If we want that the plugin requires a flag to be triggered, it is necessary to be added to `gcc/common.opt`, which is a file that collects all the options for the language- and target-independent parts of the compiler.
- Create a new instance of the struct `tree_opt_pass`, which in recent versions (since 2008) is called `gimple_opt_pass`, `rtl_opt_pass` or `simple_ipa_opt_pass`, depending on where we add the new pass.
- Once the struct `tree_opt_pass` has been created, declare it in file `gcc/tree-pass.h`, and add it to the compilation process using `NEXT_PASS` in the function `init_optimization_passes()` in file `passes.c`.
- Finally, add a gate function to read the new flag.
- With all the code written, document the new pass in `gcc/doc/invoce.texi`.

As it can be seen in the all the steps above, adding a new pass required to edit and modify several files. All this work has been simplified since the implementation

of plugins into GCC. Currently, using plugins programmers can load external shared modules, which are inserted as new passes into the compiler. We will take advantage of this feature to develop our plugin to add support to TLS into OpenMP.

Plugins are developed independently to GCC as separate files, and then compiled into shared libraries which are loaded into GCC at run-time. Essential information about plugins can be divided in two parts: (1) How the source files that code plugins are structured, including the description of each part and its meaning, and (2) how they need to be compiled and loaded into GCC.

Plugin structure

Plugins have a well-defined structure, as it is shown in Fig. 4.7. The following items describe each part of this structure.

- **Line 1**: We need to include the header for the basic GCC API. After this header we can include any of the headers located in the plugin directory[3]. For example, if we are dealing with GIMPLE nodes, we need to include `gimple.h`.
- **Line 2**: This line asserts that the plugin has been licensed under a GPL-compatible license. It is the license check, and not a functional part of the plugin.
- **Line 3**: Plugin information about itself, in case users ask for help using `gcc -v` or `gcc -help`.
- **Line 4**: GCC version checking. GCC API might not be the same between different version. It is necessary to explicit the version of GCC that our plugin is compatible with. **Line 23** finishes the plugin if the version of GCC is not correct.
- **Line 5**: The gate is a callback tripped just before the plugin is executed. If the gate returns true, the plugin is executed, otherwise it is skipped.
- **Line 6**: This is the function that actually implements the plugin pass. For example, in a GIMPLE pass, we can traverse each statement from each function being processed by the plugin.
- **Lines 7-20**: This structure is the compiler pass descriptor. Among other fields, we can describe the type of the compiler pass (`GIMPLE_PASS`, `IPA_PASS`, `SIMPLE_IPA_PASS`, or `RTL_PASS`), the name of the pass, the gate and execute function, the list of properties required, provided or destroyed by the pass and the

[3]To see where the plugin directory is located, execute `gcc -print-file-name=plugin`.

```
1  #include <gcc-plugin.h>
2  int plugin_is_GPL_compatible=1;
3  static struct plugin_info myplugin_info =
4    { .version = "001", .help = "Work in progress", };
5  static struct plugin_gcc_version myplugin_ver = { .basever = "4.6.2", };

7  static bool myplugin_gate(void) { return true; }
8  static unsigned myplugin_exec(void) { /* pass code */ }

10 static struct gimple_opt_pass myplugin_pass = { {
11   GIMPLE_PASS,
12   "plug",                        /* name */
13   myplugin_gate,                 /* gate */
14   myplugin_exec,                 /* execute */
15   NULL,                          /* sub */
16   NULL,                          /* next */
17   0,                             /* static_pass_number */
18   0,                             /* tv_id */
19   PROP_gimple_any                /* properties_required */
20   0,                             /* properties_provided */
21   0,                             /* properties_destroyed */
22   0,                             /* todo_flags_start */
23   TODO_dump_func                 /* todo_flags_finish */  } };

25 int plugin_init(struct plugin_name_args *info,
26          struct plugin_gcc_version *ver) {
27   struct register_pass_info pass;
28   if (strncmp(ver->basever, myplugin_ver.basever, strlen("4.6.2")))
29          return -1; /* Incorrect version of GCC */

31   pass.pass = &myplugin_pass.pass;
32   pass.reference_pass_name = "mudflap";
33   pass.ref_pass_instance_number = 1;
34   pass.pos_op = PASS_POS_INSERT_AFTER;
35   register_callback("plug",PLUGIN_PASS_MANAGER_SETUP,NULL,&pass);
36   register_callback("plug",PLUGIN_INFO,NULL,&myplugin_info);
37   return 0; }
```

Figure 4.7: Example of plugin structure, including the definition of the plugin, the initialization function, and the code that executes the compiler pass.

list of actions that the pass manager should do before or after the pass execution. A list of properties and actions is found in file `tree-pass.h`, where it is also described each of the fields of the pass descriptor.

- **Lines 21-30**: `plugin_init()` is the first function called when the plugin is loaded, and it is responsible for doing the initialization and registering all the callbacks required.

- **Lines 24-27**: These lines set information about the pass implemented (line 24)

and tell GCC in which point our plugin will be called. In our case, the plugin is called after the first iteration of the pass *mudflap*, which is the compiler pass executed just before GCC parses the OpenMP directives and clauses.

- **Lines 28-29**: These lines register the plugin to be handled by the PLUGIN_PASS_MANAGER, using the name established in the corresponding field of the structure that describes the compiler pass, and register the information about the plugin

Compiling and executing a plugin

Firstly, in order to execute a plugin it is necessary that the version of GCC be 4.5 or superior, and that GCC have been compiled with the plugin support enabled (easily checkable invoking `gcc -print-file-name=plugin` which returns the plugin directory). To compile a plugin we can execute the following line:

```
$ gcc -I`gcc -print-file-name=plugin`/include -fPIC -shared -O2 plugin.c -o
    plugin.so
```

Executing a plugin is as easy as running GCC with the following flag:

```
$ gcc -fplugin=/path/to/plugin/name.so
```

which tells GCC to load the shared object and execute it as part of the compiler. This kind of operation shows one of the main advantages of using plugins. Plugins allow us to program and compile them independently to GCC, and activate them just when we need.

More details about GCC and the plugin mechanism are found in Appendix B.

4.5 Solution proposed: The ATLaS compilation framework

In order to add support for Thread-Level Speculation into OpenMP, we proposed in Chap. 3 a new clause, called `speculative`. In this chapter we propose a new compilation module, in a form of GCC plugin, not only to parse the new clause, but also to generate all the changes needed to effectively parallelize the code using a Thread-Level Speculation runtime library. This compilation module, together with the TLS runtime library, makes up a new system called ATLaS (Applied Thread-Level Speculation).

4.5.1 Updating GCC to support the new `speculative` clause

We have already described the GCC architecture and the plugin mechanism to add new compiler passes. The compiler-phase of ATLaS is based on the compiler GCC 4.6.2 [72] for the C programming language, extending its functionality through a plugin. This section describes the modifications that are performed in GCC to add support for the new `speculative` clause, and the development of the plugin.

Parsing the new `speculative` clause

Although the plugin mechanism enables us to perform all the changes needed by the TLS runtime library, plugins do not allow extending the parsed language. Therefore, adding a new OpenMP clause recognized by GCC requires not only the creation of a plugin, but also modifying the GCC code itself. In order to parse the new clause `speculative`, we have extended the GNU OpenMP (GOMP), an OpenMP implementation for GCC. The main parts of the GCC architecture related with OpenMP are highlighted in grey in Figure 4.3. GOMP has four main components [130]: parser, intermediate representation, code generation, and the runtime library called `libGOMP`. In relation to GOMP, we have focused on modifying its parsing phase and the intermediate representation (IR). The generation of new code to support TLS is located in the plugin developed, and it is mainly composed of calls to the TLS library functions needed for the speculative execution.

The parser identifies OpenMP directives and clauses, and emits the corresponding GENERIC representation. We have modified the C parser and the IR to add support for the new clause `speculative`. First, we have created the GENERIC representation of the new clause like other standard clauses. Then, the compiler has been prepared to recognize and parse the clause as part of the parallel loop construct. When the new clause has been parsed and the IR is generated, our plugin detects the clause and starts all the transformations needed on the code.

Modifying GCC to detect the clause: Cumbersome details

Novillo described in [130] some details about the OpenMP implementation in GCC. In this section we will describe the changes needed[4] to parse the new `speculative` clause. These changes involved the parser and the IR of GCC.

Parser Since our speculation runtime system works with programs written in C, we need to modify the C parser in `c-parser.c`. This file represents the front end for C programs. As we said in Sect. 3.4, the new clause is designed to be used as

[4]For GCC 4.6.2.

clause of a *parallel for* directive. Thus, we have modified the masks OMP_PARALLEL_-CLAUSE_MASK and OMP_FOR_CLAUSE_MASK to include the new clause: PRAGMA_OMP_-CLAUSE_SPECULATIVE. This element has been previously added to the enumeration pragma_omp_clause in c-family/c-pragma.h. Then, we have modified the c_-parser_omp_clause_name() function to recognize the word *speculative* as clause, and c_parser_omp_all_clauses() to parse the clause and the list of speculative variables.

Each speculative variable is treated by the speculative runtime system as a shared variable. Therefore, in omp-low.c, in each switch statement where we found a case for nodes OMP_CLAUSE_SHARED, we have added a new case for the node OMP_CLAUSE_-SPECULATIVE. We have done the same in gimplify.c and tree-nested.c files. In file c-typeck.c, we have modified the function c_finish_omp_clauses(), which validates all clauses against OpenMP constraints, to include the new clause. Finally, in the switch found in c_split_parallel_clauses() in c-family/c-omp.c, we have added a new case for the node OMP_CLAUSE_SPECULATIVE.

IR Directives and clauses have defined their corresponding GENERIC nodes in tree.* files. We need to update some of these files in order to add the GENERIC code for the new clause. In tree.h we have added the code OMP_CLAUSE_SPECULA-TIVE for the new clause in the enumeration omp_clause_code. The position where we insert the code in this enumeration is important, because it is used to index the tables omp_clause_num_ops and omp_clause_code_name. According to these relative positions, we have done two changes in tree.c: the number of operands of the new clause has been added to the array omp_clause_num_ops[], and in the same position of the array omp_clause_code_name[] we have added the name of the clause. It has been also added the node OMP_CLAUSE_SPECULATIVE as a case in the switch of the function walk_tree_1.

Finally, in order to correctly dump the new clause, we have added a new case in the switch of the function dump_omp_clause() in tree-pretty-print.c

4.5.2 Plugin-based compiler pass description

Once the new clause proposed is recognized by GCC, programmers can set the speculative variables, and the plugin developed can augment the original code. Using the plugin mechanism, our system adds a new compiler pass into the GCC pipeline. This new pass performs all the transformations needed in the code when the programmer annotates a variable of a loop as speculative.

The plugin developed is based on the version 4.6.2 of GCC. The new pass is run once the compiler has transformed the code into GIMPLE, and just before GCC does the first pass related to OpenMP (omplower). Therefore, our GIMPLE transformation

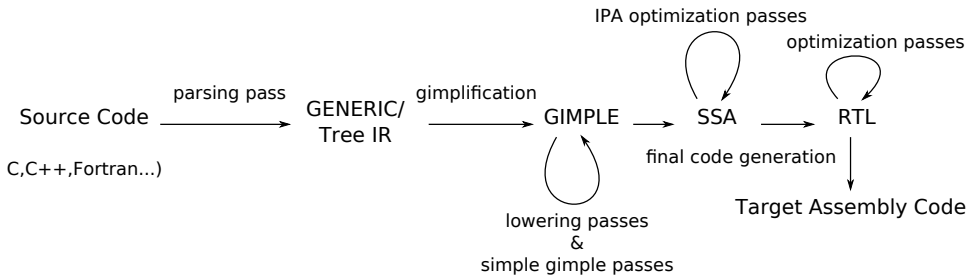

Figure 4.8: GCC passes and intermediate code representations. The plugin works on the GIMPLE representation.

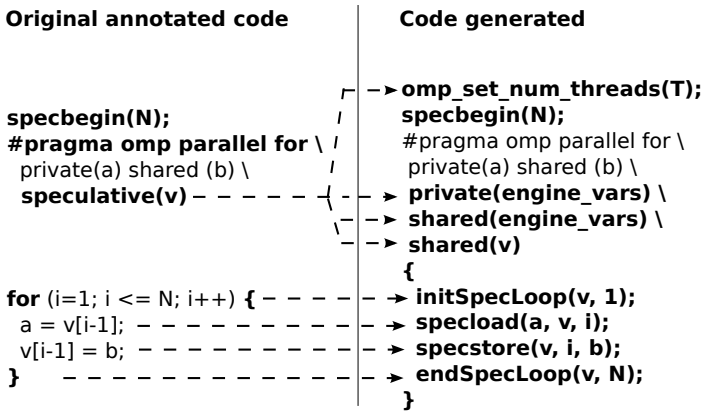

Figure 4.9: Example of code annotated and the resulting transformed code. `initSpecLoop` and `endSpecLoop` are macros that expand to more code, hidden here for legibility reasons.

pass is added before `pass_lower_omp` in `passes.c`. As we can see in Fig. 4.8, at this point we have the code in a GIMPLE representation, and the *FOR*-loop marked with the `omp parallel for` directive preserves all the clauses written by the programmer. Therefore, we have the information about which variables are shared, private, and speculative, the latter thanks to the new clause proposed. After this pass, GCC processes speculative variables as shared, while their handling as speculative will be carried out at runtime by the TLS library.

Figure 4.9 shows a brief example of the transformations made by the plugin. The parser detects the new `speculative` clause, and the new compiler pass automatically performs all the transformations needed to speculatively parallelize the loop. If the plugin does not find the `speculative` clause on the pragma, the semantic of the loop

remains identical to any other standard OpenMP loop. With the list of variables and data structures that should be speculatively updated, the plugin replaces each read of one of these variables or data element with a `specload()` function call. Similarly, all write operations to speculative variables are replaced with a `specstore()` function call. Loads or stores involving other variables do not require additional changes in the code, since all flavors of *private* and *shared* variables keep their respective semantics in the context of a speculative execution. The plugin also adds all the structures and functions needed to run the TLS system that parallelize the code. This process is completely transparent to programmers, shielding them from the intricacy of the underlying speculative parallelizing model. They only have to label the variables involved in the target loop as *private* or *shared*, as with any other OpenMP program, and label as *speculative* those variables that can lead to any dependency violation.

The scheme of the process followed by the plugin can be resumed in the following steps:

1. The plugin traverses each function of the original program looking for an OpenMP parallel loop directive with a *speculative* clause on it. If the plugin does not find the *speculative* clause on the pragma, the semantic of the loop remains identical to any other standard OpenMP loop.

2. If the plugin finds the *speculative* clause, it extracts the speculative variables pointed by the clause, and adds an OpenMP library function: `omp_set_num_threads(T)`, where `T` is the number of threads indicated in the compilation command.

3. The plugin labels as *private* or *shared* those variables needed by the runtime system. The code generated by the plugin also includes the creation of other new variables that are also added to the *private* or *shared* lists.

4. The plugin adds all the code needed to run the TLS system, including the replacing of the original loop by a new loop that drives the speculative execution.

5. The plugin traverses the GIMPLE nodes of the loop searching for readings from and writings into the speculative variables. Each read is replaced by a `specload()` function; each write is replaced by a `specstore()` function.

6. If the plugin detects situations of sum or maximum reductions, it replaces them by a `specredadd()` or a `specredmax()` functions, respectively.

Once the plugin has transformed the loop, GCC operation continues with the next passes. When the compilation ends, the resulting binary file is prepared to run speculatively.

Interface with Estebanez *et al.* TLS runtime library

The plugin-based compiler has to augment the code with the functions and structures needed for the speculative execution, and defined by the TLS runtime library. The library used, due to Estebanez, García-Yágüez, Llanos and Gonzalez-Escribano [62, 69], is largely based in Cintra's and Llanos' work (see [36, 37] for details). The plugin has to replace accesses over speculative variables with `specload()` or `specstore()` function calls. The interface of `specload()` is the following:

```
specload(UCHAR* addr, UINT size, UINT chunk_number, UCHAR* value)
```

The first parameter is the address of the speculative variable; the second parameter is the size of the variable; the third one is the number of chunk being executed (needed to infer the slot being used); and the last parameter is a pointer to a place to store the datum requested.

The interface of `specstore()` is the same than `specload()`, but the last parameter is a pointer to the value to be stored into the speculative variable.

```
specstore(UCHAR* addr, UINT size, UINT chunk_number, UCHAR* value)
```

Replacing speculative loads and stores requires the plugin to detect code lines where a write and/or read is applied, to extract the type of the speculative variable or the particular field of an speculative structure, and to perform the changes needed, including the addition of new variables to handle the temporal values required.

The plugin is also able to detect reductions applied on speculative variables, replacing them by the appropriate function calls to the TLS runtime library that handle them, called `specredadd()` and `specredmax()`. These functions have the same interface than the speculative stores:

```
specredadd(UCHAR* addr, UINT size, UINT chunk_number, UCHAR* value)
specredmax(UCHAR* addr, UINT size, UINT chunk_number, UCHAR* value)
```

In the case of sum reductions, the last parameter of the function is the value that is added to the speculative variable. For reductions in which the maximum value needs to be calculated, this last parameters is the new possible maximum value for the speculative variable. Figure 4.10 sketches the transformations performed for sum and maximum reductions.

The plugin has not only to replace accesses over speculative variables with their corresponding function calls. The TLS runtime library also requires other functions and structures that the plugin has to correctly insert into the code. Figures 4.9 and 4.10 sketch some of the transformations needed. Regarding the original loop, the plugin

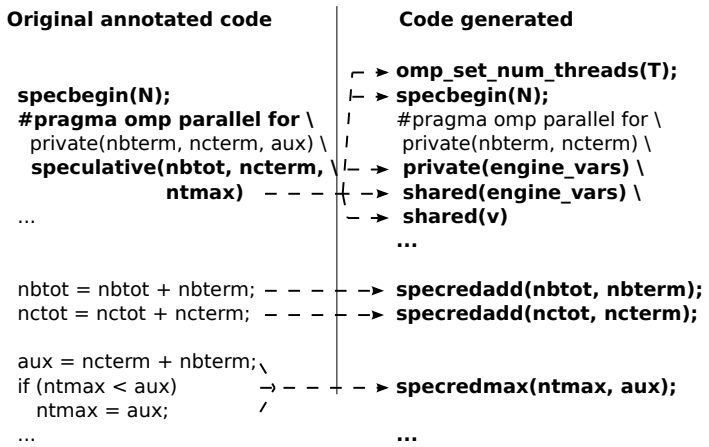

Figure 4.10: Example of sum and maximum reductions and the resulting transformed code.

replaces the parallelized loop with a new loop that drives the speculative execution. This new loop iterates over the threads, has the same body as the original (although it is augmented with extra code that ensures the correct distribution of iterations among the threads,) and commits the data stored in the speculative variables. The definition of the new loop and the code inserted before the body of the original loop are gathered in the macro `initSpecLoop()` in Fig. 4.9 for simplicity. The code lines which are required to be inserted after the body of the original are gathered in the macro `endSpecLoop()`.

Besides modifying the target loop and its body, the plugin also adds an OpenMP library function before the parallelized loop: `omp_set_num_threads()`, which sets the number of threads for its parallel execution.

Handling different types of `speculative` variables

As the TLS runtime library, the plugin developed is able to handle different types of speculative variables. A target loop may have several speculative variables and data structures, each of them being of different data types, accessed by name or address, and where data structures can be of any size. Our plugin avoids programmers to address the issues resulting from these different situations. Programmers just need to mark which variables are `speculative`, whereas the plugin and the runtime library deals with their different types, and sizes.

In order to handle this task, the plugin detects the size of the `speculative` variable, and properly sets the `size` parameter of `specload()` and `specstore()` functions. For example, it sets the `size` parameter to 1 if the speculative variables is a

`char`, to 4 if is an `integer`, to 8 if is a `double`, or the size of an structure element if it is part of a structure. The plugin also creates new variables to save temporal data that these functions need, with the same type than the target and speculative variable.

Scheduling

The current scheduling methods implemented by OpenMP are not enough to handle speculative parallelization. These methods assume that the task will never fail, and therefore they do not take into account the possibility of restarting an iteration that has failed due to a dependency violation. The scheduling method used with speculative parallelization is different from classic scheduling methods, e.g. [79, 101, 161]. Under TLS, the execution of an iteration or chunk of iterations can be discarded, so the scheduling method should be able to re-assign the squashed iteration to the same or a different thread. Although the scheduling process is implemented by the TLS runtime library, the plugin also changes part of the loop structure to allow the re-execution of iterations.

Handling complex variables and statements

The plugin is able to handle all definitions and uses of scalar variables, not only simple assignments. This includes dealing with complex statements, that are required to maintain the same order in which the multiple speculative loads and stores are executed. The plugin first handles the loads, creating new temporal variables that take part of the expression that assigns a value to the speculative variable. After replacing the loads for the corresponding `specload()`, the plugin handles the store into the speculative variable by placing a `specstore()`. An example of this situation is a writing into a speculative array with the use of a speculative variable as index.

Programmers may write other constructs that the plugin can deal with. This includes assignments from one pointer to another, accesses involving directions or the data pointed out by the pointer, assignments between entire data structures or only certain fields of those structures, speculative variables involved in casting operations, etc. The plugin handles these situations with a right setting of the size of the actual speculative variable that is read or written, and correctly typing the temporal variables required for the TLS runtime library.

Using the plugin to compile the user code

From the point of view of programmers, to speculatively parallelize a source code with our system they only have to add an OpenMP parallel loop directive and set a few parameters to the compiler. First, programmers should add the OpenMP directive

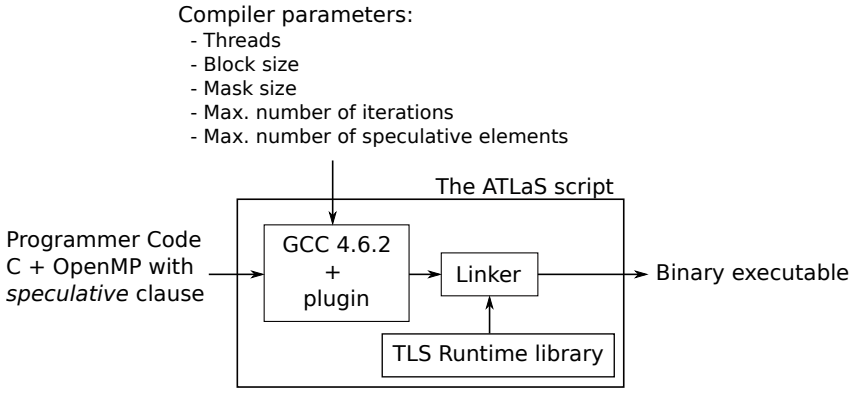

Figure 4.11: Overview of the code generation process for the *speculative* clause.

in the target loop, and classify its variables according to their usage in `private` and its variants, `shared`, and `speculative`.

Second, to compile the program, programmers should indicate the value for some parameters, such as the size of the block of iterations that will be issued for speculative execution, or the number of threads they want to launch. Unfortunately, some of these values should be embedded in the plugin. This forces the programmer to recompile the plugin before the processing of a source code that includes the `speculative` clause. We have developed a wrapper script that launches the compilation of the plugin plus the speculative runtime system. More details about the parameters and how to run this script are found in Appendix C. In short, this script has to be executed with the following parameters:

```
$ atlas –threads T –block B –maxpointer P –maxiter I –mask M –c "example.c"
```

where `I` is the maximum number of iterations that a speculative loop can execute in the program, `T` is the number of threads we want to run the program with, `B` is the size of the block of iterations, `P` is the maximum number of elements which are speculative, and `M` is the size of the mask used in the hash-based solution implemented by the TLS runtime library to recover the version copies of the speculative variables [62].

These parameters are set by the programmer and they are not very tricky to choose, because to set `maxiter` and `maxpointer` they only need to know some basic features of the target loop. For example, a loop that speculatively reads from and writes into an array of 1000 elements, and has 200 iterations sets the value of `P` to 1000, and `I` to 200. The other three parameters, the number of `threads`, the `block` size, and the `mask` size are variable, and programmers can experiment with different values to obtain the

```
 0: #define NITER 6000
 1: int array[MAX], array2[MAX];
 2: struct card{ int field; };
 3: struct card p1 = {3}, p2 = {99999}, p3 = {11111};
 4: char aux_char = 'a';
 5: double aux_double = 3.435;
 6: int i, j;
      ...
 7: #pragma omp parallel for default(none) \
 8:   private(i, j) shared(array1, p2) \
 9:   speculative(p1, p3, aux_char, aux_double, array2)
10: for ( i = 0 ; i < NITER ; i++ ) {
11:   for ( j = 0 ; j < NITER ; j++ ) {
12:     if (i <= 1000)  p1.field = array[i % 4] + j;
13:     else  array2[i % 4] = p1.field;

14:     if (i > 2000)  aux_char = i % 20 + 48 + aux_char % 48;
15:     else  aux_char = i % 20 + array[i % 4] % 10 + 48;

16:     if (i > 1500)
17:       aux_double = array[i % 4] / (i+1) + aux_double;
18:     else  array2[i % 4] = (int) (aux_double / i*j) + (array2[ (i+j) % 4] + i*j) % 1234545;

19:     if (i*j > 10000) p1 = p2; else p3 = p1;
20:   }
21: }
```

Figure 4.12: Example of the kind of situations that the plugin can deal with.

best performance to their applications.

With these simple modifications, a programmer could speculatively parallelize a code, while the rest of transformations needed are transparently performed by the plugin and the compiler. Figure 4.11 resumes the code generation process performed by the plugin and the link with the TLS runtime system, transparent to the user.

4.5.3 Validation

In order to check the correctness of our plugin and the code that it generates, we have developed a battery of regression tests. These regression tests include more than 50 loops with one or more speculative variables, scalar variables, pointers, elements from multidimensional arrays, or elements from data structures. They also cover situations with speculative variables that have different types, and loops executing a number of iterations that are variable and defined at runtime. These regression tests are developed with the aim of covering possible situations that we can find in a source code,

allowing us to check the correction of the plugin before addressing real applications. One of these tests is shown in Fig. 4.12, where we check the correct operation of the plugin with speculative accesses over variables with different sizes, and speculative accesses to data structures, including assignments between entire structures.

As we will see in the following chapter, we have also tested the plugin with synthetic and real-word applications that are not parallelizable at compile time due to several data dependencies, requiring runtime speculative parallelization. Real-world applications includes the 2-dimensional Minimun Enclosing Circle (2D-MEC) problem [166], the 2-dimensional Convex Hull problem (2D-Hull) [39], the Delaunay Triangulation using the Jump-and-Walk strategy [55, 125], a C implementation of TREE [16], and 456.hmmer, a benchmark from the SPEC CPU2006 benchmark suite [82].

The plugin is able to speculatively parallelize the target loops in these benchmarks correctly. In other words, the plugin is not only able to recognize the new clause and transform the source code correctly, but it also generates executables that run in parallel, and produce the expected outcome.

4.6 Evaluation of our OpenMP `speculative` clause

The solution proposed in the previous sections performs all the transformations needed in a source code with the goal of parallelizing it speculatively, using the TLS runtime library. This is an automatic approach that aims to be an easier, and faster way to apply TLS than the traditional manual approach. However, the use of the new `speculative` clause plus the plugin would come to nothing if it led to a slower performance. Therefore, with our prototype we seek to achieve a performance at least as fast as the performance obtained by manually parallelizing the code. In this section, we will show the speedups achieved by our prototype with some synthetic and real-world applications, and we will also compare the performances obtained by both automatic and manually approaches. Moreover, we will quantify the number of lines required to parallelize these applications to estimate the programmability of both approaches.

4.6.1 Evaluation methodology

Firstly, before showing the results of the experiment, we need to describe how the experiment is designed, which hardware is used, and how experiments are run to obtain the results that we will show later. Moreover, we also list which measures are covered in the evaluation, and which synthetic and real-world applications are used to obtain the results.

Design of the experiment

Experiments have been designed to obtain the relative performance between the automatic and manual approach, i.e. to compare the performance obtained by our prototype with the performance achieved by the manual parallelization of the same source codes. These applications has been run on a 64-processor server, equipped with four 16-core AMD Opteron 6376 processors at 2.3GHz and 256GB 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.

Performances are measured using the speedup ratio, and therefore we will use it to compare both approaches. To calculate the speedups, we need the execution times, and thus, each code were instrumented to measure the execution times of the loops that are being parallelized. Both automatic and manual parallelization started with the same sequential code, and the same loop is parallelized by both approaches.

The experiment process has been carried out as follows:

1. Each sequential code is run to obtain (1) the execution time before parallelizing, and (2) the outcome expected for each code.

2. Each sequential code is the basis on which both automatic and manual parallelization are applied. Automatic parallelization is performed by using the new OpenMP `speculative` clause proposed in the preceding chapter. The programmer parallelizes the target loop using OpenMP, classifying the variables according to their accesses using the standard and the new clauses. Manual parallelization, on the other hand, was done by manually carrying out all the transformations needed to speculatively parallelize the code. The number of lines required in each parallelization is recorded with the aim of estimating and comparing the difficulty of the process.

   Each code is executed three times by using a script that automatizes the process, and it is calculated the average time for the following sets of processors: 1, 2, 4, 8, 12, 16, 20, 24, 32, 40, 48, 56, and 64.

3. Before considering these results valid, it is verified that each of the executions has finished correctly and output the expected outcome.

4. Finally, we calculate the relative performance by each set of processors using the execution times obtained.

Evaluation measures

Results obtained in the experiments are presented using the following measures:

- **Execution times:** We will use wall-clock times. Time will be measured in seconds, with a precision of six decimals, although results presented in this chapter are rounded to three decimals. The time needed to read the input set and the time needed to output the results have not been taken into account.

- **Lines of code:** This measure indicates an estimation of the difficulty in the parallelization process, as well as the time required. Obviously, this time depends on other factors, but it is necessary to recall that using the plugin also avoids many of them.

- **Speedup:** It is the ratio between the execution time achieved by a program in sequential, i.e. running in a single processor, and the execution time achieved by the program when running in parallel in several processors. The speedup achieved by an application with P processors is defined by the following formula:

$$S_P = \frac{T_1}{T_P}$$

  where T_1 is the execution time of application run sequentially, and T_P is the execution time of the application run in parallel with P processors.

- **Relative performance:** Comparison between the automatic and the manual approach will be shown using a ratio defined by the following formula:

$$R_P = \frac{T_P^{manual}}{T_P^{auto}} = \frac{S_P^{manual}}{S_P^{auto}}$$

  where P is the number of processors, T_P^{manual} is the execution time achieved by the manual parallelization of the application with P processors, and T_P^{auto} is the execution time achieved by the automatic parallelization of the application with P processors, using the plugin proposed. This ratio can also be calculated using the speedups, being S_P^{manual} the speedup obtained by the manual approach with P processors, and S_P^{auto} the speedup obtained by the automatic approach with P processors.

Dataset

The applications run in the experiments can be divided in synthetic benchmarks and real-world applications. We have designed three synthetic benchmarks, called *Complete*, *Tought*, and *Fast*, which aim to cover three possible situations that we can find

```
 0: #define NITER 6000
 1: int array[MAX], array2[MAX];
 2: struct card{ int field; };
 3: struct card p1 = {3}, p2 = {99999}, p3 = {11111};
 4: char aux_char = 'a';
 5: double aux_double = 3.435;
 6: int i, j;
       ...
 7: #pragma omp parallel for default(none) \
 8:    private(i, j) shared(array1, p2) \
 9:    speculative(p1, p3, aux_char, aux_double, array2)
10: for ( i = 0 ; i < NITER ; i++ ) {
11:    for ( j = 0 ; j < NITER ; j++ ) {
12:       if (i <= 1000)   p1.field = array[i % 4] + j;
13:       else   array2[i % 4] = p1.field;

14:       if (i > 2000)   aux_char = i % 20 + 48 + aux_char % 48;
15:       else   aux_char = i % 20 + array[i % 4] % 10 + 48;

16:       if (i > 1500)
17:          aux_double = array[i % 4] / (i+1) + aux_double;
18:       else   array2[i % 4] = (int) (aux_double / i*j) + (array2[ (i+j) % 4] + i*j) % 1234545;

19:       if (i*j > 10000) p1 = p2; else p3 = p1;
20:    }
21: }
```

Figure 4.13: *Complete* synthetic benchmark.

```
 0: #define NITER 1000000, MAX 100
 1: int array[MAX];
       ...
 2: #pragma omp parallel default(none) private(P)  speculative(array)
 3: for ( P = 0 ; P < NITER ; P++ ) {
 4:    Q = P % (MAX) + 1;
 5:    aux = array[Q - 1];
 6:    Q = (4 * aux) % (MAX) + 1;
 7:    array[Q - 1] = aux;
 8: }
```

Figure 4.14: *Tough* synthetic benchmark.

in codes that are not parallelized at compile time. The *Complete* benchmark, shown in Fig. 4.13, aims to test if the plugin is capable of parallelizing codes that includes speculative access of data with different sizes, and speculative access to data structures. When executing this loop in parallel, all the iterations lead to dependency violation.

```
 0: #define NITER 180000
 1: int array[MAX];
 2: int i,j,k;
 3: int spec1 = 0, spec2 = 0;
 4: int iter1, iter2;
      ...
 5: #pragma omp parallel default(none) private(i, k) shared(array, iter1, iter2) \
 6:   speculative(spec1, spec2)
 7: for ( i = 0 ; i < NITER ; i++ ) {
 8:   if (i == iter1) j = spec1;
 9:   if (i == iter2) j = spec2;
10:   for (k = 0; k < array[i % MAX] + j; k++) {
11:     if (k >= 179900)  spec1 = (k + array[ (i + k) % MAX] ) % NITER;
12:     if (k <= 1200)  spec2 = array[ i % MAX];
13:   }

14:   if (i == NITER - 1) spec1 = spec2;
15: }
```

Figure 4.15: *Fast* synthetic benchmarks.

As we will see, although the speculative execution finishes successfully, the speedup obtained will be extremely poor.

The *Tough* benchmark, depicted in Fig. 4.14, was designed to heavily test the robustness of the TLS runtime library and of the underlying consistency protocol used. All of its iterations perform a load and a store on the same speculative data structure, with almost no computational load on private variables. This situation adversely affects performance, although the number of dependency violations during parallel execution is relatively small (4.46%). We will test if our prototype is able to achieve similar results to the manual parallelization.

Finally, the *Fast* benchmark, shown in Fig. 4.15, has been designed to test the efficiency of the speculative scheduling mechanism. In this benchmark, only two of the 180 000 iterations (0.001%) lead to a dependency violation. Note that this single dependency is enough to prevent the compile-time parallelization of this loop. As with the *Tough* benchmark, our main goal is testing whether the prototype is capable of obtaining as fast performances as the manual parallelization.

We have also used real-word applications in the experiments. These applications are not parallelizable at compile time due to several data dependencies, requiring runtime speculative parallelization. These applications are the 2-dimensional Minimun Enclosing Circle (2D-MEC) problem [166], the 2-dimensional Convex Hull problem (2D-Hull) [39], the Delaunay Triangulation using the Jump-and-Walk strategy [55, 125], a C implementation of TREE [16]. and 456.hmmer, a benchmark from

the SPEC CPU2006 benchmark suite [82].

The 2D-MEC problem consists in finding the smallest circle that encloses a set of points. We have used the randomized incremental approach due to Welzl [166], that solves the problem in linear time. This algorithm starts with a circle of radius equal to zero located in the center of the search space. If a point lies outside the current solution, the algorithm defines a new circle that uses this point as one of its frontiers. It is interesting to note that points that laid inside the old solution may laid outside the new one. Therefore, all points should be processed again to check if the new circle encloses them. The solution can be defined by two or three points, and the algorithm is composed of three nested loops. We have speculatively parallelized the innermost loop, that consumes 45% of the total execution time, with a 10-million-point input set.

The 2D-Hull problem solves the computation of the convex hull (smallest enclosing polygon) of a set of points in the plane. We have used Clarkson *et al.* [39] implementation. The algorithm starts with the triangle composed by the first three points and adds points in an incremental way. If the point lies inside the current solution, it will be discarded. Otherwise, the new convex hull is computed. Note that any change to the solution found so far generates a dependency violation, because other successor threads may have been used the old enclosing polygon to process the points assigned to them. The probability of a dependency violation in the 2D-Hull algorithm depends on the shape of the input set. For example, if N points are distributed uniformly on a disk, the i-th iteration will present a dependency with probability in $\theta(\sqrt{i}/i)$. If points lie uniformly on a square, the probability of a dependency will be in $\theta(\log(i)/i)$.

We have compared the performance of both automatic and manual approaches of the 2D-Hull using three different, 10-million point input sets. The first one, *Kuzmin*, is an input set that follows a Gauss-Kuzmin distribution, where the density of points is higher around the center of the distribution space. This input set leads to very few dependency violations, since points far from the center are very scarce. The *Square* and *Disc* input sets are uniform distributions of points inside a square and a disc, respectively. It is easy to see that the *Square* input set leads to an enclosing polygon with fewer edges than the *Disc* input set, thus generating fewer dependency violations.

Following real-world application is the randomized incremental construction of the Delaunay Triangulation using the Jump-and-Walk strategy, which was introduced by Mücke *et al.* [55, 125]. This incremental strategy starts with a number of points, called anchors, whose containing triangles are known. The algorithm finds the closest anchor to the point to be inserted (the *jump* phase), and then traverses the current triangulation until the triangle that contains the point to be inserted is found (the *walk* phase). After this location step, the algorithm divides this triangle into three new triangles, and then updates the surrounding edges to keep the Delaunay properties. This

local modification to the current Delaunay solution may lead to dependency violations, since other threads may have traversed the old solution while trying to add new points. The expected amount of dependency violations generated by the Delaunay Triangulation depends on the number of processors and the length of the traversing path. It is easy to see that, the shorter the distance between the closest anchor and the point to be inserted, the fewer triangles that are visited in the walk and the smaller the probability of a dependency violation. This fact suggests that the algorithm should work with many anchors. However, the bigger the number of anchors, the more distance comparisons have to be performed to find the closest anchor to our point, thus degrading sequential performance. Our implementation uses a number of anchors that represents a good balance between these effects for the input size used. Our implementation is composed by two loops: The first one builds a Delaunay Triangulation of the first 5 000 points, that will be used later as anchors, while the second loop inserts all the remaining points (up to one million). We have speculatively parallelized this second loop.

The TREE problem [16], unlike the previous three applications, does not suffer from dependency violations, but it is still not parallelizable in compile time because it has dependency structures that are dependent on the input data. We have parallelized the loop that iterates over the bodies and computes the forces on them, which has more than 150 code lines. Compilers also find hurdles in several sum and maximum reductions contained in the loop, which ATLaS detects and handle properly. We have run this benchmark with a 4096-body input set.

Finally, 456.hmmer is a benchmark from the SPEC CPU2006, which performs an analysis of a protein sequence using profile hidden Markov models (HMMs). We have parallelized the outermost loop in function `P7Viterbi()`, located in the source file `fast_algorithms.c`, which consumes 90% of the total execution time with any of the workload sets defined by SPEC CPU2006. Although the speedup achieved is under $1\times$, we include the results for this benchmark as a demonstration of the capabilities of the compile-time system, which is able to seamlessly process such a complex code as 456.hmmer. The low performance obtained is due to the large number of dependency violations, which makes the runtime system unable to achieve better results.

4.6.2 Performance results and programmability

Figure 4.16 shows the speedups achieved using the proposed OpenMP `speculative` clause with the synthetic benchmarks described in the previous section. The parallel execution of the *Fast* benchmark in our shared-memory parallel system returns a maximum speedup of $44.5\times$ with 64 processors, while showing an efficiency of more than 90% when dedicating up to 32 processors for this task. These results indicate

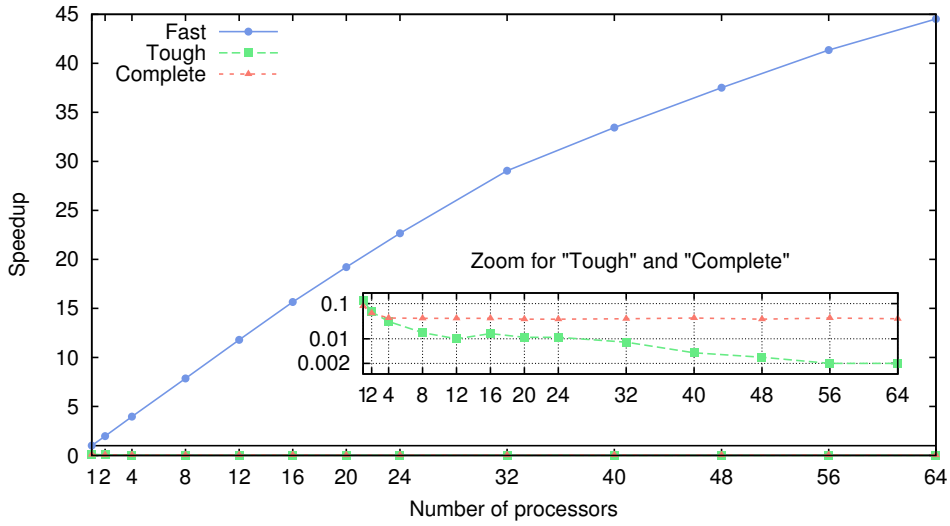

Figure 4.16: Speedups achieved using the `speculative` clause with synthetic benchmarks. Results are shown by number of processors.

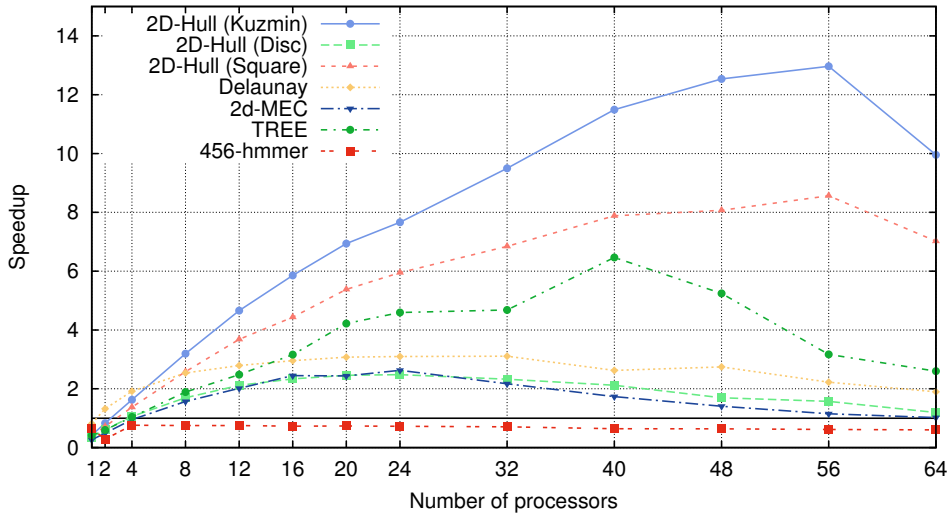

Figure 4.17: Speedups achieved using the `speculative` clause with real-world applications. Results are shown by number of processors.

that the overhead due to the runtime speculative library is negligible.

The two other synthetic benchmarks achieve expected results, with extremely poor speedups, which are justified by their characteristics. Both synthetic benchmarks, *Tough* and *Complete* are not designed to obtain speedup, but to test robustness and the wide range of situations that the system is able to face, respectively.

Figure 4.17 shows the speedups achieved using the new clause with some real-world applications. For the 2D-MEC benchmark, our solution achieves minor speedups, with peaks of $2.6\times$. Although these are not big figures, the manual use of the TLS library to parallelize this application requires more than ten hours of a very specialized work, while our compile-supported solution simply requires to declare as `speculative` the variables that hold the solution found so far.

In the case of 2D-Hull, as we have described above, results depend on the input set. Performance varies from a $2.4\times$ speedup with the *Disc* input set, which causes a huge number of dependency violations, to a $13\times$ speedup with the *Kuzmin* input set, which leads to fewer violations. As with the previous benchmark, the parallelization of the loop with our solution is straightforward, while the manual changes needed to use the TLS runtime library needs more than thirty hours of carefully replacing all loads and stores with function calls and changing the loop structure to support thread scheduling.

Delaunay's execution produces a high number of dependency violations, which affects the speedup. Delaunay achieves a peak performance of $3.1\times$ speedup. The programming effort to obtain a speculative version for this benchmark is very similar to the one needed for 2D-Hull, while the `speculative` clause reduces this effort vastly.

TREE obtains a peak of $6.5\times$ speedup. This benchmark is characterized by the presence of reductions over sum and maximum operations that involve `speculative` variables. These situations are easily resolved by our proposed clause, while handling them manually requires more than ten hours of programming effort.

Finally, the runtime system is not able to improve the sequential performance of 456.hmmer, due to large number of dependency violations in the loop parallelized. Without manually transforming this loop to break these dependencies, is not possible to achieve better results than the $0.6\times$ speedup achieved. In any case, a manual parallelization of this loop requires more than twenty hours, whereas the `speculative` clause frees the programmer from this task.

Automatic parallelization moves the workload from the programmer to the compiler. This is a great deal if the performance achieved by the automatic approach is as good as the obtained by the manual one. In Fig 4.18 we display the relative performance of both automatic and manual approaches for the real-word applications, whereas Table 4.1 shows the numerical differences, also including the syn-

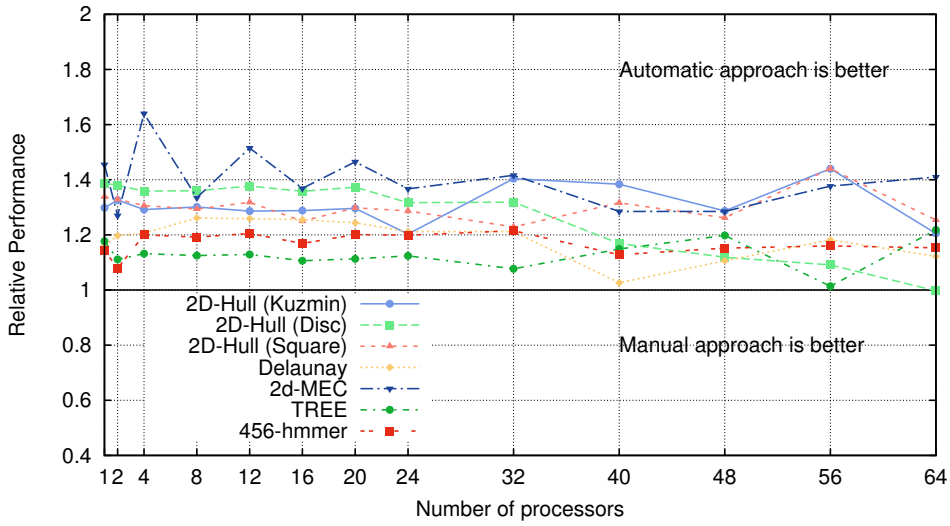

Figure 4.18: Relative performance of the automatic vs. manual approach with real-world applications.

Application	Relative performance by number of processors							
	8	**16**	**24**	**32**	**40**	**48**	**56**	**64**
Complete	1.323	1.398	1.208	1.204	1.280	1.184	1.399	1.275
Tough	1.063	1.087	1.132	1.209	0.985	1.188	1.168	1.044
Fast	1.078	1.063	1.040	1.051	1.041	1.035	1.032	1.010
2D-Hull (Kuzmin)	1.301	1.288	1.203	1.404	1.384	1.287	1.440	1.205
(Disc)	1.360	1.358	1.317	1.319	1.169	1.119	1.091	0.997
(Square)	1.294	1.250	1.287	1.229	1.316	1.262	1.440	1.254
Delaunay	1.261	1.255	1.212	1.212	1.026	1.106	1.182	1.122
2D-MEC	1.335	1.369	1.367	1.416	1.285	1.285	1.377	1.410
TREE	1.125	1.106	1.124	1.077	1.149	1.198	1.014	1.218
456.hmmer	1.192	1.169	1.199	1.215	1.128	1.152	1.160	1.153
Geom. Mean	1.219	1.211	1.205	1.231	1.158	1.178	1.202	1.150

Table 4.1: Runtime performance of the code generated by the plugin versus the runtime performance of the code parallelized manually.

thetic benchmarks. 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. The performance achieved by the applications parallelized using the `speculative` clause is around 20% faster than the performance scored by the manual parallelization on geometric average. The reason is that the

Application	Number of lines		TLS-related facts		
	Auto	Manual	#Variables	#Loads	#Stores
Complete	1	60	5	6	8
Tough	1	33	1	1	1
Fast	1	43	2	3	3
2D-Hull	1	136	1	20	24
Delaunay	1	186	1	30	34
2D-MEC	1	50	10	3	6
TREE	1	42	4	4	5
456.hmmer	1	72	1	11	12
Average	1	77.75	3.1	9.8	11.6

Table 4.2: Number of lines required in both automatic and manual approaches, together with the number of speculative variables, and the amount of speculative loads and stores required to be replaced with function calls to the TLS library.

manual transformation of the source code may prevent the application of certain compiler optimizations. In contrast, our automatic transformation system works with the GIMPLE intermediate representation, after the first phases of the compiler have been triggered.

Regarding programmability, the use of the proposed clause dramatically reduces the number of lines required in comparison with the former, manual way of parallelizing a code using the TLS library, as Table 4.2 shows. Parallelizing a code with the proposed `speculative` clause only requires one line of code (the modified OpenMP pragma), while parallelizing the same code manually requires tens to thousands of new lines, depending on the number of accesses to speculative variables. Such reduction in the number of required lines is not the only advantage. Parallelizing the code with the plugin only requires classifying the variables within the loop according to their usage, whereas the manual alternative is not only a hard, error-prone task, but also requires an in-depth knowledge of the TLS library.

4.7 Conclusions

In this chapter, we propose a compile-time system that automatically adds the code needed to handle the speculatively parallel execution of a loop, and implements a new OpenMP clause, called `speculative`, that labels those variables that may lead to a dependency violation. We have used the plugin mechanism provided by GCC to support the new OpenMP clause. Using this clause, programmers can point out the speculative variables, not needing to know anything about the speculative parallelization model. In order to parallelize a code, programmers are only required to add

one line (the OpenMP pragma plus the `speculative` clause), instead of the significant amount of lines required by the manual parallelization, which depends on the number of accesses to speculative variables. The parser detects the new OpenMP clause proposed, and the plugin-based compiler pass performs automatically all the transformations needed to speculatively parallelize the loop.

In this chapter we have also resumed some of the knowledge acquired during the development of the plugin and the modifications carried out on the GCC compiler. Our aim is that the information gathered in this chapter will help other researchers to modify and extend the GCC compiler through the plugin mechanism.

Moreover, we have evaluated our compile-time system together with the runtime library developed by Estebanez, García-Yágüez, Llanos, and Gonzalez-Escribano [62, 69], obtaining significant speedups in applications that are not parallelizable at compile time because of the dependency between iterations. Comparing the performance obtained by the automatically generated codes with the manually parallelized codes, we have also found that the use of our compile-time system leads to better performance.

We expect that implementing this new clause in a mainstream compiler, together with the automation of the whole process of the speculative parallelization, will help Thread-Level Speculation to be mature enough for use in production. Moreover, adding speculative support to the OpenMP standard would greatly increase the number of loops that could be parallelized with this programming model. Our proposal would let to transform *any* loop into a parallel loop.

The work and the conclusions described in this chapter has been published in the following papers:

- A New GCC Plugin-Based Compiler Pass to Add Support for Thread-Level Speculation into OpenMP. Sergio Aldea, Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. Accepted in Euro-Par 2014. Volume 8632 of Lecture Notes of Computer Science. To appear.

- An OpenMP extension that supports Thread-Level Speculation. Sergio Aldea, Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. Submitted to IEEE Transactions on Parallel and Distributed Systems in April 2014.

- Una extensión para OpenMP que soporta paralelización especulativa. Sergio Aldea, Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. *Actas XXV Jornadas de Paralelismo*, Valladolid, Spain, September, 17-19, 2014.

Chapter 5

Speculative Parallelization Niches Detection and Variable Classification

We consider that the use of the OpenMP `speculative` clause defined, implemented and evaluated in the previous chapter simplifies to a great extent the parallelization of existing applications. However, to take advantage of this clause, two important tasks still depends on the programmer: the choice of the best candidate loop to be parallelized, and a classification of the usage of the variables of this loop.

Our aim is to help the programmer with both tasks. In this chapter, 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 analyzing how particular language constructs are used in real-world applications, and helps the programmer to parallelize the code, for example using OpenMP clauses, and more particularly, our OpenMP `speculative` clause proposed in the preceding chapters. It is important to highlight that this framework not only helps the programmer that wants to use speculative parallelization, but also any programmer looking for a parallelization of the code in a shared-memory architecture.

To show this 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.

5.1 Problem description

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 [31].

Fortunately, there exist different shared-memory parallel programming models that aim to facilitate parallel programming, being OpenMP [33] 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 from 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* [96]. 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 for a given input set 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. Further analysis may be required to ensure that, after parallel execution, final values stored in private variables meet sequential semantics. Figure 5.1 shows an example of such parallelizable loop. If a single variable is found that does not fit into 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, regardless of the use of TLS programming techniques, this *dependency 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.

```
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
}
```

Figure 5.1: Example of a loop with private (`i` and `v[]`) and read-only shared (`a[]`) data structures.

In this chapter, 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 transforms the source code of a C application into a single XML [23] 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 [54], works as follows:

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 [18] 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 [82]. 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.

Summing up, the main contributions of this chapter are the following:

- We present a framework that 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 the following section, and helps to close a gap described in the literature, since traditional profilers focus primarily on functions and inner loops [124].

- 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 dependency violation among iterations.

- The flexibility offered by the XML representation allows extending this framework for other purposes, such as automatic source code optimization.

The rest of this chapter is organized as follows. Section 5.2 describes some related approaches. Section 5.3 introduces two problems that affect the parallelization of a source code: Granularity, and data dependencies between instructions. Section 5.4 describes the overall architecture of the framework developed, together with its components. To evaluate the proposed solution, we have performed a detailed analysis of the opportunities for parallelization in the C applications of the SPEC CPU2006 benchmark suite, that it is shown in Sect. 5.5. Finally, Sect. 5.6 concludes this chapter and summarizes our contributions to the literature.

5.2 State of the art

As it has been stated, when programmers parallelize source codes, one of the issues they have to face is the possible dependencies between iterations, which may avoid the correct parallelization of a given loop. In this section we describe what is the meaning of data dependencies, and how these dependencies are determined by the accesses to the variables used in the loop iterations. This section also explores some of the different approaches to discover parallelism, and the importance of a correct selection of the loops to be parallelized. Moreover, it also gathers different ways to represent source codes using XML, which can be used to extract valuable information from

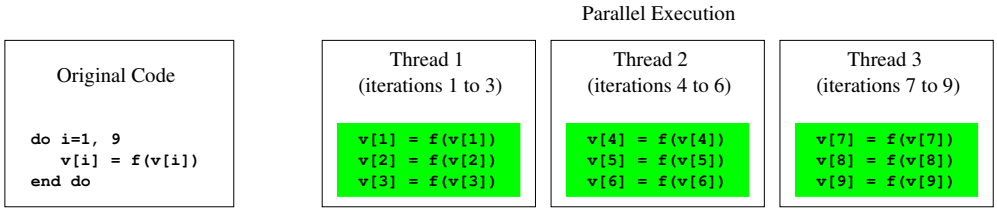

Figure 5.2: Parallel execution of a loop without dependencies using three threads. Because each one of the iterations is independent, compilers can execute them in parallel.

them. Finally, we will introduce Cetus, the source-to-source transformation compiler that we have used as a starting point in the development of BFCA.

5.2.1 Parallel execution and variable classification

Nowadays, parallel systems are widely present in several areas thanks to the development of CMP (Chip Multi-Processor) architectures. However, parallel computation has been a classical problem for more than half a century. As Lamport said in a paper of 1974, in which he describes several methods for obtaining parallel execution of nested loops [104]:

> *"Any program using a significant amount of computer time spends most of that time executing one or more loops".*

Therefore, loops are the main target to apply parallelism. Different iterations of a loop can be executed at the same time by different processors, as Figure 5.2 shows. At a software-level, this code parallelization can be done manually or automatically. On one hand, there exist various parallel programming languages, parallel extensions and library functions, such as OpenMP [33], MPI [75], UPC [28], X10 [34], Chapel [31], or Fortress [152]. However, most of these solutions require a deep knowledge about the problem to parallelize, the underlying architecture and the communications library, while most of the code already developed that can benefit from these solutions are sequential. To avoid this problem, parallelization can be automatically achieved by compilers.

Automatic parallelization, on the other hand, can be done at compilation time, as we saw in Chap. 2, or at execution time (inspector/executor model and speculative parallelization). However, both options are closely attached to the presence of potential dependencies in the code.

5.2.2 Parallelism discovery and loop selection

A correct selection of loops to be speculatively 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 [141, 165]. Although extracting and using correctly this information has a performance penalty, and includes scalability problems [99], many works have shown that it is not pointless. For example, obtaining an optimal loop selection, Wang *et al.* [165] get speedups of 20% in SPEC2000 integer benchmarks, and Packirisamy [134] 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 [35, 84, 93, 109, 111, 167]. Johnson *et al.* [93] 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.* [111] estimate the parallel performance of each loop in terms of the probability and cost of speculation failure. As we will see, our proposal is not only suitable to detect speculative parallelization niches, but also reveals hurdles that may affect any kind of parallelization. Finally, POSH [109] 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 dependency [162, 170], the cost of re-executing iterations due to dependency violations [58], the Amdahl's Law [27], the different overheads of a parallel execution [57], the frequency of dependencies [157], or speedup estimations based on graphs [67, 165]. Our solution does not use theoretical cost models. Instead, it characterizes loop coverages using real executions and classifies based on the potential dependency 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 with this technique [99]. This 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, our solution not only points out which loops are better to be parallelized,

but also identifies hurdles presented in a code that may affect the parallelization.

5.2.3 Frameworks to transform source code

Our solution, called BFCA (BonaFide C Analyzer) takes advantage of the benefits derived from using XML [23]. With XML, we can directly represent, analyze and manipulate the program structure. As McArthur *et al.* [119] 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 [45, 120], exchanging [22, 85], differencing [41], program slicing [74], generation of UML models [144], addressing source code [42], source code transformations [46], or fact extraction [43, 44].

We can list other benefits of using XML to represent source codes, which are the following:

- XML is simple and extensible. Programmers have flexibility to create tags and attributes.
- Support for multiple query languages, such as XQuery or XPath, being the last used in the development of BFCA.
- Support for complex transformations (XSLT, SFX, or TextReader).
- Support for cross-referencing.
- Easy parsing with all existing XML tools, as well as easy exchange between CASE tools.
- XML is well suited for representing hierarchical data. The structure of the source code is explicitly represented by the nesting of elements in the XML document.
- Support for a document format as well as data format. This duality is widely used to both save the structure of a source code, and its formatting information.
- Support for tasks that cannot be done on plain text representations, such as counting all the occurrences of a syntactic construct, finding the number of

functions called by a function, or counting the number of functions calling a particular function.

- Large interest from the open source community. XML is widely used and accepted as standard.

There are some works that use XML to represent source code in order to extract some information. One of the first XML representations of source code is JavaML [11], 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 [3], srcML [114], XSDML [118], JaML [64] and PALEX [112]. 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 [115] and Zou's and Kontogiannis' proposal [174]. Other data, including the flow information, are implicitly stored. This feature allows BFCA to save memory resources, unlike other approaches, such as the XREF-model [10], which stores the relations across multiple files; ACML [74], 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 [4], and Putro and Liem [140], 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 [119], or Cordy's proposal [48]. 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 [154], 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, in order to avoid the low scalability associated with the bottom-up parsers, and presented in some proposals such as Power's and Malloy's [137], BFCA follows a top-down approach. Moreover, BFCA is not graph-based, as GraX [61], GXL [85], or Wagner's *et al.* [164] 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.

Cetus

As we have seen, using XML to represent source code has several advantages. In order to obtain the XML representation of a source code, instead of starting from scratch, we decided to take benefit of an existing software, called Cetus. Cetus [12, 54] is a compiler infrastructure written in Java for source-to-source transformation of C programs developed by Purdue University (Indiana, USA). It is under a modified Artistic License [29], a license similar to the Perl Artistic License, which allows distribution of Cetus' modifications, being public these changes and known by the Purdue University.

Cetus provides several functions, such as auto-parallelization of loops through private and shared variables analysis, and automatic insertion of OpenMP directives [33]. Cetus uses ANTLR version 2 to build a recognizer and interpreter of C grammar. This choice is not casual, because ANTLR has also its origins in Purdue University. Using this grammar, 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.

Cetus provides an API that is very useful to manipulate nodes in the Intermediate Representation, and to get information about nodes. Cetus and its API will be the starting point of the developed framework, leveraging its IR to create the XML representation of the source code. Appendix A gathers a more detailed description of Cetus.

After briefly describing the relevant literature, we will now discuss which factors affect the parallelization.

5.3 Granularity and data dependencies

In order to parallelize loops we mainly face with two problems: Granularity, and dependencies between instructions.

The first problem, granularity, comes from the start-up and synchronization overhead needed to execute a parallel loop. If a loop has not enough work to compensate the cost of a parallel launching, the loop is not parallelized. Therefore, a compiler should tend to parallelize the outer loops rather than the inner loops, with the aim of minimizing the cost associated to synchronization. Reducing the frequency of synchronization is equivalent to increasing the granularity of the parallel iterations.

The second problem is the existence of data dependencies. Formally, a dependency is a relation between two statements of the program, which ensures that the meaning of the program is preserved. A dependency defines a constraint that determines the execution order of the instructions, and thus the transformations that can be

applied to the code.

This problem is not new. In 1966, Bernstein already defined three conditions [19] to consider parallelizable two sequences of instructions or successive iterations of a loop. These conditions are defined in a mathematical form, and could be summarized as follows. Two iterations I_1 and I_2, being I_1 executed before I_2 according to sequential semantics, can be safely executed in parallel if:

1. iteration I_2 does not read a location that has been written by iteration I_1. If this occurs, this situation is called *RAW dependency* (Read After Write) or *true dependency*.

2. iteration I_2 does not write into a location that has been read by iteration I_1. If this occurs, this situation is called *WAR dependency* (Write After Read) or *anti dependency*.

3. iteration I_2 does not write into a location that has been written by iteration I_1. If this occurs, this situation is called *WAW dependency* (Write After Write) or *output dependency*.

If iterations in a loop violate some of the Bernstein's conditions, the loop is not parallelizable. In other words, a compiler only transforms a sequential loop into a parallel one when it is able to check that none of the Bernstein's conditions are violated.

There are two types of dependencies, data and control dependency. A control dependency is caused by the control flow. For example, in the case of an *IF* condition, there is a control dependency between the statement that defines the condition and every statement inside the *IF* block, which cannot be executed before the condition has been checked. This paper does not focus on this type of dependency.

A data dependency ensures that data is read and written in the correct order. For example, in the following code, the order in which statement in line 1 (S_1) statement in line 2 (S_2) are executed is indifferent. However, statement in line 3 has to be executed after S_1 and S_2.

```
1  A = 2
2  B = 1
3  C = A + B
```

Allen and Kennedy, in their book [7], offer a formal definition of this concept:

> *"There is a data dependency from statement S_1 to statement S_2 if and only if*

1. *both statements access the same memory location and at least one of them stores into it, and*
2. *there is a feasible run-time execution path from S_1 to S_2".*

In order to detect data dependencies in source codes, moderns compilers perform several tests. A large number of data dependency test has been proposed (see [138] for an evaluation of them). The most well-known tests are, in order of increasing complexity and computation cost, GCD (Greatest Common Divisor) [15], Banerjee test [15], I-test [100], Omega test [139], and Polyhedral model [17].

The dependencies between iterations are determined by the dependencies between variables within these iterations. We can classify each variable in terms of these dependencies. Our proposed solution address this issue.

5.4 Solution proposed: BFCA

In the preceding sections we have exhibited the importance of an accurate selection of the loops to be parallelized. With the aim of solving this problem, we propose a prototype that combines static analysis of source code with profiling information. As we will see, this prototype provides useful information that will help us in the detection of loops that are good candidates for parallelization. The prototype outputs information regarding all loops in an application, including loop coverage, loop suitability for parallelization, a classification of all variables inside loops based on their accesses, and other hurdles that restrict parallelization.

5.4.1 Requirements specification

First of all, we will formally define the requirements that the proposed system has to fulfill:

Functional requirements

[FQR-01] The system has to convert the Intermediate Representation (IR) of Cetus into an XML tree.

[FQR-02] The system has to recognize all the node types proposed by the Cetus IR which has a corresponding language element in C.

[FQR-03] The system has to annotate each *FOR* loop with its length in number of code lines (this is a functionality not offered by the original software of Cetus).

[FQR-04] The system has to use the opportunities offered by XML technologies in order to seek and locate all the code features required.

[FQR-05] The system has to classify variables regarding their usage in the code, in *private*, *shared*, and *speculative*. This classification will be done following the guidelines that will be described in Sect. 5.4.6.

[FQR-06] The system has to report which variables are either written or read, or both written and read in the context of each *FOR* loop.

[FRQ-07] The system has to classify *FOR* loops in terms of the variable usage.

[FQR-08] The system has to report if a *FOR* loop has other loops nested within.

[FQR-09] The system has to report if a *FOR* loop is affected by *GOTO* or *BREAK* statements that affect to the regular execution flow of the code.

[FRQ-10] The system has to report about the percentage of the *FOR* loops affected by hurdles to its parallelization, such as the pointer usage, memory management instructions, I/O functions and static variables.

[FRQ-11] The system has to detect static variables affecting a *FOR* loop in any nested *FOR* loop and any function call in any level.

[FRQ-12] If a *FOR* loop is affected by a static variable, the system has to provide the trace until the static variable. This includes nested *FOR* loops and function calls. For each function of the path, the system has to provide the source file and the line number.

[FRQ-13] The system has to be able to handle profiling information provided by the Intel compiler in an XML format (this requirement takes advantage of the XML documents generated by the Intel compiler, and avoids to adapt the output of others profiler tools to our interests).

[FQR-14] Each *FOR* loop has to be annotated with the inclusive time (percentage and absolute), the exclusive time (percentage and absolute), and the length of the loop in lines of code.

[FRQ-15] For each *FOR* loop of the path, the system has to provide the source file, the line number, and the inclusive and exclusive time.

[FRQ-16] The system has to report about the weight of each *FOR* loop in a source code in terms of execution time, and lines of code.

[FRQ-17] The system has to report about the coverage (in percentage of execution time) of the different kind of *FOR* loops.

[FQR-18] The system has to be able to rebuild a C code throughout an XML tree which encodes it, being the rebuilt code functionally equivalent to the original C code.

[FRQ-19] The system has to provide its output in a human legible format.

Non-functional requirements

[NFR-01] The system has to inform the users about the right way to execute it upon an incorrect launch by the user.

[NFR-02] The system has to provide a result in a time proportional to the size of the input.

[NFR-03] The system has to be able to be executed from any path.

Constraints

[CNS-01] The system will not create XML trees from source codes written in languages different than C.

[CNS-02] The system will not recognize other profiling information that the provided by the Intel compiler.

[CNS-03] The system will only work with C programs.

5.4.2 Framework architecture overview

Figure 5.3 shows the architecture of the BonaFide C Analyzer (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 [54], 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

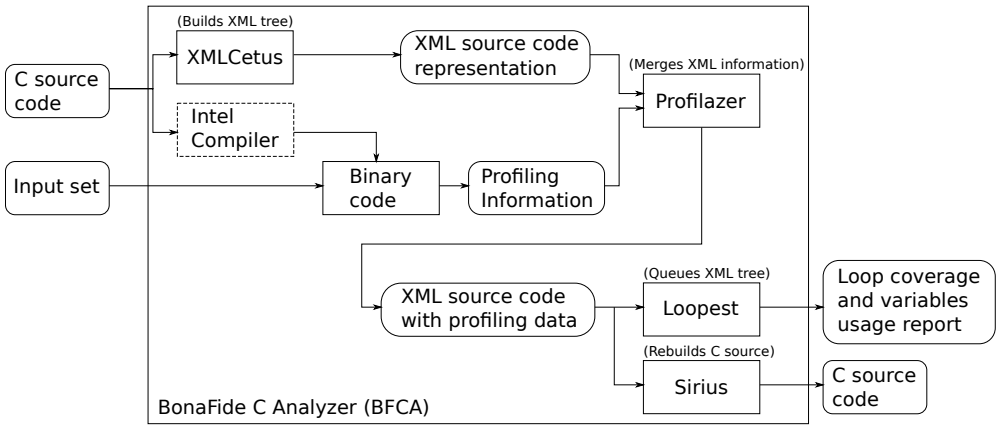

Figure 5.3: Architecture of the BonaFide C Analyzer.

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 are able to 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[1]. This differs from what other profilers do, such 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 in order 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 instead.

A second module, called Profilazer, receives the XML file generated by XML-Cetus 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.

[1]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.

This augmented XML tree is the input for 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 dependency 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. This feature will be covered in the following chapter.

Finally, we have developed a tool that converts the XML tree back into C language, in order to check the correctness of the process, and to take advantage of the possible augmented codes produces by Loopest. This module that performs such transformation is 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. 5.5.3, the development of the XPath queries needed for the present functionalities of BFCA is much simpler than directly modifying Cetus for the same purpose.

To sum up, we will synthesize below the functionalities of each module:

1. **XMLCetus:**
   - *Input:* A C source code file.
   - *Process:* It creates an XML file from the Cetus IR.
   - *Output:* An XML file that represents the C code.
2. **Profilazer:**
   - *Inputs:* It has two inputs: the XML file created by XMLCetus, and the XML file provided by the Intel compiler, with execution times of loops in the code.
   - *Process:* It augments the entry XML file that represents the C code, adding the information provided by the XML Intel profiling file.
   - *Output:* An augmented XML that represents a C code, with execution times annotated as attributes in every loop.

3. **Loopest:**

   - *Input:* The augmented XML file created by Profilazer.
   - *Process:* Using XPath queries, it analyzes the input XML file to classify variables usage in loops, and to extract other useful information.
   - *Output:* An exhaustive analysis report with the classification of each variable in each loop in terms of private, shared, and speculative, among other functionalities that will be later described, such as recognition of pointer arithmetic, memory management, I/O functions usage in loops, or relevance for each loop in terms of execution time.

4. **Sirius:**

   - *Input:* An XML file that represents a C code.
   - *Process:* It generates a correct C file using the entry XML file.
   - *Output:* A new C source code file, functionally-equivalent to the original C source code.

5.4.3 Key aspects of our proposal

BFCA has some key aspects that are worthy of remark. It has a helpful value as source code analyzer, extractor of features that characterize source codes, and as a reference guide to programmers in the loop-based parallelization process:

- BFCA is focused on loop-level parallelism.
- BFCA profiles the execution time for each *FOR* loop. The coverage of loops needs to be known in order to determine which loop deserves to be parallelized. Parallelizing low-demanding loops incurs in overwhelming the application due to OpenMP thread management overheads, which leads to slowdowns. Therefore, it is completely necessary to select those loops which are profitable in terms of execution time.
- With the XML representation of the source code, BFCA handles the structure of each *FOR* loop, and how these loops are nested. This static representation is combined with the execution times of the *FOR* loops, and provides programmers useful information to decide which loops could be suitable to be parallelized. This kind of information is not provided by traditional profilers, as [124] pointed out.

- It classifies *FOR* loops in different categories with respect to their suitability for parallel execution and taking into account the existence of potential dependency violations. For this purpose, BFCA classifies variables accessed into the *FOR* loops in private, read-only shared and speculative variables, this is, variables that can lead to any dependency violation. Extracting these data, BFCA is able to estimate an upper bound of the degree of parallelism that could be extracted with compile-time techniques, and the potential degree of parallelism which may be possible to extract with runtime techniques, such as speculative parallelization.

- As a dependency violation is not the only impediment to parallelize a loop, our system is also able to perform an in-depth analysis to characterize the *FOR* loops with respect to different hurdles that may also affect parallelization. Some of these situations are the use of pointer arithmetic and variables, and memory management function calls, which affects 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 to meet sequential semantics.

- BFCA does not only provides a loop classification, and the number of loops that falls in each category described above. It also returns a detailed classification of all variables inside all *FOR* loops in terms of their definition and use. Programmers will find this information extremely useful, both to decide which loops is more profitable to be parallelized in first place, and to guide the parallelization process.

- As a result of the use of XPath capabilities, BFCA could be easily customized and extended to generate a detailed report on the existence of these and other characteristics in the code being examined. BFCA is currently able to identify *FOR* loops affected by *GOTO* and *BREAK* statements which may affect the execution flow, as well as correctly handle both static and global variables. Other situations could be detected by adding appropriate rules.

- Using the acquired information, BFCA could augment the XML tree with additional statements and directives. This would allow inserting OpenMP directives, or even special code constructs to handle the speculative parallelization of promising loops. Then, BFCA will transform the resulting, augmented code back to C. We will further explore this possibility in the following chapter.

After briefly reviewing the overall framework architecture and the key aspects of our proposal, we will now discuss each one of the system modules in more detail. The

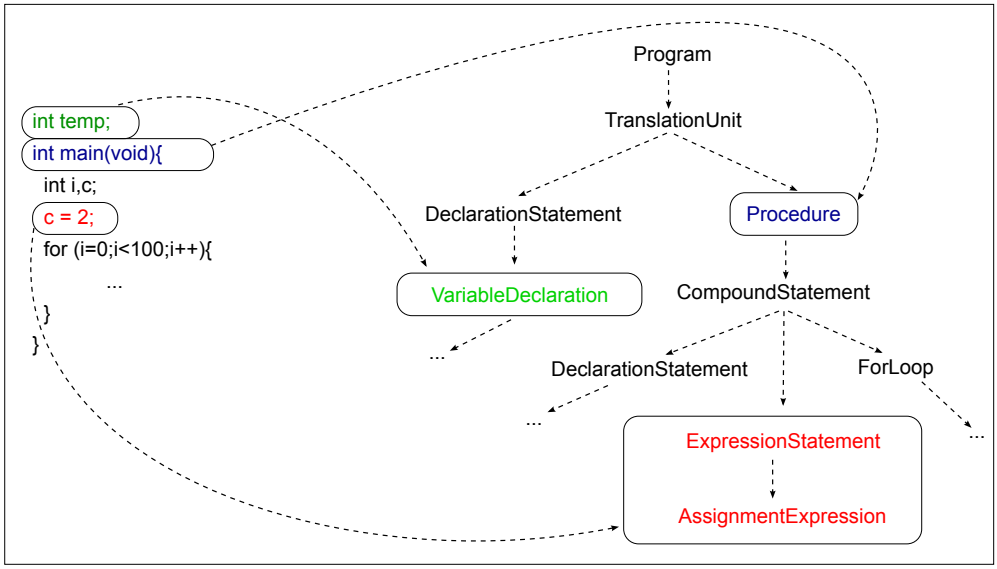

Figure 5.4: Cetus IR tree structure example. Each part of the source code corresponds with a node in the tree. A "TranslationUnit" is a file containing source code.

purely technological aspects of the development of XMLCetus, Loopest and Sirius were addressed in our M.Sc. thesis [5]. However, the version of BFCA presented in this dissertation is a modification of the version presented in the cited work, so here we will detail the changes performed.

5.4.4 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 [54] (see also Sect. A.1). Cetus 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 5.4 shows an example of Cetus IR from a C source code.

Although Cetus is a powerful tool, adding new functionalities to it requires an in-depth knowledge of Java, Cetus Intermediate Representation, 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 mod-

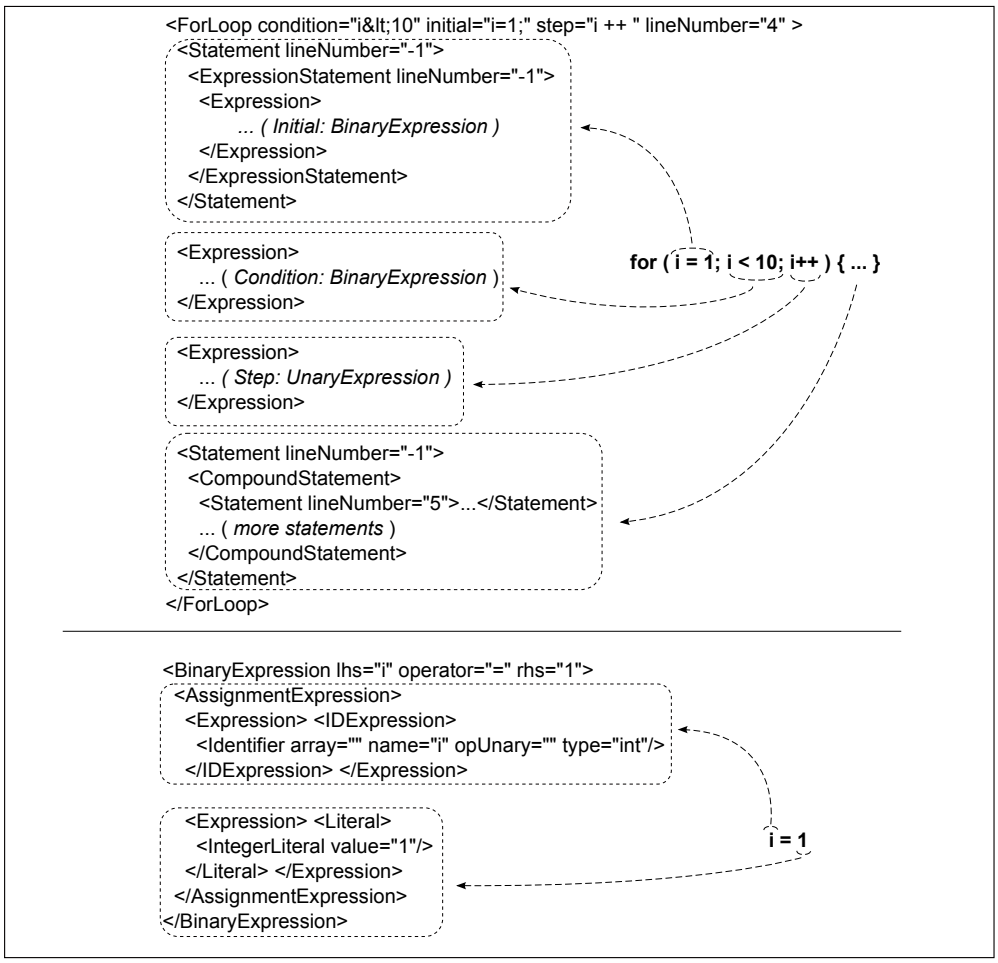

Figure 5.5: XML code generated by XMLCetus for a *FOR* loop statement (top), and the binary expression that represents the loop initialization (bottom).

ifications 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 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 repre-

sents the XML DOM tree, and it is printed into a new XML file. Figure 5.5 shows an example of this XML generated.

The following steps explain the transformation procedure from Cetus IR to XML DOM tree.

- Beginning with the first node, *Program*, XMLCetus recursively descends all its TranslationUnit children nodes, that represents the source code files passed to Cetus. The tree is traversed in preorder, depth-first search, getting the children of each node as a list of nodes. Through a casting operation, this 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. This is done with the help of a loop that traverses each node. This loop finds, with subsequent checks, the class the node is an instance of, using the *instanceof* Java operator. If one node can be 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 that represents the DOM tree, and it is printed into a new XML file.

5.4.5 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 an XML tree augmented with the inclusive and exclusive execution times of every *FOR* loop in the original source code. 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. 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.

Each of these times is annotated in two forms: As an absolute time, and as a percentage respect to the total execution time of the program. Moreover, Profilazer also

```
<ForLoop
  absTime="613804091"
  absTimePercent="10.0"
  condition="i&lt;num_moves"
  entryCount="1540780"
  initial="i=0;"
  length="104"
  lineNumber="8174"
  selfTime="474855179"
  selfTimePercent="7.7"
  step="i ++ ">
```

Figure 5.6: Example of ForLoop annotated after Profilazer.

annotates the number of times a loop is dynamically reached during the execution of the program. An example of a *FOR* loop annotated with execution times by Profilazer is shown in Fig. 5.6.

Adding profiling information to the XML document

In order to estimate the value of each loop in the code according to its execution time, we need a compiler or a tool that provides us profiling information. 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. Within the set of available options, the Intel compiler is the alternative that better satisfies our needs, since it provides an XML output, instead of generating a text file with the source file annotated with execution times, as Sun's profiler or OProfile.

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.

Intel's profiler information

XML output files generated by the Intel compiler are structured in two blocks. First, they have profiling information about each function in the source code. Then, and this is the information which we are interesting in, they have profiling information about

```
<loops reportlevel="1" >"
  <loop>
    <ticks_perc>0.0</ticks_perc>
    <ticks_abs>8169</ticks_abs>
    <self_perc>0.0</self_perc>
    <self_abs>7242</self_abs>
    <entry_cnt>70</entry_cnt>
    <function><![CDATA[in_check]]></function>
    <src_file>
      <name><![CDATA[sjeng.c]]></name>
      <line>252</line>
      <path><![CDATA[/home/seralde/CETUS/pruebasBenchmarks/458.sjeng/Fuente
          /sjeng.c]]></path>
    </src_file>
  </loop>
...
</loops>
```

Figure 5.7: Extract of the file generated by Intel Compiler.

each *FOR* loop. Figure 5.7 shows an extract of the generated XML file.

Each loop are annotated with its exclusive (*self*) and inclusive (*ticks*) execution time in ticks (clock rate) as an absolute value, and as percentage respect to the total. Moreover, each loop contains a reference to the function and the source file in which it is contained, with the line number. All this information is essential to unambiguously identify each loop, and thus, to correctly annotate loops in the XML document passed by XMLCetus.

Development details

Profilazer extracts execution times annotated in the profiling report generated by the compiler. For each loop in this report, it seeks the same loop in the XML file provided by XMLCetus. Once the loop is found, Profilazer adds new attributes to annotate its execution time.

To add profiling information to the correct *FOR* loop, it is necessary to identify each loop with its line number and source file. However, the version of Cetus used (1.1) does not preserve line number information in all situations. It means that matching between *FOR* loops in profiling reports and *FOR* loops in the XML representation is not possible, and hence, it is impossible to add profiling information to these loops. We have studied this problem and we found out that the problem is always caused by *include* statements within *ifdef* blocks, as it was already described in subsection "*Cetus known problems*" in App. A. Knowing the definitions applied in each benchmark, we eliminate the *ifdef* blocks, preserving or erasing the *include* statements in function

of the applied definitions.

Profilazer's internal operation is described below.

1. First, Profilazer builds two XML documents using the XMLCetus output, which represents the source code, and the XML file outputted by the compiler.
2. Profilazer extracts each loop of the profiling XML document through an XPath expression.
3. For each loop, Profilazer extracts execution times, entry counts, and file and line number in the source code.
4. As each loop in the XML profiling document matches with a single loop in the XMLCetus document which represents the source code, Profilazer searches for these loops using the line number and the file name.
5. Then, *ForLoop* nodes from the XML representing the source code are annotated with the execution times extracted from the profiling XML document.
6. Finally, the new XML document is printed into a file, with the same name than the original XML file created by XMLCetus plus the suffix "profiled".

In step two, Intel Profiler fails because sometimes it annotates the same loop twice or more times, with different values. The sum of these different values is often bigger than the total time of the whole source code, thus the sum of these values is not the correct execution time for a loop. This error is not documented by Intel, and the biggest value in each case is quite close to values extracted by other profilers for those same loops. Therefore, we decide to choose the biggest value in those cases in which the compiler annotates the same loops twice or more times.

As we stated previously, obtaining the relevance of each loop in terms of execution time is very useful to choose which loop is more profitable to parallelize. Besides this, an XML tree representing the source code, augmenting with profiling information, can be used to automatically parallelize a loop, either by introducing OpenMP directives, or by building heuristics to choose automatically the target loop. The first path will be covered in the following chapter. The second one is listed as potential future work in the Conclusions of this Ph.D. thesis.

5.4.6 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

```
// (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) ])
           )
```

Figure 5.8: 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.

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, to either insert automatic parallelization directives, or directives and functions that allow the speculative parallelization of the original source code. In this chapter we will cover the first two functionalities; the third one will be discussed in the following chapter.

Loopest relies on XPath capabilities to perform queries on the augmented XML tree returned by Profilazer. XPath syntax is easy to learn, and allows building 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. 5.8 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. 5.8 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.

Variable classification in the context of a *FOR* loop

In order to develop BFCA it is necessary to precisely define a set of rules to classify variables into *private*, *shared* and *speculative*. There are a couple of considerations regarding this classification. First, a variable passed by reference to another function will be considered as a write. Second, a variable passed by value to another function will be considered as a read.

Variable classification is described below. Table 5.1 resumes this classification.

a) Omitted variables There are some variables that are ignored in this classification:

1. From the point of view of Loopest, variable declarations without initialization are ignored in the dependency analysis. These variables are not taken into account until they are used, read or written.

2. The numerical value of variable memory addresses will be considered constant.

Type	Description
Private	• Variables that control the execution of a loop. • Variables that are always written in an iteration before being read, and are not read after the loop execution. • Variables that are only written and read after the loop, or are written before being read. • Data structures that only contains private variables.
Shared	• Variables that are only read. • Variables that are involved in the main loop control (but they are not the control variable), as long as they are read-only variables. • Data structures that all their elements are shared. • Global variables that are only read inside the loop.
Speculative	• Every variable that is not private or shared.

Table 5.1: Simplified variable classification in terms of their accesses.

b) Private variables The variables that fulfill the following requirements are classified as *private*:

P01 Variables that are **always** written in an iteration **before** being read, and are not read **after** the loop execution (in temporal terms). This include variables that control inner loops. If the variable is read before the outermost loop, it does not affect.

[P01-Note] Pay attention to GOTOs or nested loops that affect the execution flow.

P02 Variables that are only written and not read **after** the loop, or are written before being read (output value of the variable is overwritten). It does not affect if variables are read before the loop.

[P02-Note] Pay attention to GOTOs or nested loops that affect the execution flow.

P03 Data structures that **only** contain private variables. If any variable is not used, not private, shared or speculative, the data structure continues being private.

c) Shared variables The variables that fulfill the following requirements are classified as *shared*:

C01 Variables that are **only read**. **Exception**: The variable that controls the loop, being only read inside the loop body, will be considered as private.

C02 Variables that are involved in the main loop control (but they are not the control variable), as long as they are not modified during the loop execution, i.e., being read-only variables. If these variables are not even read, they should be declared as shared, because they are always read at least one time in each iteration in the loop control.

C03 Data structures that **all** their elements are shared.

C04 Global variables that are **only** read inside the loop.

d) Speculative variables In general terms, a `speculative` variable is every variable that is not private or shared. Following rules discriminate all possible cases.

E01 Variables that in a same iteration are **always** written before being read, and are read outside the loop.

E02 Variables that are **only** written inside the loop, and also used outside the loop.

E03 Data structures that it is not possible to determine whether they are private (rule P03) or shared (rule C03) variables at compile time.

E04 Variables that are passed by reference to a procedure.

E05 Variables that, in the context of an iteration, are first read and then written. In the case of statements as `a2 = f(a2)`, the implied variable is speculative. The variable value is first read in the right-side and then written.

E06 Variables that are **only** used inside a loop with expressions that have the following form are variables that support speculative reductions:

- `VAR = VAR + f(x)`
- `VAR = VAR * f(x)`
- `MAX(VAR,f(x))`
- `MIN(VAR,f(x))`

E07 Global variables (also data structures) that are written inside the loop, because there is not an easy way to guarantee that this value will not be used after the loop.

These rules help us to identify which loops are more profitable to be parallelized, including which loops might be suitable to be parallelized by speculative means.

How Loopest works

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[2] 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 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. Summarizing the classification of the previous pages, 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 are 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 category 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 tree, 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

[2]This feature has been added with the help of the *ListUtils* package, provided by Apache Commons [90], that allows applying operations such as *union* or *intersection* over sets of variables.

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 [37]. This feature will be covered in the following chapter.

"Well-formed" *FOR* loops

Among the features that Loopest extracts from source codes, the characterization of *FOR* loops into "well-formed" loops is one of the most important in order to choose a loop to be parallelized. A "well-formed" *FOR* loop is a loop that satisfies the following properties:

- It has a single control variable.
- All three fields of the *FOR* structure (initialization, conditional evaluation, and increment) are being used,
- It does not perform any change to the control variable inside the loop body.

This kind of *FOR* loops are more feasible to be parallelized, mostly because the iteration space is known in advance, and hence, Loopest reports whether each loop fulfills these requirements.

Extracting code features

As we have stated above, Loopest is able not only to classify variables within loops as private, shared, or *speculative*, but also to provide more useful information about pointer arithmetic usage, memory management functions, I/O functions or even static variables affecting a loop in any nested level of loop or function. Moreover, Loopest is also capable of leveraging profiling information provided by the subsystem Profilazer to determine the importance of each loop in terms of execution time.

From the point of view of Loopest, extraction of new features, as pointer arithmetic usage in a loop, only requires new XPath queries. Adding profiling information into reports is even more straightforward since Loopest only needs to extract the values of two attributes (exclusive and inclusive time) once it detects a new loop. This shows one of the main properties of Loopest: Extensibility. It is relatively easy to modify it in order to detect new features in codes.

Loopest's reports have a block for each loop in the source code. Figure 5.9 shows an extract of the report generated for 458.sjeng, describing the *FOR* loop that starts

```
174: Line number: 174
174: Number of lines: 29
174: Inclusive Time Percent: 0.3
174: Exclusive Time Percent: 0.1
174: It doesn't contain pointer arithmetic.
174: It contains a BREAK statement.
174: Number of Loop Control Variables: 1
174: It contains all its parts (initial, condition and step)
174: Loop Control Variable: (register int) i.
174: It hasn't ancestor loops.
174: Variables read: (static const int) bishop_o[4], ....
174: Number of variables read: 7
174: Variables written: (register int) ndir, ....
174: Number of variables written: 3
174: Variables always written before read and not appear outside the loop:
    (register int) ndir, ...
174: Number of variables read and written: 3
174: Variables read and written: (register int) ndir, ....
174: Variables only read: (static const int) bishop_o[4], ....
174: Private Variables: (register int) ndir, (register int) a_sq, ...
174: Shared Variables: (static const int) bishop_o[4], ...
174: It is a ''Well-formed'' FOR Loop.
174: It is ''Well-formed'' FOR Loop with only private or shared variables.
```

Figure 5.9: Extract of report generated for 458.sjeng. Somes lines are cut to better presentation.

in line 174. Information shown contains an estimated number of lines of the *FOR* loop, extracted from the IR generated by Cetus. Moreover, the report shows the inclusive and exclusive times of the loop in percentage respect to the total execution time, thanks to the contribution of Profilazer. Other useful information reported by Loopest is pointer arithmetic usage, statements that break the execution flow, read and written variables, variable classification into private, shared and speculative categories, and whether each loop fits into the "well-formed" category defined above. This part of the report is interesting to individually characterize *FOR* loops. However, it is even more interesting a report about the source code as a whole, summarizing the results. Figure 5.10 shows the final lines of the report generated for 458.sjeng benchmark of SPEC CPU2006.

These final lines show a good characterization of the source code, from the point of view of its *FOR* loops. First, the report shows the percentage of "well-formed" *FOR* loops and the percentage of these loops that only hold private and read-only shared variables, and therefore may be parallelized at compile time. Subtracting the second value from the first, we can obtain the percentage of loops that are potentially speculatively parallelizable. Last features extracted by Loopest are the percentage of loops that use pointer arithmetic, memory allocation functions, I/O functions, or are

```
_____________________________
Number of FOR loops: 216
% of ''Well-formed'' FOR Loops: 91.66667%
% of total time of the ''Well-formed'' FOR Loops: 19.599998%
% of ''Well-formed'' FOR Loops with only Private and Shared variables:
    51.38889%
% of total time of the ''Well-formed'' FOR Loops with only Private and
    Shared variables: 17.7%
% of FOR Loops with pointer arithmetic: 10.185185%
% of total time of the FOR Loops with pointer arithmetic: 9.6%
% of FOR Loops with memory allocation functions: 0.0%
% of total time of the FOR Loops with memory allocation functions: 0.0%
% of FOR Loops with I/O functions: 4.166667%
% of total time of the FOR Loops with I/O functions: 0.0%
% of FOR Loops affected by static variables: 18.981482%
% of total time of the FOR Loops affected by static variables: 8.6%
_____________________________
```

Figure 5.10: Final lines of a report generated by Loopest for 458.sjeng.

affected by static variables. As a practical application, and a demonstration of the capabilities of BFCA, in Section 5.5 this framework will be used to process some of the SPEC CPU2006 benchmarks.

Searching for static variables

Recognition of loops affected by static variables, namely, variables whose lifetime extends across the entire run of the program, is interesting for parallelization purposes, because the value of a write on a static variable in the context of a loop has to be preserved after the loop. However, a dependency on a static variable is not always obvious because the variable could be inside a nested loop, or nested function call. Therefore, it is very useful to know the execution path that reaches the static variable that affects a particular loop. Figure 5.11 shows the execution path that reaches the static variable that affects the loop in line 7633 of 482.sphinx3. Loopest only shows the first appearance of a static variable affecting the loop. Doing manually this task is really time-consuming, because nested loops, and above all nested function calls, complicates maintaining the trace. Therefore, providing an automatic method of obtaining these traces is very useful.

Loopest detects static variables that affect a particular loop in the following way:

1. First, Loopest searches for static variables within the body of the current loop, including nested loops. If Loopest finds a static variable affecting the loop, stops the search and prints the trace of the execution path that reaches that static variable.

```
7633: Static variable in line: 1210 : (static mylist_t * ) head

Path:
 for loop  (file: sphinx3.c (7633), incl. time: 0.0%, excl. time: 0.0%)
  lextree_build  (file: sphinx3.c (7634))
   for loop  (file: sphinx3.c (8350), incl. time: 0.0%, excl. time: 0.0%)
    for loop  (file: sphinx3.c (8364), incl. time: 0.0%, excl. time: 0.0%)
     for loop  (file: sphinx3.c (8371), incl. time: 0.0%, excl. time: 0.0%)
      for loop (file: sphinx3.c (8398), incl. time: 0.0%, excl. time: 0.0%)
       lextree_node_alloc  (file: sphinx3.c (8409))
        __mymalloc__  (file: sphinx3.c (8303))
         (static mylist_t * ) head  (file: sphinx3.c (1210))

7633: Loop affected by static variables.
```

Figure 5.11: Trace of the static variable affecting loop in line 7633 of 482.sphinx3.

2. If there is not a static variable affecting the loop within its body, Loopest searches for static variables within functions called from the current loop, nested loops inside these functions, or functions called inside these functions. Note that the level of nesting could be very high, and this is precisely why Loopest is so valuable. It automatically performs a task that would be very time-consuming to do manually.

3. Once a static variable is found in any nested level, Loopest prints the trace from the first *FOR* loop before the variable occurrence.

To search static variables inside functions called within loops, Loopest keeps a list with functions that is gradually extended with new functions as it finds them in nested function calls. Once a function is called, Loopest assumes that possible future paths to this function call will be the same. Thus, once a function is analyzed, it is annotated to not be analyzed more times.

To keep the trace, Loopest maintains different lists with information of which loops (identified by line number) are contained within which functions, which functions are called by other functions, as well as the line number of these function calls. With all this information, Loopest is able to recover the trace and rebuild the path from the first *FOR* loop until the static variable found.

5.4.7 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 [95] capabilities, and uses template rules to translate the XML document back to C. To apply the XSLT transformation rules, we have chosen the Saxon tool [158], due to its open-source nature and because it implements XPath and XSLT 2.0.

The structure of the XSLT program developed consists in a set of template rules, one for each element of the DOM tree that should be transform back into C language elements. These template rules generate the C code that corresponds to the identified DOM element, triggering other rules when necessary. For example, in the case of a *BinaryExpression* element, the corresponding template rule generates the correct binary operator, according to the value of the corresponding attribute, and decides the application of the appropriate template rules to the left and right side expressions of the binary operator. In other cases, Sirius defines modes for the template rules in order to apply them depending on the context. For example, a node *Initializer* can be either a child of a simple variable declarator node or an "struct" declarator. This differentiation is necessary to correctly rewrite the code.

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. Besides formatting information, the only difference between the original source file and the generated one involves constants defined in the original code with `#define` macros, that now appear with their defined value.

With this transformation Sirius finishes the compilation framework process. Purely technological details of Sirius, Loopest and XMLCetus can be found in our M.Sc. thesis [5].

5.4.8 Validation

Software developed and presented in this document needs to be validated. As it has been already described, the framework are divided in four subsystems or modules. The validation process for each one described below.

XMLCetus and Sirius

Validation of XMLCetus has been achieved using Sirius, which rebuilds a C code throughout its XML representation. Therefore, if the C code rebuilt by Sirius is correct, and functionality equivalent to the original C code, this constitutes a proof-of-work that validates the correct operation of both XMLCetus and Sirius.

There have been developed more than 75 regression tests in order to validate these two subsystems. Since the methodology followed in the development of XMLCetus and Sirius is a *evolutionary prototyping*, both subsystems have been developed by using an incremental construction of prototypes, each one having more functionality and covering more test cases than the previous one. The aim of these tests is to cover every possible situation in a code. This methodology was chosen because it allows a rapid development and to perform early tests to check the correction of the prototypes. The regression tests developed can be found in the digital support attached to this document.

Profilazer

The different auxiliary tools that have been used to assist and validate Profilazer operation are described below. With the exception of the first tool, all of them are included in the digital support attached to this document.

- First of these tools is **loopprofileviewer**, provided by the Intel compiler. This tool allows visualizing the profiling XML files generated by the compiler in a nice human read format. It is also allows sorting each function or loop profiled by name, line, absolute execution time (inclusive or exclusive), and percentage of execution time. Using this tool it was easy to detect repetitions in annotated *FOR* loops, with different profiling information for each one, as it was described previously. This tool was also helpful to compare Intel profiling information with other provided by Sun tools, which were used in previous developments.

- **LoopFinger**: It is used to check whether the line numbers of the *FOR* loops annotated by XMLCetus are the same than their line numbers in the original source code. This tool was developed to solve a problem with the original Cetus software, already described in page 112, that does not correctly maintain the source code line numbers. Therefore, we developed this tool to track this problem.

```
$ LoopFinger <XML_file.xml> <sourceCode>.c
```

- **ProfilazerCheck**: The XML file generated by the Intel profiler often contains several repetitions for the same loop, with different times annotated each time. As it was described before, the repetition with the longest time annotated is the correct. This tool is used to check whether each *FOR* loop is correctly annotated with the longest time.

```
$ ProfilazerCheck <XML.profiled>.xml <profiling-information>.xml
```

Loopest

Loopest has been validated using more than 50 regression tests. Like with XMLCetus, development methodology followed with Loopest is a *evolutionary prototyping*, and thus, the validation process is quite similar to the one followed in the development of XMLCetus and Sirius. These regression tests can be found in the digital support attached to this document.

5.5 Evaluation of the solution

To demonstrate the capabilities of our analysis framework, we will characterize a set of C benchmarks in order to measure the potential gain that could be obtained with the parallelization of some of their loops. As long as we will 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 [82]. As it was described in Chap. 2, these benchmarks are based in real-world applications, and are a good touchstone to test BFCA capabilities.

The structure of this section is the following. First, the evaluation methodology followed is described, as well as the data set used. SPEC CPU2006 C benchmarks will be used to obtain a characterization of loops from these applications. Second, we will discuss the opportunities for parallelization of each SPEC CPU2006 C benchmark considered, classifying *FOR* loops in different categories with respect to their suitability for parallel execution and taking into account the existence of potential dependency violations. Third, 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. With regard to the latter feature, in order to demonstrate the potential of the framework, we reproduce here the examples of traces of *FOR* loops affected by the presence of static variables in a given nested level of loop or function call. Finally, we will quantify costs associated to the use of BFCA in terms of execution times and sizes of the generated XML files.

5.5.1 Evaluation methodology

In Sect. 5.2.1 it was described the importance of data dependencies and other situations in order to parallelize source code. Therefore, building a framework that extracts information about source codes and characterizes *FOR* loops, which are the main target to parallelize, is a valuable tool. In order to evaluate this framework in real situations, the use of benchmarks from suites such as SPEC CPU is a good alternative to test the potential and capabilities of the framework. This section describes which features are extracted from the C benchmarks that have relevance in parallelization tasks, how these features are measured, and which datasets are used in the experiments.

Design of the experiment

Experiments have been designed to satisfy the goals presented in Sect. 1.2. One of the goals is the use of profiling information to determine the relative importance of each *FOR* loop in a source code, in terms of execution time. This goal is related to the characterization of loops features, because using profiling information we can determine the importance of each feature. In order to show framework capabilities and obtain this characterization, the C benchmarks of SPEC CPU2006 will be used. The experiment process has been carried out as follows:

1. XMLCetus is executed with each source code to obtain an XML-based representation for them.

2. All C benchmarks are compiled using the Intel compiler with the option "*-profile-loops=all*", which instruments the source code to obtain profiling information when the application is executed.

3. Using a script that automatizes the process, each benchmark is executed with the three workload sets defined by SPEC CPU2006, ranked increasing workload: *Test*, *Train*, and *Reference*. In some cases, a workload requires the execution of the same benchmark several times. Regardless of this, and with the aim of avoiding the effects of an irregular execution, each input set is executed three times, and the average percentage is obtained.

4. Each execution generates an XML file with the profiling information. This profiling XML file is renamed with the name of the application and the name of the workload used. In the cases that the benchmark is executed more than once, an XML tree is generated for each execution. Besides, for each set of three executions which are used to obtain the average execution times, we create a new XML tree that saves this average time for each tuple benchmark-workload set.

5. Once we have the profiling XML trees, Profilazer is executed to add profiling information into the XML files created by XMLCetus. When there are various profiling XML files for a same workload set, different XML files are generated for each profiling file.

6. Finally, Loopest is executed with the profiled XML files, and statistics are extracted. When there are different profiled XML files for a single workload and application average values are calculated for these statistics.

The second part of the experiment aims to show the potential of the framework performing automatically a task that it would be endless manually: To obtain the trace from the *FOR* loops until we reach a static variable that affects the loops, no matter the level of nesting of loops or function calls. The process of this phase is very similar to the previous one. Unlike the previous phase, we do not seek to obtain an exhaustive characterization of these traces. Therefore, this study is only focused on those benchmarks in which the dependency on static variables is significant in terms of execution time. These times are obtained in the previous phase, as we saw above. Therefore, in this point we only focus on the three benchmarks that fulfill this requirement: 429.mcf, 458.sjeng, and 482.sphinx3. Only the most relevant results will be shown. Obtaining these traces is a very similar process to the one followed in the previous phase:

1. As a result of the previous phase, the benchmarks selected for this experiment are already compiled and executed, as well as their corresponding XML trees are also obtained.

2. A profiled XML file corresponding to the reference workload is picked for each benchmark.

3. Loopest is executed using each profiled XML file, obtaining and saving the resulting reports.

4. By searching in each report, only relevant or significant situations are chosen to show them as examples of the potential of the developed framework.

Finally, it is obtained an empirical estimation of the costs of using the framework, in terms of execution times and sizes of the XML files generated. To obtain execution times, the command *date* is used before and after executing each subsystem.

Evaluation measures

The results obtained through the experiments are presented using the following measures:

- **Coverage time**: Percentage of execution time that *FOR* loops that satisfy a particular condition with regard to the total running time. Coverage times shown in the tables are referred to the three working sets defined in the SPEC CPU2006 benchmark suite.

- **Lines of source code**: This measure acts as an estimation of the complexity and dimension of source codes, to put both BFCA execution times and size of XML files into perspective.

- **BFCA execution times**: Time will be measured in seconds, with a precision of two decimals. This measure provides an empirical estimation of the time complexity of the developed framework.

- **Source code and XML files sizes**: Size will be measured in kilobytes, with a precision of two decimals. This measure provides an empirical estimation of the spatial complexity.

- **Rates between times and between sizes**: In order to set reference points to compare times associated to the framework it is necessary to obtain the execution times of the original software of Cetus and the time consumed by XML-Cetus. The ratio allows comparing how different the first values are in comparison with the second values. We will also calculate the ratio between the size of the original C source code and the size of the corresponding XML file, to measure the overhead incurred.

Dataset

As we stated before, we will use C benchmarks extracted from SPEC CPU2006 to obtain a characterization of the *FOR* loops from these applications. SPEC CPU2000 C benchmarks could be characterized too, but from the point of view of academic interests, SPEC CPU2006 is the last benchmark suite, also the most complex and with a larger dimension, and thus, it is more interesting to characterize these benchmarks. However, SPEC CPU2000 benchmarks are very useful to experimentally estimate time and spatial complexity of the framework operation. They will be used, together with SPEC CPU2006, to measure times execution of original Cetus in comparison to XMLCetus, execution times of Profilazer and Loopest operation, and size of the XML files generated by XMLCetus in comparison to original source files.

In order to obtain the more accurate characterization results, we use the three input sets that SPEC 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 bigger

Application	Lines of code	Description
401.bzip2	7 292	Julian Seward's bzip2 version 1.0.3, modified to do most work in memory, rather than doing I/O
429.mcf	2 044	Vehicle scheduling. Uses a network simplex algorithm to schedule public transport
433.milc	12 837	A gauge field generating program for lattice gauge theory programs with dynamical quarks
456.hmmer	33 210	Protein sequence analysis using profile hidden Markov models (profile HMMs)
458.sjeng	13 291	A highly-ranked chess program that also plays several chess variants
462.libquantum	3 454	Simulates a quantum computer, running Shor's polynomial-time factorization algorithm
464.h264ref	46 142	A reference implementation of H.264/AVC, encodes a videostream using 2 parameter sets
470.lbm	875	Implements the "Lattice-Boltzmann Method" to simulate incompressible fluids in 3D
482.sphinx3	18 280	A widely-known speech recognition system from Carnegie Mellon University

Table 5.2: Description of the SPEC CPU2006 benchmarks used in the experiments. Lines of code are calculated using SLOCCount [168].

workload set used to obtain execution times, and hence, the final performance results. The conditions of use of these benchmarks suite include the mandatory use of these inputs 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 [81, 82].

With regard to SPEC CPU2006 benchmark suite, and taking into account that the prototype is designed to exclusively handle C programs, we have only selected a subset of C benchmarks of this suite. Table 5.2 includes a description and shows the size of the considered benchmarks. This list includes all C applications except 400.perlbench, 403.gcc and 445.gobmk. The reason to exclude them for this study is that the version of the Cetus framework used is not able to process them, probably because these benchmarks are fragmented into many C files (53 for 400.perlbench, 155 for 403.gcc, and 89 for 445.gobmk) with many conditional compilation flags, and a great number of lines (around 116 000 for 400.perlbench, 484 000 for 403.gcc, and 157 000 for 445.gobmk).

As with CPU2006, we only use the C programs of CPU2000 [81]. Table 5.3 sum-

Application	Lines of code	Description
164.gzip	7 600	Data compression program written by Jean-Loup Gailly for the GNU project
175.vpr	13 600	Placement and routing program that implements a technology-mapped circuit in a FPGA chip
177.mesa	81 800	Free OpenGL 3-D graphics library
179.art	1 200	Neural network used to recognize objects in a thermal image
181.mcf	1 900	Vehicle scheduling. Uses a network simplex algorithm to schedule public transport
183.equake	1 200	Simulation of seismic wave propagation in large basins
186.crafty	20 700	High-performance Computer Chess program that is designed around a 64 bit word
188.ammp	12 900	Modeling large systems of molecules usually associated with Biology
197.parser	10 300	Syntactic parser of English
254.gap	62 500	It implements a language and library designed mostly for computing in groups
256.bzip2	3 900	Julian Seward's bzip2 version 0.1, performs no file I/O other than reading the input
300.twolf	19 200	Place and route simulator

Table 5.3: Description of the SPEC CPU2000 benchmarks used in the experiments. Lines of code extracted from [81].

marizes descriptions of the benchmarks used in the experiments. This table contains all the C benchmarks of CPU2000, excluding 176.gcc, 253.perlbmk and 255.vortex, because Cetus fails in their processing. Two of these benchmarks are the same program (different versions) than used in CPU2006, 403.gcc and 400.perlbench, and thus, Cetus experiences the same problem described above. Regarding the experimental evaluation, as we stated before, we have run each input set three times and obtained the average percentage.

In order to obtain the results presented in this chapter, it was necessary to create different scripts that automatize the process. They are essentially *bash* scripts that automatically execute the benchmarks, obtain the profiling XML files, execute Profilazer to obtain profiled XML documents, and execute Loopest to get the final results presented in this chapter. Since the description of these scripts is not relevant, and targets (SPEC CPU benchmarks) of these scripts are protected by licences, they are not described here, neither included in the digital support attached to this document.

Regarding to the execution of benchmarks, it is necessary to be sure that they are

correctly run, and generate the correct output files. Most of the developed scripts used the GNU tool *diff* to check whether results are equal to the expected results. However, this tool has a limitation: It only compares textual outputs. SPEC determines a tolerance to be admitted in numerical results for some benchmarks. *diff* is not able to take into account these numerical tolerances (in real numbers). Therefore, for those benchmarks, it was necessary to use the SPEC version of *diff*, called *specdiff*, which is a Perl script that admits these tolerance. Like SPEC CPU benchmarks, *specdiff* are attached to a license and it cannot freely distributed. For this reason, this script is not included in the digital support either.

5.5.2 Loop characterization of SPEC CPU2006 C benchmarks

As we have stated before, we have characterized a set of C benchmarks from SPEC CPU2006 in order to both demonstrate the capabilities of our framework, and 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 are taken into account.

First, this section shows opportunities for parallelization in each of the SPEC CPU2006 C benchmarks analyzed, classifying their *FOR* loops in three categories that will be seen later. Second, a more in-depth analysis is performed to extract the use of different types of variables, structures and functions inside *FOR* loops, whose usage conditions the opportunities for parallelization. Finally, a characterization of the usage of the static variables in those benchmarks will be shown.

Loop characterization with respect to potential dependency 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 dependency 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 dependency 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 dependency 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.

Application	Well-formed *FOR* Loops			
	% of loops	Coverage Test	Coverage Train	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	**8.42**	**43.87**	**45.53**	**48.22**

Table 5.4: Well-formed *FOR* loops.

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, in order to obtain loop coverages in different circumstances. We 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 5.4 and 5.5 summarize the results of the study of our SPEC CPU2006 C benchmarks. For each benchmark considered, the tables show the following information:

- Summary of "well-formed" *FOR* loops (Fig. 5.12a), 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.

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**

Table 5.5: Opportunities for parallelization techniques.

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

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

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

Figure 5.12: (a) Example of a "well-formed" *FOR* loop. (b) "Well-formed" loop that only holds private (`i`, and `v[]`) and read-only (`a[]`) data structures. (c) "Well-formed" loop not safely parallelizable at compile time, because the statement can lead to a dependency violation.

Table 5.4 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. 5.12b). As can be seen in the last four columns of Table 5.5, 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 more than 20% of the total execution time.

- Summary of loops that cannot be safely parallelizable at compile time (Fig. 5.12c). 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 dependency 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 [37].

  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.

Loop characterization with respect to parallelization hurdles

As we stated above, it is not enough for a loop to be free of potential dependency 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 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 5.6 and 5.7 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. 5.13a). 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 (around 80%) reflects the importance of supporting pointer arithmetic in compile-time or runtime parallelization schemes, a problem that has been recently solved in the general case [62]. 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 explain 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 [6].

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 5.6: Relevance of challenges for parallelization techniques: Pointers and memory management.

- Summary of loops that perform calls to memory management functions, such as *malloc()* or *free()* (Fig. 5.13b). 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 speed up these benchmarks.

- Summary of loops that contain I/O function calls (Fig. 5.13c). As in the pre-

```
int * p, q;
for (i=0;i<100;i++)
{
    ...
  p = &q;
    ...
}
```
(a)

```
int * p;
for (i=0;i<100;i++)
{
     ...
 p=(int*)malloc(..);
 free (p);
     ...
}
```
(b)

```
for (i=0;i<100;i++)
{
     ...
 printf(...);
     ...
}
```
(c)

```
static s;
for (i=0;i<100;i++)
{
     ...
 s = function(i);
     ...
}
```
(d)

Figure 5.13: (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.

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**

Table 5.7: Relevance of challenges for parallelization techniques: I/O activity and use of static variables.

vious 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 without operating system support. 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 neither be a priority for automatic parallelization techniques.

- Percentage of loops that are affected by static variables (Fig. 5.13d). 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.

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. Recall that static variables keep their value across different calls of the function that define them. Searching manually these static variables is a daunting task, and writing down the trace until reaching a definition of a static variable requires a substantial effort.

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.

In this section we will see how we can automatically solve this problem in seconds, even in large programs like the benchmarks from SPEC CPU2006. Moreover, the framework does not only provide just the trace, but also execution times, and that is a very useful information in order to filter loops by importance, in terms of their execution times. Since only the loops of three benchmarks from SPEC CPU2006 are significantly affected by static variables (see Table 5.13), we will focus on them. These benchmarks are 429.mcf, with a 34.2% of coverage in the reference workload set, 458.sjeng, with an 8.6% of coverage), and 482.sphinx3 (with a 4.5% of coverage). Others benchmarks have a coverage in the reference workload set lower than 1%, and therefore, they will not be considered in detail.

The 429.mcf benchmark is one of the shortest benchmarks in SPEC CPU2006, and it only has 33 *FOR* loops. Thus, it is not unusual that only two loops are affected by static variables, as we can see in Fig. 5.14. These two loops are affected each one by static variables which are directly in their bodies. Information presented by Loopest includes not only the dependency trace of the static variable, but also the inclusive and exclusive time of each loop in the trace. This information is important to weight the relevance (in terms of execution time) of the dependency on the static variable. Loop in line 1937 has an inclusive and exclusive time of 34.2%, which is the same percentage that we found in Table 5.13.

The 458.sjeng benchmark has 41 loops affected by static variables, but only three of them has a time, inclusive or exclusive larger than 0.0%. These loops are shown in Fig. 5.15. The sum of the exclusive times of each loop is 8.6%, the same time as the coverage for the reference workload set shown in Table 5.13.

Traces shown in Fig. 5.15 manifest the value and usefulness of the developed framework in a much better way than the previous benchmark . In each trace, the

```
for loop  (file: 429mcf.c (1909), incl. time: 0.0%, excl. time: 0.0%)
 (static BASKET * ) perm[((50+300)+1)]  (file: 429mcf.c (1910))

for loop  (file: 429mcf.c (1937), incl. time: 34.2%, excl. time: 34.2%)
 (static long) basket_size  (file: 429mcf.c (1942))
```

Figure 5.14: Excerpt of the report returned by Loopest for 429.mcf, notifying the use of the static variables `perm` and `basket_size`, and describing both the functions and loops affected, together with their coverage.

```
for loop  (file: sjeng.c (8174), incl. time: 10.5%, excl. time: 7.8%)
 see  (file: sjeng.c (8193))
  setup_attackers  (file: sjeng.c (10022))
   (static const int) rook_o[4]  (file: sjeng.c (9855))

for loop  (file: sjeng.c (8725), incl. time: 1.4%, excl. time: 0.8%)
 check_legal  (file: sjeng.c (8727))
  is_attacked  (file: sjeng.c, (2443))
   (static const int) rook_o[4]  (file: sjeng.c (164))

for loop  (file: sjeng.c (9537), incl. time: 99.8%, excl. time: 0.0%)
 search_root  (file: sjeng.c (9550))
  gen  (file: sjeng.c (9072))
   (static move_s * ) genfor  (file: sjeng.c (2844))
```

Figure 5.15: Excerpt of the report returned by Loopest for 458.sjeng, notifying the use of the static variables `rook_o` and `genfor`, and describing both the functions and loops affected, together with their coverage.

dependency with the static variable is situated two nested levels below the *FOR* loop. Namely, within the loop, there is a function call, inside this function there is another function call, and finally within the last function we can find the static variable defined or used. Obtaining manually all these traces is a hard and time-consuming task. However, by using the framework this task can be automatically achieved without spending time. The report that Loopest generates, using the XML file created by XMLCetus, and the annotations made by Profilazer, provides information about the trace of the dependency of a loop to a static variable, and moreover, the execution times of each loop in every traces, as we will see with the following benchmark.

The last benchmark analyzed, named 482.sphinx3, is more complex than the previous benchmark studied. It has 67 *FOR* loops affected by static variables. Since simpler cases have been seen in the 429.mcf and 458.sjeng benchmarks, in this point, we will only focus on the more complex case. Figure 5.16 shows an excerpt of the report generated by BFCA when analyzing 482.sphinx, which has seven nested loops

```
for loop (file sphinx3.c (12504); incl. time 2.0%; excl. time 0.2%)
 for loop (file sphinx3.c (12505); incl. time 0.1%; excl. time 0.1%)
  fe_frame_to_fea (file sphinx3.c (12510))
   fe_spec_magnitude (file sphinx3.c (12895))
    for loop (file sphinx3.c (12928); incl. time 0.0%; excl. time 0.0%)
     for loop (file sphinx3.c (12932); incl. time 0.0%; excl. time 0.0%)
      for loop (file sphinx3.c (12939); incl. time 0.0%; excl. time 0.0%)
       for loop (file sphinx3.c (12943); incl. time 0.0%; excl. time 0.0%)
        fe_fft (file sphinx3.c (12950))
         for loop (file sphinx3.c (13037); incl. time 0.0%; excl.time 0.0%)
          (static int) k (file sphinx3.c (13037))
```

Figure 5.16: 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.

and three nested function calls. These numbers provide an idea of the usefulness of the developed framework and make clear its value.

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 characterizations are merely a demonstration of the kind of 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.

5.5.3 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 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 [168]).

Application	Lines of code	Time in seconds		Rate of slowdown	Size on Kilobytes		Rate
		Cetus	XMLCetus		Source	XML	
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**

Table 5.8: Generation times of XML documents for SPEC CPU2006 benchmarks, rate of slowdown respect to Cetus' original execution, and file sizes.

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.

5.5.4 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.

XMLCetus performance

Tables 5.8 and 5.9 resume the execution times of both XMLCetus and the original Cetus framework. The right part of the tables shows the sizes in Kilobytes of both

Application	Lines of code	Time in seconds: Cetus	Time in seconds: XMLCetus	Rate of slowdown	Size on Kilobytes: Source	Size on Kilobytes: XML	Rate
164.gzip	7 600	6.99	7.79	1.11	238	2 600	10.92
175.vpr	13 600	9.84	12.47	1.27	535	5 944	11.11
177.mesa	81 800	28.69	59.55	2.08	1 434	24 168	16.85
179.art	1 200	2.42	2.99	1.24	27	512	18.96
181.mcf	1 900	7.88	10.02	1.27	42	632	15.05
183.equake	1 200	2.44	2.89	1.18	39	756	19.38
186.crafty	20 700	11.55	16.14	1.40	809	8 756	10.82
188.ammp	12 900	11.12	13.93	1.25	337	6 940	20.59
197.parser	10 300	8.65	10.54	1.22	333	4 868	14.62
254.gap	62 500	17.78	23.56	1.33	2 150	29 804	13.86
256.bzip2	3 900	3.74	4.46	1.19	129	1 368	10.60
300.twolf	19 200	16.05	20.46	1.27	500	10 844	21.69
Average	**19 733**	**10.60**	**15.40**	**1.32**	**548**	**8 099**	**15.37**

Table 5.9: Generation times of XML documents for SPEC CPU2000 benchmarks, rate of slowdown respect to Cetus' original execution, and file sizes.

the original code and its corresponding XML representation for each analyzed benchmark. As expected, the larger the source code, the longer 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 and CPU2000 benchmarks, with peak values for each suite of 1.50 (464.h264ref), and 2.08 (177.mesa). These differences, which do not alter the user experience, are significantly reduced respect to results described in Power and Malloy's work [137], where the rate of slowdown reaches a 25.2 on average for different benchmarks.

Regarding file sizes, as Power and Malloy [137], Maruyama and Yamamoto [118], and Maletic *et al.* [114] 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, and 15.37 times larger for SPEC CPU2000 benchmarks. XML file sizes generated by BFCA are bigger than files generated by McArthur's *et al.* [119], 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 gen-

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**

Table 5.10: Execution times of Profilazer and Loopest for SPEC CPU2006 C benchmarks.

erates bigger XML files than srcML's (five times larger on average than the original source code [114]), because BFCA does not stop the creation of XML nodes at the expression level. This approach is good in order 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 [74], with XML files more than a hundred times larger than the original source code.

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 *ForLoop* nodes. It hardly means a few bytes to the total size, and thus, these sizes are not shown in this study.

Tables 5.10 and 5.11 show 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,

Application	Lines of code	FOR Loops	Time in seconds	
			Profilazer	Loopest
164.gzip	7 600	78	16.19	1.79
175.vpr	13 600	346	36.08	6.19
177.mesa	81 800	756	224.38	13.59
179.art	1 200	71	5.38	1.94
181.mcf	1 900	34	3.62	1.76
183.equake	1 200	86	5.09	2.22
186.crafty	20 700	223	207.87	6.54
188.ammp	12 900	253	44.58	5.69
197.parser	10 300	450	118.51	5.26
254.gap	62 500	1 140	1 142.19	52.01
256.bzip2	3 900	103	14.01	2.86
300.twolf	19 200	746	279.91	8.33
Average	**19 733**	**357.17**	**174.82**	**9.02**

Table 5.11: Execution times of Profilazer and Loopest for SPEC CPU2000 C benchmarks.

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.

5.6 Conclusions

This chapter addresses the problem of automatic characterization and coverage of sequential loops. We handle this problem with the development of 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, including the runtime library used in this Ph.D. thesis. As we will see in the following chapter, this information can be processed to automatically define OpenMP clauses, including our `speculative` clause proposed in Chap. 3.

Our framework, which is based on Cetus, takes 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.

This framework can use for 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. We hope that the use of this framework will help runtime parallelization technology to be mature enough for its inclusion in mainstream compilers.

The third use for this framework is the instrumentation of the augmented XML code with automatic parallelization directives, such OpenMP and our proposed `speculative` clause, in order to automatically transform the code into a speculative, parallel version of the code, and letting Sirius translate the modified XML tree back into C language, finishing the process. We will examine this possibility in detail in the following chapter. We consider that this solution is a step towards the automatic use of TLS techniques.

The work and conclusions described in this chapter has been published in the following papers:

- The BonaFide C Analyzer: Automatic Loop-level Characterization and Coverage Measurement. Sergio Aldea, Diego R. Llanos, and Arturo Gonzalez-Escribano. *The Journal of Supercomputing*, 2014. Online, DOI:10.1007/s11227-014-1091-3.

- Towards a compiler framework for thread-level speculation. Sergio Aldea, Diego R. Llanos, Arturo Gonzalez-Escribano. *Proceedings of the 19th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP 2011)*, Ayia Napa, Cyprus, February 9-11, 2011. pages 267–271.

- Extending a source−to−source compiler with XML capabilities. Sergio Aldea, Diego R. Llanos, Arturo Gonzalez−Escribano. *Actas XXI Jornadas de Paralelismo*, Valencia, Spain, September 7−10, 2010.

- XMLCetus y Sirius: análisis y traducción de código C utilizando herramientas XML. Sergio Aldea, Diego R. Llanos, Arturo Gonzales-Escribano. *Technical Report IT-DI-2010-001*, Department of Computer Science, Universidad de Valladolid, 2010.

Chapter 6

Automatic Synthesis of Speculative Code

The preceding chapter proposed a system, called BFCA, able to analyze the source code, and generate helpful reports that programmers may use to parallelize source codes. In this case, once BFCA provided the analysis about how the variables are used in the context of a *FOR* loop, programmers should write the OpenMP constructs necessary to parallelize the source code, including our `speculative` clause. In this chapter, we propose to update BFCA to avoid such an error-prone, often time-consuming task, by automatically instrumenting the source code with the necessary OpenMP directives and clauses.

6.1 Problem description

The system proposed in Chap. 5, called BFCA, has some useful features, such as the handling of an XML-based Intermediate Representation (IR) of the source codes that enables to characterize them, or the generation of reports with information about the loops in a source code. Using this information, programmers may be able to parallelize the code, e.g. using the OpenMP standard, or our proposed `speculative` clause in order to apply speculative parallelism. Nevertheless, programmers are forced to annotate the source code manually.

Although the OpenMP standard is relatively easy to use, some level of expertise is advisable in order to produce efficient parallel code. Programmers may use OpenMP incorrectly, or may classify the variables in a parallel region inaccurately, which makes the process of parallelize a source code an error-prone, time-consuming

task.

To solve this problem, there are many proposals that provide an automatic classification of the variables [143, 163], and as we will see in Sect. 6.2, many others that goes beyond and propose systems that automatically generate the OpenMP constructs that are necessary to parallelize a source code. However, neither of them is focused on Thread-Level Speculation (TLS), being the feedback provided by these systems tied to the OpenMP standard.

In this chapter, we extend functionalities of BFCA (see Chap. 5) to automatically generate the OpenMP directives and clauses needed to parallelize a source code speculatively, including our proposed `speculative` clause. This new version will be called BFCA+. By doing this, programmers not only avoid the manual parallelization of the source code, but also leave to BFCA+ the analysis of the code, and the insertion of the OpenMP constructs.

The rest of this chapter is organized as follows. Section 6.2 lists some of the related approaches that also generates OpenMP constructs. Section 6.3 explains how BFCA is updated to augment the source and generate the OpenMP-based speculatively parallel version of the code. Section 6.4 evaluates the solution proposed in this chapter. Finally, Sect. 6.5 concludes this chapter.

6.2 State of the art

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 [87, 91, 92], 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 [21], a source-to-source framework that uses the polyhedral model to optimize the code and generate OpenMP parallel code automatically.

The polyhedral model[1] [17] is used by several proposals to enhance the code and obtain the information needed to create the OpenMP directives automatically. Graphite [160] is a branch of GCC that applies the polyhedral model to different pur-

[1] In the polyhedral model, the loops are described using a mathematical abstraction the optimizations are mathematical transformations on this abstract description.

poses, 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 [76], a similar proposal but focused on a different compiler: LLVM [106]. Other works that also use the polyhedral model such as Par4All [9, 159] and PYPS [77], are based on PIPS [8], 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.* [108], Cetus [54], or several of the proposals seen above), Gaspard2 [156] 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 [127] 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.

Like most of the approaches seen 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. The following section details how BFCA+ operates to create OpenMP-based parallel versions from sequential source codes.

6.3 Solution proposed: BFCA+

As we have seen above, there are many proposals that generate standard OpenMP constructs to parallelize source codes automatically, but they are not intended to address an automatic TLS-based parallelization. All the feedback returned by these systems are tied to the OpenMP standard. Instead, BFCA aims to characterize source code and the loops within from a speculative point of view. In this Ph.D. thesis, we have addressed the problem of characterizing source codes, we have proposed a new OpenMP `speculative` clause, and we have developed a compile-time system that processes this clause to automatically generate the code necessary to parallelize an application speculatively with our TLS runtime library.

In this point, the next obvious step is to close the circle, avoiding the manual intervention of programmers when instrumenting the source code with the OpenMP constructs. We propose an update to BFCA, called BFCA+, that adds a new feature:

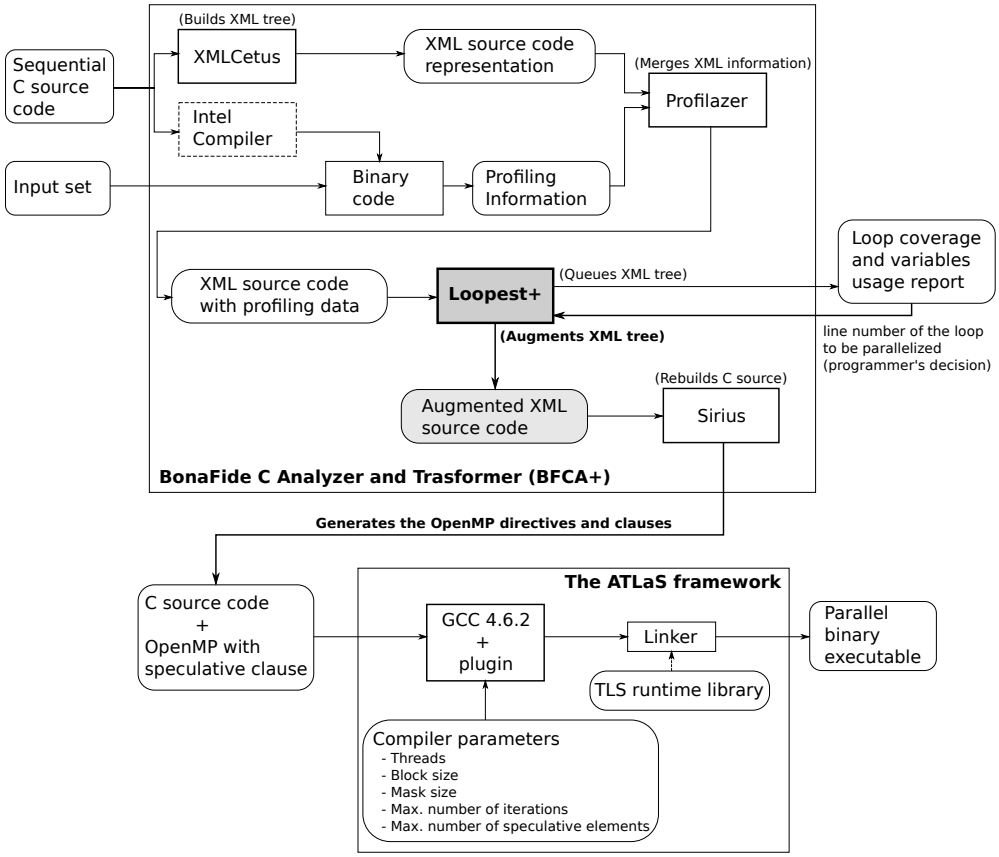

Figure 6.1: Overview of the BFCA+ plus ATLaS architecture, that analyzes the code, generates an OpenMP-based speculatively-parallel version of the code, and finally compiles it to create an executable that runs in parallel speculatively.

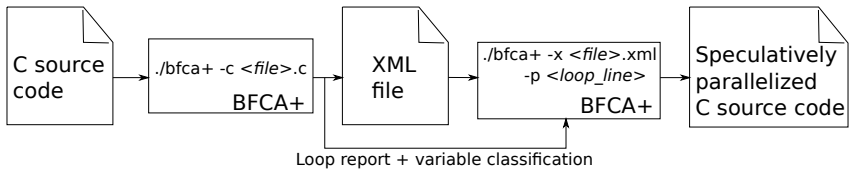

Figure 6.2: Overview of the process that transforms a sequential C code into a speculatively parallel one. BFCA+ generates the OpenMP constructs, including the `speculative` clause, automatically.

```
<ForLoop annotation="OpenMP">
  <Annotation annotation="#pragma omp parallel for default(none) \
    schedule(static) private(i,j) shared(var1,var2) \
    speculative(spec_vars_list) />
```

Figure 6.3: Example of ForLoop augmented with OpenMP constructs.

```
<xsl:template match="ForLoop[@annotation='OpenMP']">
<xsl:copy>
  <xsl:copy-of select="@*"/>
    <Annotation>
      <xsl:attribute name="annotation">
        <xsl:value-of select="$pragma"/>
      </xsl:attribute>
    </Annotation>
  <xsl:apply-templates/>
</xsl:copy>
</xsl:template>
```

Figure 6.4: XSLT code to augment a ForLoop with OpenMP constructs.

To use the information about the variable usage inside *FOR* loops as feedback, generating the proper OpenMP directives and clauses to parallelize a certain loop.

Figure 6.1 details the proposed solution, showing how BFCA+ and ATLaS are combined to ultimately generate an executable that runs in parallel speculatively. BFCA+ analyzes the use of each variable in each loop and classifies them into private, shared, or speculative. This classification is reported to the programmer, who should decide which loop is parallelizable and more profitable in such a case, by using the rest of the information provided by BFCA+ as well. Then, BFCA+ is executed once again, augmenting this time the loop that the programmer has pointed out with the OpenMP parallel directive and the corresponding OpenMP clauses, including our `speculative` clause proposed. These two consecutive BFCA+ runs are shown in Fig. 6.2. The subsequent transformation of the code to actually parallelize the code is performed by ATLaS, our GCC-plugin-based compile-time system described in Chap. 4. 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.

This solution performs all the instrumentation task that consists of annotating a parallel loop with the corresponding OpenMP constructs automatically. Instead of manually annotating the parallel loop, programmers are relieved of such an error-prone, and often tedious task. Figure 6.2 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 Loopest+, which is an enhanced version of the Loopest subsystem presented in Chap. 5. Figure 6.3 shows an example of this operation. Loopest+ 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+ itself creates. This includes the insertion of the `speculative` clause with those variables that may lead to a dependency violation. Figure 6.4 shows the XSLT code that inserts the OpenMP pragma into the XML representation of the source code.

Once Loopest+ 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.

Example of use

As it has been described above, the process of augmenting a parallel *FOR* loop using BFCA+ is split in two steps. Figure 6.5 shows how the programmer should run BFCA+ to parallelize the loop in the line 66 of the *Tough* benchmark shown in Fig. 4.14. First, the programmer should run BFCA+ with the source code, to obtain a report with the characterization of each loop. With this information, the programmer can choose which loop BFCA+ should parallelize. Then, the programmer runs BFCA+ again with the XML file that represents the source code (generated by BFCA+ in the first execution) and the line number of the loop to be parallelized. After these two runs, BFCA+ generates a source code in which the parallel loop is annotated with the OpenMP constructs, including the OpenMP standard clauses and our `speculative` clause, which are created using the variable classification made by BFCA+.

Once the source code is properly annotated, it can be compiled using ATLaS, creating the executable that is able to run in parallel speculatively.

6.3.1 Validation

In order to validate the solution proposed in this chapter, we have used the 75 regression tests that have been used with Loopest and Sirius.

```
$ ls
tough.c
$ bfca+ -c tough.c
...
tough.c:66:  Line number:  66
tough.c:66:  Number of lines:  7
tough.c:66:  Inclusive Time Percent:  34.2
tough.c:66:  Exclusive Time Percent:  34.2
tough.c:66:  It doesn't contain pointer arithmetic.
tough.c:66:  Number of Loop Control Variables:  1
tough.c:66:  Loop Control Variable:  (int) P.
tough.c:66:  Variables read and written:  (int) Q, (int) aux,
tough.c:66:                               (int) array[100].
tough.c:66:  Variables only read:  (int) P.
tough.c:66:  Private Variables:  (int) Q, (int) aux, (int) P.
tough.c:66:  Speculative Variables:  (int) array[100].
tough.c:66:  It is a well-formed FOR Loop.
...

$ bfca+ -x tough.profiled.xml -p 66
$ ls
tough.c   tough.profiled.spec.c
tough.profiled.xml   tough.profiled.spec.xml

$ ./atlas --threads 4 --block 100 --maxpointer 100 --maxiter 1000000
    --mask 127 -e tough.exe -c tough.profiled.spec.c
...
$ ls
tough.c   tough.profiled.spec.c
tough.exe
tough.profiled.xml   tough.profiled.spec.xml
```

Figure 6.5: Example of BFCA+'s run to parallelize the loop in line 66 of the *Tough* benchmark. It is also shown how ATLaS compiles the source code and generates the executable.

6.4 Evaluation of the solution

In Chap. 4 we showed how ATLaS is able to generate all the extra code needed to handle a speculatively parallel execution of a certain application. In order to perform this, ATLaS requires the use of OpenMP, and specifically, our `speculative` clause, which points out those variables that may lead to a dependency violation. Without BFCA+, programmers need 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+ solves this problem freeing programmers from this error-prone, tedious task. To demonstrate the capabilities of BFCA+, we will generate the OpenMP constructs (including our proposed `speculative` clause) for the same benchmarks used in Chap. 4.

6.4.1 Evaluation methodology

The evaluation of BFCA+ requires verifying the correct generation of the OpenMP constructs, and more significantly, the accurate classification of the variables in the loop to assure its correct parallel execution.

Design of the experiment

The experiments have been designed in order to verify that the source codes automatically generated by BFCA+ are the same as the codes manually parallelized by a programmer. Moreover, the experiments also include the verification that the generated source codes are correctly processed by ATLaS, thus generating all the code necessary to handle their speculatively parallel execution. These experiments have been run in the same machine that Chap. 4's.

As every automatic process, the automatic generation of OpenMP constructs by BFCA+ may be not error-prone. Consequently, we will show a situation of an inaccurate classification of the variables, and the reasons behind this performance.

The experiment process has been carried out as follows:

1. Each sequential code is processed by BFCA+ to obtain the classification of the variables, and a characterization of every loop in the code. This run allows us to verify that each variable is correctly classified in *private*, *shared*, or *speculative*.

2. Each sequential code is one more time processed by BFCA+, but this time indicating the line number of the loop that we aim to parallelize speculatively. These loops are the same as the loops parallelized during the experiments of Chap. 4.

3. The resulting source codes, which have been augmented with the OpenMP constructs, are verified to ensure that these constructs are the same as the constructs manually inserted in the benchmarks for the experiments of Chap. 4.

4. These source codes are processed by ATLaS to verify that it is able to correctly generate the speculative code.

5. Finally, the resulting binary files are executed to verify that they correctly run in parallel, and generate the expected results.

Evaluation measures

Due to the particular nature of the experiments performed, they do not produce any different result than the obtained in Chap. 4. Instead, in this chapter, we will show

the transformation process from a sequential code into a parallel one for one of the synthetic benchmarks and one of the real-world applications. First, we will show how the OpenMP constructs are generated, and then, we will show how these OpenMP constructs are processed by ATLaS to generate a TLS-based speculative version of the original source code.

Dataset

As it has been said, the applications used to evaluate BFCA+ are the same as the applications used to evaluate ATLaS, whose descriptions are found in Chap. 4.

```
1  #define NITER 180000
2  int array[MAX];
3  int i,j,k;
4  int spec1 = 0, spec2 = 0;
5  int iter1, iter2;
6  ...
7
8
9
10
11
12
13
14
15 for ( i = 0 ; i < NITER ; i++ ) {
16    if (i == iter1) j = spec1;
17    if (i == iter2) j = spec2;
18
19    for (k = 0;
20         k < array[i % MAX] + j;
21         k++) {
22
23      if (k >= 179900)
24        spec1 =
25          (k+array[(i+k)%MAX])%NITER;
26      if (k <= 1200)
27        spec2 = array[ i % MAX];
28     }
29  }
30
31  if (i == NITER - 1) spec1 = spec2;
32 }
```

(a)

```
#define NITER 180000
int array[MAX];
int i,j,k;
int spec1 = 0, spec2 = 0;
int iter1, iter2;
...

specbegin(NITER)

#pragma omp parallel default(none) \
 private(i, k) \
 shared(array, iter1, iter2) \
 speculative(spec1, spec2)

 for ( i = 0 ; i < NITER ; i++ ) {
   if (i == iter1) j = spec1;
   if (i == iter2) j = spec2;

   for (k = 0;
        k < array[i % MAX] + j;
        k++) {

     if (k >= 179900)
       spec1 =
         (k+array[(i+k)%MAX])%NITER;
     if (k <= 1200)
       spec2 = array[ i % MAX];
    }
 }

 if (i == NITER - 1) spec1 = spec2;
}
```

(b)

Figure 6.6: OpenMP constructs generated by BFCA+ for the *Fast* synthetic benchmark (a), and the sequential version of the code (b). Performance results of this code are shown in Fig. 4.16 (page 85), and Table 4.1 (page 87).

6.4.2 Generation of OpenMP constructs

BFCA+ has been tested with all the synthetic benchmarks and real-world applications used and described in Chap. 4. In this section, we show how BFCA+ correctly processes the *Fast* the synthetic benchmark, and the 2D-MEC application.

Figure 6.6(b) shows the generation of OpenMP constructs for the *Fast* synthetic benchmark (Fig. 6.6(a)). BFCA+ is able to correctly classify the variables accessed in the loop of line 15. Variables `i` and `k` are *private*, since they are always written in the context of an iteration. Variables `array`, `iter1`, and `iter2` are only read, and thus they are classified as *shared*. Finally, variables `spec1` and `spec2` are classified as *speculative*, because their accesses may lead to a dependency violation. Once BFCA+ has augmented the source code, ATLaS is able to process it in order to generate all the extra code needed to handle its speculatively parallel execution. Figure 6.7 depicts the intermediate representation generated by ATLaS after processing the source code augmented by BFCA+. ATLaS adds the clauses needed for the variables used by the TLS runtime system, augments the code to handle the correct assignation of iterations to each thread, and replaces each access to the speculative variables with their corresponding speculative accesses: `specstore_pointer()` and `specload_pointer()`. Finally, the execution of the resulting code leads to the expected output.

As an example of the correct operation of BFCA+ with a real-world application, Fig. 6.8 depicts the OpenMP constructs generated for the 2D-MEC application. The variables that are only read in the loop are classified as *shared*, whereas the variables that are always written before being read in the context of an iteration are classified as *private*. Although the variables `solrx`, `solry`, and `puntdef` are only written in the loop, they are first read after the loop, and thus they are classified as *speculative*. Following the OpenMP standard, these variables should be classified as *lastprivate*, but in our TLS runtime system they are treated as *speculative*, ensuring that after the loop they have their expected value.

BFCA+ also classifies the variables `centrox`, `centroy`, and `cuad_radio` as *speculative*, because they are always read before being written, in the context of an iteration. Therefore, if their accesses are not protected, a certain iteration may read a value of one of these variables that may have been changed by another iteration, leading to a dependency violation.

As every automatic process, the automatic generation of OpenMP constructs by BFCA+ may be not error-prone. Consequently, BFCA+ may make a mistake classifying the variables accessed in a certain loop. Figure 6.9 shows an example of this incorrect performance. The variable `stack` is classified as `private`, because it is always written before being read in the context of an iteration. However, the index that determines which position is accessed each time is changed in line 87, and thus the writing in line 88 may affect to another iteration. Therefore, it should be classified as

`speculative`. The reason behind this erroneous classification is that BFCA+ is not currently able to detect if the index of an array access is changed during the execution of the iteration.

6.5 Conclusions

This chapter addresses the problem of the automatic synthesis and generation of OpenMP constructs. We handle this problem updating BFCA to leverage its output, i.e., a characterization of the source code, and a report about each *FOR* loop, as well as a classification of each variable in each loop according to their accesses. This updated version of BFCA, called BFCA+, 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 our TLS runtime-library. 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.

This Ph.D. thesis is focused on TLS, and the preceding chapters have proposed a new OpenMP clause for variables that may lead to dependency violation, and a compile-time system, ATLaS, that use this clause to automatically generate the code that it is necessary to parallelize a certain loop speculatively. The solution proposed in this chapter automatically synthesizes and generates the OpenMP constructs, including our `speculative` clause, to parallelize a source code using our TLS runtime library.

A future improvement for BFCA+ is the implementation of some heuristics to select the loop to be parallelized automatically. By doing this, programmers could avoid deciding which loop is more profitable to be parallelized, relying on BFCA+ this decision, and bypassing any manual intervention.

We are looking forward to publishing the work and conclusions presented in this chapter.

```
#pragma omp parallel default(none) private(i, k, ini, current, tid, retflag, value) \
 shared(array, iter1, iter2, spec1, spec2, 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;
 i = varblock[0][current]+ini;

 if (i > upper_limit - 1)
   goto labelSquash_1;
 varblock[2][current] = varblock[2][current]+1;

labelStartIteration_1:

 if (iteration == iter1) {
   // j = spec1;
   if (specload_pointer((unsigned char *) &(spec1),
       sizeof (spec1), current, (unsigned char *) &j) == -1)
     earlySquash(1);
 }
 if (iteration == iter2) {
   // j = spec2;
   if (specload_pointer((unsigned char *) &(spec2),
       sizeof (spec2), current, (unsigned char *) &j) == -1)
     earlySquash(1);
 }

 for (k = 0; k < array[i % MAX] + j; k++) {
   if (k >= 179900) {
    // spec1 = k + array[(i + k) % MAX]) % NITER;
    value = k + array[(i + k) % MAX] % NITER;
    specstore_pointer((unsigned char *) &(spec1),sizeof (spec1),current,(unsigned char *) &value);
   }
   if (k <= 1200) {
    // spec2 = array[i % MAX];
    value = array[ i % MAX];
    specstore_pointer((unsigned char *) &(spec2),sizeof (spec2),current,(unsigned char *) &value);
   }
 }

 if (iteration == NITER-1) {
   // spec1 = spec2;
    if (specload_pointer((unsigned char *) &(spec2), sizeof (spec2),
                         current, (unsigned char *) &value) == -1)
       earlySquash(1);
    specstore_pointer((unsigned char *) &(spec1),sizeof (spec1),current,(unsigned char *) &value);
 }

labelEndIteration_1:

 if ((i != varblock[1][current] + ini) && (i < NITER - 1)) {
   i = i + 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
```

Figure 6.7: Code generated by ATLaS after processing the source code of the *Fast* benchmark augmented by BFCA+. For legibility reasons, many of the details of this representation are omitted.

```
#pragma omp parallel for default(none) \
 private(rx,ry,k, yqr, ypr, xqr, xpr, xpq, ypq, lambda1, lambda2, denom, \
 spec_centrox, spec_centroy, spec_cuad_radio, \
 spec_puntdef, spec_solrx, spec_solry) \
 shared(px,py,qx,qy,j,input) \
 speculative(centrox, centroy, cuad_radio, solrx, solry, puntdef)
for (k = 0; k < (j - 1); k++) {
  rx = input[0][k];
  ry = input[1][k];

  spec_centrox = centrox;
  spec_centroy = centroy;
  spec_cuad_radio = cuad_radio;

  if (cuad_dist(rx,ry,spec_centrox,spec_centroy) - spec_cuad_radio > EPS) {

    xpq = px - qx;
    xqr = qx - rx;
    xpr = px - rx;
    ypq = py - qy;
    yqr = qy - ry;
    ypr = py - ry;
    denom = 2 * (xpq * ypr - ypq * xpr);

    if (denom == 0) { printf ("Error: 3 aligned points.\n"); exit (-1); }
    if (py != ry) {
      lambda1 = (yqr * ypr + xpr * xqr) / denom;
      spec_centrox = (px + qx) / 2 + lambda1 * (py - qy);
      spec_centroy = (py + qy) / 2 + lambda1 * (qx - px);
    } else {
      if (py != qy) {
        lambda2 = (yqr * ypq + xqr * xpq) / denom;
        spec_centrox = (px + rx) / 2 + lambda2 * (py - ry);
        spec_centroy = (py + ry) / 2 + lambda2 * (rx - px);
    }}

    spec_cuad_radio = cuad_dist (px, py, spec_centrox, spec_centroy);
    centrox = spec_centrox;
    centroy = spec_centroy;
    cuad_radio = spec_cuad_radio;
    spec_solrx = rx;
    spec_solry = ry;
    spec_puntdef = 3;
    solrx = spec_solrx;
    solry = spec_solry;
    puntdef = spec_puntdef;
  } // end if
}// end for k

...
// Readings of px, py, centrox, centroy, cuad_radio, solrx, solry, puntdef.
```

Figure 6.8: OpenMP constructs generated by BFCA+ for the 2D-MEC application. Performance results of this code are shown in Fig. 4.18 and Table 4.1, page 87.

```
#pragma omp parallel for default(none) \
 private(p, skpslf, eps2, phi0, acc0, pos0, \
  dx, dy, dz, dr2, dr2inv, drinv, phim, \
  dr5inv, q, nbterm, ncterm, sptr, k, aux, \
  phiq, stack) \
 shared(iter, eps, pos, root, mass, rcrit2, \
  usquad, quad, subp, phi, acc) \
 speculative(nbtot, nctot, ntmax)

for (p=1; p<=iter; p++) {
  eps2 = eps*eps;
  phi0 = 0.0;
  acc0[1] = 0.0;
  acc0[2] = 0.0;
  acc0[3] = 0.0;
  pos0[1] = pos[p][1];
  pos0[2] = pos[p][2];
  pos0[3] = pos[p][3];
  nbterm = 0;
  ncterm = 0;
  skpslf = false;
  sptr = 1;
  stack[sptr] = root;

  while (sptr > 0) {
    q = stack[sptr];
    sptr = sptr - 1;

    dx = pos0[1] - pos[q][1];
    dy = pos0[2] - pos[q][2];
    dz = pos0[3] - pos[q][3];
    dr2 = dx*dx + dy*dy + dz*dz;

    if (q < INCELL) {
      if (q != p) {
        dr2inv = 1.0 / (dr2 + eps2);
        phim = mass[q] * sqrt(dr2inv);
        phi0 = phi0 - phim;
        phim = phim * dr2inv;
        acc0[1] -= phim * dx;
        acc0[2] -= phim * dy;
        acc0[3] -= phim * dz;
        nbterm = nbterm + 1;
      } else { skpslf = true; }
    } else {
      if (dr2 >= rcrit2[q]) {
        dr2inv = 1.0 / (dr2 + eps2);
        drinv = sqrt(dr2inv);
        phim = mass[q] * drinv;
        phi0 = phi0 - phim;
        phim = phim * dr2inv;
        acc0[1] -= phim * dx;
        acc0[2] -= phim * dy;
        acc0[3] -= phim * dz;
        if (usquad) {
          dr5inv = dr2inv * dr2inv * drinv;
          phiq = dr5inv *
                 (0.5 * ((dx*dx - dz*dz) *
                  quad[q][1] + (dy*dy - dz*dz)
                        * quad[q][4]) +
                 dx*dy * quad[q][2] + dx*dz *
                 quad[q][3] + dy*dz *
                 quad[q][5]);

          phi0 = phi0 - phiq;
          phiq = 5.0 * phiq * dr2inv;
          acc0[1] -= phiq*dx + dr5inv *
              (dx*quad[q][1] + dy*quad[q][2]
               + dz*quad[q][3]);
          acc0[2] -= phiq*dy + dr5inv *
               (dx*quad[q][2] + dy*quad[q][4]
               + dz*quad[q][5]);
          acc0[3] -= phiq*dz + dr5inv *
               (dx*quad[q][3] + dy*quad[q][5]
               - dz*(quad[q][1] + quad[q][4]));
        }

        ncterm = ncterm + 1;

        } else {
          for (k=1; k<=NSUBC; k++) {
            if (subp[q][k] != NULO) {
              if (sptr >= MXSPTR) {
                terror("TRWALK: STACK OVERFLOW");
              }

              sptr = sptr + 1;
              stack[sptr] = subp[q][k];
            }
          }
        }
      }
    } // while

    if (!skpslf) { terror(" TRWALK: MISSED
                          SELF-INTERACTION"); }
      phi[p] = phi0;
      acc[p][1] = acc0[1];
      acc[p][2] = acc0[2];
      acc[p][3] = acc0[3];
      nbtot = nbtot + nbterm;
      nctot = nctot + ncterm;
      int aux = ncterm+nbterm;
      if (ntmax < aux) {
        ntmax = aux;
      }
} // for (p=1; p<=nbody; p++)
```

Figure 6.9: OpenMP constructs generated by BFCA+ for the TREE application. Performance results of this code are shown in Fig. 4.18 and Table 4.1, page 87.

Chapter 7

Conclusions

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 architectures 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. Speculative parallelization techniques are not an exception, and requires manual and careful intervention by expert programmers.

This Ph.D. thesis addresses this problem defining a new OpenMP clause, called `speculative`, which points out those variables that may lead to dependency violation, and a compile-time system that, using the information on the accesses to variables made by the OpenMP clauses, automatically adds all the extra code lines to handle the speculative execution of a program. This frees programmers from the manual augmentation of the source code required by the speculative parallelization.

Before instrumenting a loop with OpenMP constructs, including our proposed `speculative` clause, programmers firstly need to extract certain information about the source code that they aim to parallelize. Without automatic tools, programmers have to manually extract the information: variable usages within each loop, I/O functions that complicate or even preclude the parallelization, and more important, to determine whether it is worth parallelizing a loop or the thread-management overheads would be larger than the benefit of parallelizing.

This Ph.D. thesis addresses the problem of automatic characterization and coverage of sequential loops, with the aim of finding parallelization niches in widely-used benchmarks that may benefit from software-based speculative parallelization. To do this, we have proposed a system that takes advantage of an XML-based representation of the source and combines profiling information to extract all this information. Be-

sides, we have also proposed a system that leverages this information, automatically synthesizing and generating the OpenMP constructs needed to parallelize the source code speculatively.

We believe that the implementation of the new OpenMP clause in a mainstream compiler, together with the automation of the whole process of the parallelization, will help thread-level speculation to be mature enough for its inclusion in production tools.

7.1 Summary of results and contributions

The contributions of this Ph.D. thesis, ordered by goals, are the following. Publications associated with each goal are also listed.

7.1.1 Goal 1: Evaluation of compilers parallelization capabilities

We have measured the parallelization capabilities of commercial compilers, uncovering the limitations of their automatic parallelization techniques. The study revels that automatic parallelization only achieves a 19% of speed-up on average for SPEC CPU2006 benchmarks.

1. Evaluación de compiladores comerciales usando SPEC CPU2006. Sergio Aldea, Diego R. Llanos, Arturo Gonzalez–Escribano. *Actas XIX Jornadas de Paralelismo*, Castellón, Spain, September 17–19, 2008.

2. Using SPEC CPU2006 to Evaluate the Secuential and Parallel Code Generated by Commercial and Open-source Compilers. Sergio Aldea, Diego R. Llanos, Arturo Gonzalez-Escribano. *The Journal of Supercomputing, 59(1)*, January 2012, pages 486-498.

7.1.2 Goal 2: OpenMP `speculative` clause proposal and definition

We have added support for TLS into OpenMP. To achieve this goal, we have proposed a new OpenMP clause to handle those variables that could lead to any dependency violation. This new clause is called `speculative`. This new clause would allow executing in parallel loops whose dependency analysis cannot be done at compile time.

3. Support for thread-level speculation into OpenMP. Sergio Aldea, Diego R. Llanos, and Arturo Gonzalez-Escribano. *Proceedings of the 8th international conference on OpenMP in a Heterogeneous World (IWOMP'12)*, Barbara M. Chap-

man, Federico Massaioli, Matthias S. Müller, and Marco Rorro (Eds.). Springer-Verlag, Berlin, Heidelberg, 2012. pages 275-278.

7.1.3 Goal 3: OpenMP `speculative` clause design, implementation and evaluation

In order to implement the clause proposed in the previous goal, we have developed a GCC plugin-based compiler pass to support the new clause `speculative` into the GCC's OpenMP implementation. This pass transforms the loop with the corresponding `omp parallel for` directive, inserting the runtime TLS calls needed to (a) distribute blocks of iterations amongs processors, (b) perform speculative loads and stores of speculative variables (pointed out using the new clause), and (c) perform partial commits of the correct results calculated so far. The TLS runtime library used in this Ph.D. thesis have been developed by Estebanez, García-Yágüez, Llanos, and Gonzalez-Escribano [62, 69], using the same design principles than the speculative parallelization library developed by Cintra and Llanos [36, 37].

Moreover, we have augmented the existing documentation about GCC plugins. We have found that the documentation about building plugins is scarce. Therefore, we have described how to build, link to GCC, and execute a plugin. We also have described the plugin internal structure and helpful details for programmers.

Finally, we have evaluated the capabilities of the proposed clause and the compile-time system that uses that clause to automatically parallelize real-world applications speculatively. The code generated automatically not only achieves speedups in applications that are not parallelizable at compile-time by conventional automatic techniques, but also performs better than the code parallelized manually. The speedups achieved by the automatic approach are around 20% better than the speedups obtained by the manual approach. With the new OpenMP `speculative` clause, programmers can parallelize those applications avoiding all the hurdles involved in the manual-speculative parallelization.

4. A New GCC Plugin-Based Compiler Pass to Add Support for Thread-Level Speculation into OpenMP. Sergio Aldea, Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. Accepted in Euro-Par 2014. Volume 8632 of Lecture Notes of Computer Science. To appear.

5. An OpenMP extension that supports Thread-Level Speculation. Sergio Aldea, Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. Submitted to IEEE Transactions on Parallel and Distributed Systems in April 2014.

6. Una extensión para OpenMP que soporta paralelización especulativa. Sergio Aldea, Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano.

Actas XXV Jornadas de Paralelismo, Valladolid, Spain, September, 17-19, 2014.

7.1.4 Goal 4: Speculative parallelization niches detection and variable classification

This Ph.D thesis proposes a system, called BonaFide C Analyzer (BFCA), 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 analyzing how particular language constructs are used in real-word applications, and helps the programmer to parallelize the code, including the runtime library used in this Ph.D. thesis. This information can be processed to automatically define OpenMP clauses, including our clause proposed in the second goal, in order to point out the *speculative* variables.

Using BFCA, we have conducted an extensive study of the C applications present in the SPEC CPU2006 benchmark suite [82]. The study does not only characterize in both quantitative and qualitative terms the loops of these applications regarding their suitability for parallel execution. 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.

7. Extending a source−to−source compiler with XML capabilities. Sergio Aldea, Diego R. Llanos, Arturo Gonzalez−Escribano. *Actas XXI Jornadas de Paralelismo*, Valencia, Spain, September 7−10, 2010.

8. XMLCetus y Sirius: análisis y traducción de código C utilizando herramientas XML. Sergio Aldea, Diego R. Llanos, Arturo Gonzales-Escribano. *Technical Report IT-DI-2010-001*, Department of Computer Science, Universidad de Valladolid, 2010.

9. Towards a compiler framework for thread-level speculation. Sergio Aldea, Diego R. Llanos, Arturo Gonzalez-Escribano. *Proceedings of the 19th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP 2011)*, Ayia Napa, Cyprus, February 9-11, 2011. pages 267–271.

10. The BonaFide C Analyzer: Automatic Loop-level Characterization and Coverage Measurement. Sergio Aldea, Diego R. Llanos, and Arturo Gonzalez-Escribano. *The Journal of Supercomputing*, 2014. Online, DOI:10.1007/s11227-014-1091-3.

7.1.5 Goal 5: Automatic synthesis of speculative code

We have developed a solution that leverages the BFCA's classification of all variables used inside each loop to automatically generate an OpenMP-based parallel version of the loop, using the *shared*, *private* and the proposed *speculative* clause. The resulting framework, called BFCA+, frees the programmers from a manual intervention to modify and instrument the source code with such OpenMP constructs, which is an error-prone and often tedious task.

As a result, the target loop will be guaranteed to correctly run in parallel, with a parallel performance that will depend on the actual number of dependency violations that will arise at runtime. This solution can be also useful to instrument a parallel loop with the OpenMP standard.

This work will be submitted for publication during the year 2014.

7.2 Answer to the research question

> *Is it possible to develop a compile-time mechanism able to (1) detect susceptible niches for speculative parallelization, (2) evaluate their impact in terms of parallel execution time, and (3) automatically transform such sequential source code into a parallel speculative version?*

The first two questions are affirmatively answered with the help of BFCA. Using this tool we have classified *FOR* loops from SPEC CPU2006 benchmarks. It is obtained that on average a 37.7% of the *FOR* loops are potentially speculatively parallelizable loops, and moreover, they have a coverage of 28.34% on average. Therefore, they represent more than a quarter of the total execution time, which is a significant time. Extraction of these two numbers demonstrates that (1) it is possible to detect susceptible niches for speculative parallelization, and (2) it is also possible to evaluate their impact in terms of execution time. Determining whether a loop is potentially parallelizable is quite hard due to the large number of variables that are involved. For this reason, our proposed solution, which allows performing this task automatically, has a great value.

Regarding the third question, we have developed a compile-time system, ATLaS, that proves that it is possible to automatically transform a sequential source code into a speculatively parallel one. This transformation has been made in several steps.

First, using the characterization of the *FOR* loops, BFCA+ automatically augments the source with OpenMP constructs, including our proposed `speculative` clause. This clause is then parsed by ATLaS to generate all the code needed to handle a speculatively parallel execution of the source code. This process does not require any other manual intervention by the programmer than the selection of the loop to be parallelized. Therefore, the transformation from a sequential version of the code into a parallel one is performed automatically, including the instrumentation of the code with OpenMP constructs, and the generation of the additional code to handle the speculative execution.

7.3 Future directions

There are still some issues that we would like to address, which define the future directions for this research:

- To build heuristics to automatically choose target loops. With these heuristics, programmers would avoid deciding which loop is more profitable to be parallelized, relying on BFCA this decision, and bypassing any manual intervention.
- To extend our solution beyond `FOR` loops, in order to be able to recognize, analyze, and parallelize any region in the source code susceptible to be parallelizable.
- To develop a source-to-source compiler to parse the source the code and directly generate the XML representation, avoiding the dependency with a third-party software such as Cetus.
- With all these points solved, to develop a compiler that automatically parallelize loops or any parallel region in the source code, which would be chosen following built-in heuristics. This compiler would generate the final binary executable, that depending on the parallel region would be executed in parallel speculatively, or using the OpenMP standard runtime system.

Appendix A

Related Technologies

The development of the prototypes presented in this Ph.D. thesis requires various technologies. As a starting point for BFCA, Cetus provides an API and a very interesting underlying architecture for our purposes. Cetus builds an internal representation of a C code that we can use to easily create an XML tree that represents it. Therefore, this chapter resumes both kind of technologies used. On the one hand, Cetus as the first stone of the development. On the other hand, XML technologies as the tools that extract all the functionalities that we demand in the construction of the framework.

A.1 Cetus

Cetus [54] has already been introduced in Chap. 5. Since the first subsystem of BFCA, called XMLCetus, is a modification of the original Cetus program, it constitutes the starting point of the developing of BFCA.

In order to clarify some aspects regarding XMLCetus and its operation, this section has the aim of briefly explaining some concepts and essential knowledge about Cetus.

A.1.1 Essential knowledge about Cetus

There are some essential concepts required for the construction of BFCA. This section describes several of them.

Major classes

The main kind of blocks are described below. A further description of these classes and their methods can be found in the API provided by the Cetus Project [30].

- **Program**: A *Program* object represents an entire program, which can embrace one or various *TranslationUnit*. This node is the root of the syntactic tree build by Cetus.
- **TranslationUnit**: A *TranslationUnit* object represents a unique source file. It only appears as direct descendant of the *Program* node.
- **Procedures**: These objects represent each function defined in a C program. When these functions are called from the code, they are represented as *FunctionCall* nodes.
- **Declarations**: These objects appear in many places of the tree. As descendants of a *TranslationUnit*, they represent global declarations of variables, functions or even classes. As parameters of *Procedures*, they represent declarations of formal parameters. And as children of *ClassDeclaration* nodes, they represent methods and data members.
- **Expressions**: *Expressions* objects represent assignments, function calls and mathematical computations.
- **Statements**: These nodes have two purposes: To provide control constructs, and to provide wrappers for *Declarations* and *Expressions*. An example of the first purpose is the *IfStatements*, whereas *DeclarationStatement* and *ExpressionStatement* are examples of the second one.
- **Specifiers**: These objects are generally used in lists. For example, a variable declaration may be prefixed with a list of specifiers such as *const* and *float*.

IR-node functions

Cetus provides an API that is very useful to manipulate nodes in the Intermediate Representation, and to get information about a particular node, its parent node or its child nodes. The most useful functions are the "get-type" functions, and they will be used to build XMLCetus. Figure A.1 shows an usage example of these functions.

Syntax tree invariants

There are several syntax tree invariants which the IR tree satisfies. Knowledge of these invariants was essential to develop XMLCetus. These invariants are three:

Invariant 1 If an object has a parent, then it has exactly one parent. Consequences are:

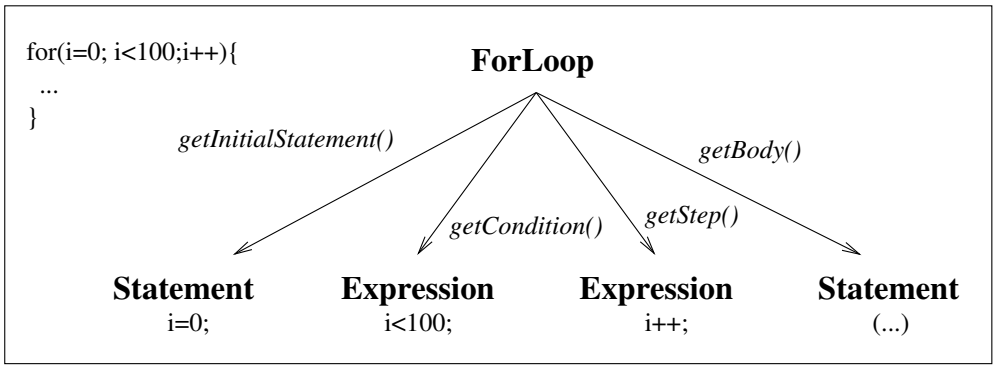

Figure A.1: *get()* functions in a *ForLoop* node.

1. You may not take an object that has a parent and add it as the child of another object.
2. If you want to use the same object in more than one place in the syntax tree, you must clone the original object.
3. Cloned objects are identical to the originals except their parent is null.

Invariant 2 An object *O* can enumerate all of its children. Consequence is:

1. An iterator object can be created with *O* that can iterate over *O*'s children.

Invariant 3 An object controls the addition and removal of its children.

1. An object cannot become the child of another object without its permission.
2. Before an object can set its parent reference to another object, this last one must already recognize the first is a child. Namely, the object must already in the list of the children of the other object.
3. The child reference and parent reference must be set in that order.

A.1.2 Simple example of Cetus running

Figure A.2 shows the transformation of a simple C code that Cetus performs if the *-parallelize-loops* option enabled. The main options which can be enabled in the Cetus running are following explained:

-parallelize-loops : It annotates loops with parallel directives. This option includes the activation of several options: -ddt, -privatize, -reduction, -induction, and -ompGen.

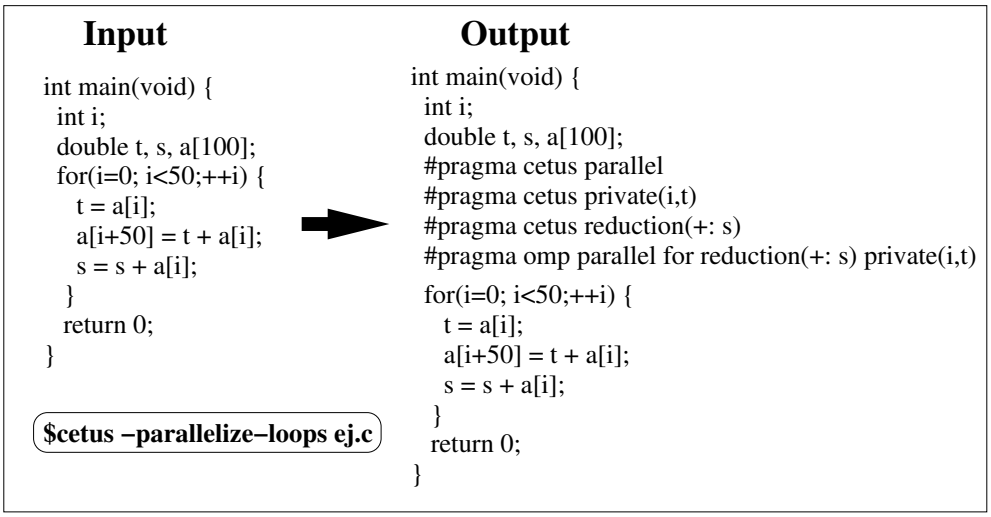

Figure A.2: Example of Cetus running.

-ddt It enables the Data Dependence Testing.

-privatize It performs the scalar/array privatization analysis.

-reduction It performs the reduction variable analysis.

-induction It performs the induction variable substitution.

-ompGen It generates OpenMP pragmas.

-verbosity=N Degree of status messages (0-4) that the user wish to see.

Activation of *-parallelize-loops* option causes the transformations seen in Fig. A.2. In this figure, we can note two kind of directives. One type is OpenMP's directives, as consequence of *-ompGen* option, and the other is internal to Cetus, that annotates loops with information that Cetus will use to create automatically the OpenMP pragmas. Note that the generation of the OpenMP pragmas can only be enabled if Cetus is able to determine whether the code is parallelizable at compile time. Once we have described how Cetus works in the surface, we are prepared to understand Cetus in more depth.

A.1.3 Class hierarchy design and Intermediate Representation

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 hier-

Figure A.3: Some nodes of the Cetus IR.

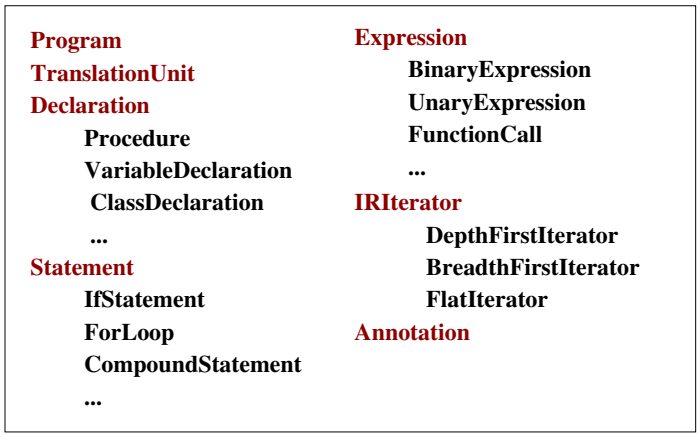

Figure A.4: Cetus hierarchy.

archy and accessed through its 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.

The Cetus IR (Fig. A.3) has been built through a set of classes, which fit within a class hierarchy (Fig. A.4). 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.

Figure A.5 shows an example of Cetus IR from a C source code. There are two basic building components in Cetus: *TranslationUnit* blocks, each one representing a

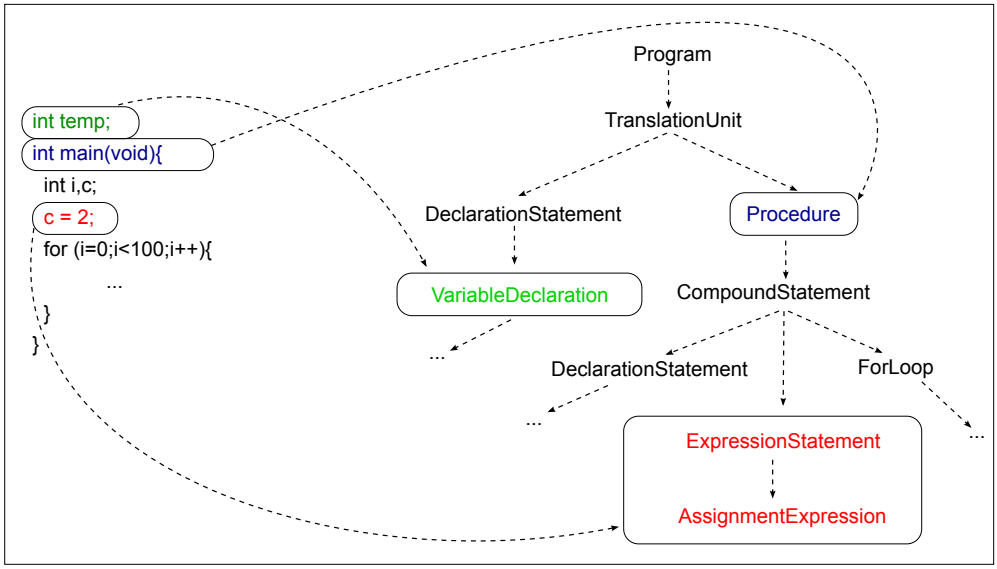

Figure A.5: IR Tree Structure Example.

different source file; and *Procedures*, representing individual functions. Each *Procedure* is composed of a list of simple or compound *Statements* that represent the flow control of the program in a hierarchy form. Namely, there are compound *Statements* as *IF* blocks or *FOR* loop that include within them other *Statements* (simples or compounds), which reference, for example, *THEN* and *ELSE* blocks, or the body of a loop. *Expression* blocks represent operations, such as variable assignments.

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 the compiler framework explained in Chap. 5, we have modified Cetus to build an XML representation of its Intermediate Representation, and we use XML standard tools to perform queries and modifications to the structure.

A.1.4 Cetus known problems

Experience using and modifying Cetus has led us to understand better its internal operation and to discover some errors that they are worthwhile to note.

- **Line numbers**: Cetus IR does not maintain the line number of some statements, e.g. the line number of variable declarations. This is a problem in order to analyze the code, because is not always possible to use the statement line

number as reference. Since essential elements of the analysis are *FOR* loops, line numbers in this nodes are critical. However, Cetus, as its programmers recognized, does not maintain the correct line number in many statements. They do not provide more details, thus experimentation has been necessary to find the source of the problem. After performing several tests we find what the problem is: Whenever a `#include` statement is within a `#ifdef` preprocessor directive, lines following this directive are wrong. This error is also detected when an `#include` is situated within a "`/* */`" comment block.

Since a modification of the Cetus grammar was not possible due to its complexity, the solution to this problem is to rewrite code, commenting/eliminating lines depending on the condition of the `#ifdef`. Namely, code is modified as whether it has not these preprocessor directives (just those that contains `#include`). We can see below a real example of this error, extracted from a SPEC CPU2000 benchmark, and how to correct it.

Original

```
#ifdef NO_TIME_H
  #include <sys/time.h>
#else
  #include <time.h>
#endif

#ifndef NO_FCNTL_H
  #include <fcntl.h>
#endif
```

Correction

```
//#ifdef NO_TIME_H
//  #include <sys/time.h>
//#else
    #include <time.h>
//#endif

//#ifndef NO_FCNTL_H
    #include <fcntl.h>
//#endif
```

Since `NO_TIME_H` and `NO_FCNTL_H` are not defined in the code neither passed by the compiler, we can rewrite these lines as above to avoid the error with line numbers.

If the choice is to comment lines, it is mandatory to comment with "`//`". In the case of `#include` within a comment block, it is also necessary to re-comment this line with "`//`".

- **Definitions within `#ifdef`**: Source codes transformed by Cetus and the Cetus IR do not preserve preprocessor directives. Cetus only preserves those lines that satisfy conditions depending on the definitions. As some of these definitions are passed by the compiler, and thus not defined in the code, Cetus removes some lines incorrectly. To solve this problem, it is necessary to write these definitions in a header file, e.g. named `compiler.h`, instead of passing them by the

compiler. In this way, Cetus does not remove those lines inside `#ifdef` that are satisfied. Below a real example can be seen, extracted from a SPEC CPU2006 benchmark. If the compiler is executed with the following definitions:

```
$ gcc -g -DSPEC_BZIP -DSPEC_CPU -DBZ_UNIX bzip2.c -o 401bzip2
```

the header file must contain the following definitions.

```
1 #define SPEC_BZIP
2 #define SPEC_CPU
3 #define BZ_UNIX
```

- **Same `#includes` preceded by different definitions**: In some cases in source codes of SPEC CPU, a same `#include` is presented twice or more times in source codes, but with different previous definitions, which condition the header because there are some `#ifdef` within it. Therefore, in each inclusion of the header, the code's behavior is different. Since Cetus does not conserve `#define` definitions (Cetus resolves their names in their usage), when it creates the new code with the Cetus IR, such source codes are wrong. Fortunately, this is an uncommon practice in SPEC CPU benchmarks, but it still remains without solution.

- **Source files prefixed with numbers**: Cetus renames some structures with words prefixed with the source file name. If one of these source files begin with a number, structures renaming will be prefixed with a number, and compiler will fail because variables or constants cannot prefixed with a number.

A.2 XML technologies

The prototype BFCA proposed in this dissertation makes an intensive use of XML technologies. There are several reasons to use XML [23]. Using XML, we can represent source code in a structure form and easily build an XML tree from Cetus IR thanks to a Java API. Once a source code is represented as an XML tree, it is possible to use all the tools provided by the community to manipulate, exchange, transform (XSLT), and search (XPath or XQuery). In order to develop the framework and its subsystems, XPath and XSLT are essential to make easy and complex searches in the source code, and to transform the XML representation back into C language again to check the correctiveness of the representation and changes performed on it.

```
<bookstore>
  <book category="Fantasy">
    <title lang="en">A Game of Thrones</title>
    <author>George R. R. Martin</author>
    <year>2003</year>
    <price>15</price>
  </book>
  <book category="Children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="Business">
    <title lang="en">What Would Google Do?</title>
    <author>Jeff Jarvis</author>
    <year>2009</year>
    <price>20</price>
  </book>
</bookstore>
```

Figure A.6: Example of XML document. A bookstore has three books, whose category is expressed as attribute of the book node. By nesting tags we indicate the title, author, year and price of each book.

A.2.1 XML basics

XML, designed by the World Wide Web Consortium (W3C), allows the specification of user-defined markup tags adapted to the content of the document. This property is extremely valuable because it gives freedom to programmers to build XML treed. In the case of representing source code using XML, this freedom means flexibility to represents any programming language. After all, programmers can represent any kind of information with XML.

In broad outlines, an XML document are composed by a set of tags which have a structure and a hierarchical relationship between them. XML tags are not predefined, and the programmer must define his/her own tags. A visual description of an XML example is shown in Figure A.6.

A.2.2 XPath

XPath is the XML Path Language [18]. It is a query language that allows selection of nodes depending on routes and conditions. These routes are expressed as absolute paths from the first node of the XML tree, or as relative paths from a particular position in the tree. Moreover, it is possible to indicate conditions that narrow search

conditions. XPath provides the feature of navigating the XML tree and selecting nodes by different criteria.

Current version of XPath is 2.0, and it is the version used in the framework. Reason of using this version is the expanded set of functions and operators. In fact, XPath is a subset of XQuery 1.0, which is a more complex and complete language to query collections of XML data. More information about the XPath syntax and its usage is on the cited bibliography.

A.2.3 XSLT

The main functionality of the developed framework is to obtain an XML representation of a C source code, and to used this representation to extract information that otherwise would be more complicate to obtain. However, building an XML tree that represents a particular C source code requires a validation process. It is necessary to be sure that this XML tree matches at 100% in terms of functionality with the original C source code. In order to check this situation, the use of a transformation language as XSLT [95] is essential.

XSLT (Extensible Stylesheet Language Transformation) is an XML-based language that transforms XML files into different representations. XML provides the information and XSLT shapes the form of its representation. In this way, it is possible to transform a XML file that represents a C code into an equivalent C source file. Therefore, back to the framework, using XSLT it is possible to transform the XML tree back into C, and check if this C file is correct, compile, execute, and has the same functionality than the original program.

A.2.4 Saxon

Saxon [94] is an XSLT and XQuery processor created by Michael Kay. Current developing line, Saxon 9 (last version is 9.3), implements XSLT 2.0 and XQuery 1.0. Saxon offers different versions depending on the user's needs. Version used in this project is Saxon-B 9.1.0.1, since it has proven enough, and new versions require paid licenses and do not offer new functionalities that we could be interested. The version used is free, and provides the basic conformity level to these languages (XSLT and XQuery), allowing to use all features excepting XML Schema processing. Saxon's code is written in Java and thus can be used from Java programs, as Cetus and the subsystems developed as part of the prototype BFCA.

Appendix B

GCC Details for Plugin Development

This appendix gathers some of the knowledge acquired before and during the development of the GCC plugin presented in this Ph.D. thesis. Part of this knowledge can be found in the GCC Internals [71], but it has been significantly augmented with information extracted directly from the source code.

B.1 GCC passes

All the transformations that GCC performs on the source code are organized in passes. Each pass performs on the code in a particular intermediate representation. Developing a plugin requires deciding which kind of pass it will be. Figure B.1 shows a summarized view of all the passes and the different intermediate representation which GCC works with. This figure reveals some kind of organization between passes.

GCC passes are hierarchically grouped in families. These families are implemented in the GCC source code using structures. The list of these structures can be found in `tree-pass.h`. A brief description of some of these passes can be found in the internal documentation of GCC [1].

- `all_lowering_passes`: This structure gathers all the passes needed to lower the code into a shape in which optimizers can work. Among these passes we find the pass that builds the Control Flow Graph (CFG), or the pass that lowers the OpenMP constructs into GIMPLE.

- `all_small_ipa_passes`: It keeps all the small interprocedural optimization passes, which do all their process at once, without splitting it into several stages

[1] See `http://gcc.gnu.org/onlinedocs/gccint/Passes.html`.

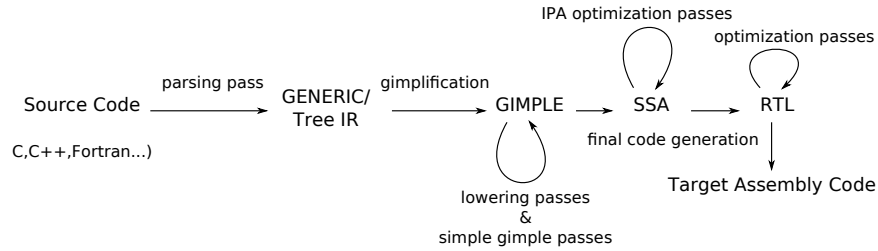

Figure B.1: GCC passes and the different representations

as a regular IPA pass does. Among these passes we find the pass that expands parallel regions defined using OpenMP into their own functions to be invoked by the thread library, or the pass that builds the SSA representation.

- `all_regular_ipa_passes`: It gathers the rest of interprocedural optimization passes. For example, one of these passes analyzes the control graph and decides the inlining plan.
- `all_lto_gen_passes`: This structure keeps the two Link Time Optimization (LTO) passes [2]. LTO enables optimization passes across several compilation units. For example, LTO enables to make inlining of a function defined in `foo.c` and called in `bar.c`.
- `all_passes`: It gathers all the GIMPLE scalar and loop optimizations, and the RTL optimizations.
- `current_pass`: It represents the current pass.

The passes of these groups are scheduled in the function `init_optimization_passes()` of `passes.c`, in the same order that they have been described above.

B.1.1 Pass description

Each pass is described using the following structure, located in `tree-pass.h`:

```
struct opt_pass
{
  /* Optimization pass type.  */
  enum opt_pass_type type;
```

[2] See `http://gcc.gnu.org/onlinedocs/gccint/LTO.html` for more details

```
  /* Name of the pass, used as a fragment of the dump file name.  */
  const char *name;

  /* Gate function.  */
  bool (*gate) (void);

  /* Execute function.  */
  unsigned int (*execute) (void);

  /* A list of sub-passes to run, dependent on the gate predicate.  */
  struct opt_pass *sub;

 /* Next in the list of passes to run, independent of the gate predicate. */
  struct opt_pass *next;

  /* Static pass number, used as a fragment of the dump file name.  */
  int static_pass_number;

  /* The timevar id associated with this pass.  */
  timevar_id_t tv_id;

  /* Sets of properties input and output from this pass.  */
  unsigned int properties_required;
  unsigned int properties_provided;
  unsigned int properties_destroyed;

  /* Flags indicating common sets things to do before and after.  */
  unsigned int todo_flags_start;
  unsigned int todo_flags_finish;
};
```

This structure has several fields with a different meaning each one:

- `type`: This is the type of the pass. It must be one of the following:
  - `GIMPLE_PASS`: This is a pass that works with the source code in GIMPLE representation.
  - `IPA_PASS`: This is an interprocedural pass, namely, a pass that works the entire program, and not just a single function or a single block of code.

  - SIMPLE_IPA_PASS: This is a small interprocedural pass. The difference between a regular and a small inter-procedural pass is that the latter does everything at once, without splitting its process into several stages as a regular IPA pass does. This kind of pass is useful for easier prototyping and development of a new inter-procedural pass.
  - RTL_PASS: This is a pass that works with the source code in RTL representation.

- *name: The name of the pass. This name is important because it is used to reference the pass, for example, in the passes scheduling, or at the moment of adding a new pass before or after a certain pass.

- *gate(): The gate function. This is the function that is executed before the pass operation. It could be use to enable/disable the pass. If the return value is true, the "execute function" is executed; otherwise, its execution is skipped.

- *execute(): The execute function. This is the function called by the pass and implements the pass operation. The return value could contain TODOs to execute in addition to those in todo_flags_finish.

- *sub: Passes may be organized hierarchically. sub points to the first child pass. Its execution depends on the value returned by the gate function, because these passes are related.

- *next: This structure points to the next pass in the list of passes, as they were scheduled in the function init_optimization_passes() of passes.c. Its execution does not depend on the value returned by the gate function, because the passes are independent.

- static_pass_number: It is automatically generated by the pass manager, and it is used in the dump file.

- tv_id: We use this variable to set a separate timer for the pass. This timer is automatically started and stopped by the pass manager. The list of timers is located in timevar.def.

- properties_required: The list of properties required for this pass.

- properties_provided: The list of properties that the source code satisfied after the pass.

- properties_destroyed: The list of properties that the pass destroys.

- `todo_flags_start`: List of actions that the pass manager should carry out before the pass execution.
- `todo_flags_finish`: List of actions that the pass manager should carry out after the pass execution.

Pass properties

In the pass descriptor, there are three fields that set up some properties related with state or transformations done in the source code. The list of properties that a pass can require, provide or destroy is located in `tree-pass.h`. They are the following:

- `PROP_gimple_any`: This property ensures that it is allowed a full GIMPLE grammar.
- `PROP_gimple_lcf`: This property means that the control flow has been lowered.
- `PROP_gimple_leh`: This means that the exception-handling has been lowered.
- `PROP_cfg`: This property ensures that the function treated by the pass has a non-none control flow graph (CFG).
- `PROP_referenced_vars`: This property means that we have data about the variables referenced in the function. This property is set up by the Dataflow Analysis (DFA), which finds all the variables referenced in the function.
- `PROP_ssa`: This means that the GIMPLE tree is in SSA form.
- `PROP_no_crit_edges`: This property is satisfied when all the critical edges of the CFG have been split.
- `PROP_rtl`: This means that the function or block of code treated by the pass is in RTL form.
- `PROP_gimple_lomp`: This property ensures that all the OpenMP directives have been lowered into explicit calls to the runtime library (libgomp).
- `PROP_cfglayout`: This property means that the CFG has been organized into a more efficient order. It is used in the RTL representation.
- `PROP_gimple_lcx`: This means that operations with complex numbers have been lowered to scalar operations.
- `PROP_trees`: This is a property that gathers the following four properties: `PROP_gimple_any`, `PROP_gimple_lcf`, `PROP_gimple_leh`, and `PROP_gimple_lomp`.

TODO flags

A `TODO` flag describes an action that the pass manager should carry out before or after the pass execution. The list of `TODO` flags is located in the file `tree-pass.h`, and tells the pass manager to carry out the following actions:

- `TODO_dump_func`: This flag tells the pass manager to dump an "image" of the current state of the source code into a file. This flag only has effect when the compiler is executed with some `-fdump-xxx` option.
- `TODO_ggc_collect`: It tells the pass manager to do the garbage collection.
- `TODO_verify_ssa`: It verifies if the GIMPLE representation is in a correct SSA form.
- `TODO_verify_flow`: It verifies that the flow information in the control flow graph (CFG) is correct. This is used by passes that modify the flow and need to check if everything is correct.
- `TODO_verify_stmts`: It verifies if all the statements are correctly built. This is used by passes that manually add new statements into the source code.
- `TODO_cleanup_cfg`: This flag tells the pass manager to cleanup the CFG, erasing unreachable edges or blocks of code.
- `TODO_dump_cgraph`: This flag tells the pass manager to dump the callgraph into a debugging file.
- `TODO_remove_functions`: Used by IPA passes, it removes functions just as before inlining. IPA passes might be interested to see bodies of extern inline functions that are not inlined to analyze side effects. The full removal is done just at the end of IPA pass queue.
- `TODO_rebuild_frequencies`: It rebuilds function frequencies. Passes are in general expected to maintain profile by hand, however in some cases this is not possible: For example, when inlining several functions with loops frequencies might run out of scale and thus needs to be recomputed.
- `TODO_verify_rtl_sharing`: Go through all the RTL insn (doubly-linked chain of objects that represents a function) bodies and check that there is no unexpected sharing in between the subexpressions.

- `TODO_update_ssa`: When a pass creates new symbols, these symbols need to be renamed and it is also necessary to do the mapping between the old and the new names registered. This flag updates the SSA form, inserting PHI nodes for newly exposed symbols and virtual names marked for updating, and also pruning the excess of PHI nodes.

- `TODO_update_ssa_no_phi`: It updates the SSA form without inserting any new PHI nodes. It is usually used by passes that insert themselves all the PHI nodes.

- `TODO_update_ssa_full_phi`: It updates the SSA form inserting PHI nodes everywhere they are needed, without doing a prune.

- `TODO_update_ssa_only_virtuals`: This flags only processes the symbols that are marked to be renamed. The rest of old to new mappings for real names are explicitly destroyed.

- `TODO_remove_unused_locals`: Some passes generate local variables that remain unused after the pass execution. This flag removes these local variables from `cfun->local_decls`, which a structure that stores the local declarations of the current function being compiled. This action reduces the size of dump files and the memory footprint for `VAR_DECLs`.

- `TODO_df_finish`: Call `df_finish_pass()` at the end of the pass. This function is located in `df-core.c`, and removes all the problems related with the Dataflow Analysis (DFA). This function is the last action to do before finishing the DFA.

- `TODO_df_verify`: It calls `df_verify()` at the end of the pass if checking is enabled. This function is located in `df-core.c`, and verifies that there is a place for everything and everything is in its place, from the point of view of the DFA.

- `TODO_mark_first_instance`: This flag is internally used for the first instance of a pass.

- `TODO_rebuild_alias`: It rebuilds the aliasing information, which means that the alias analysis is done again.

- `TODO_update_address_taken`: It rebuilds the addressable-vars bitmap and does the register promotion.

- `TODO_rebuild_cgraph_edges`: It rebuilds the callgraph edges.

- `TODO_update_ssa_any`: It gathers the following `TODO`s: `TODO_update_ssa`, `TODO_update_ssa_no_phi`, `TODO_update_ssa_full_phi`, and `TODO_update_ssa_only_virtuals`. Internally used in `execute_function_todo()`, function from `passes.c` that performs all `TODO` actions that should to be done on each function.

- `TODO_verify_all`: It gathers the following `TODO`s: `TODO_verify_ssa`, `TODO_verify_flow`, and `TODO_verify_stmts`.

B.1.2 Dump files generated by the passes

Passes may only monitor the source code, but they usually make transformations into the code. These transformations and changes can be seen if we activate the flag `TODO_dump_func` in the `todo_flags_finish` field of the pass descriptor, and we also enable the flag `-fdump-tree-all`[3] in the GCC compilation. As a result of these two actions, GCC generates a dump file which contains the state of the source code just after the changes are done by the pass. The file name for the dump file generated is

```
<source-file>.<static_pass_number>[itr].<pass-name>
```

where `i` represents an IPA pass, `t` represents a GIMPLE tree pass, and `r` is for RTL passes. For example, the pass `omplower`, which is a `GIMPLE_PASS`, generates the following dump file: `examplefile.c.009t.omplower`.

The `static_pass_number` is automatically generated by the pass manager, and it may not match with the order in which the passes are executed.

B.2 GCC plugins structure

A plugin has a well-defined structure:

- **License check**: This is more a "political" part of the plugin rather than functional. Every plugin has to define the following global variable to assert that the plugin has been licensed under a GPL-compatible license.

```
1 int plugin_is_GPL_compatible = 1;
```

[3]This option enables all the dump files generated by passes that process the intermediate language tree, the GIMPLE passes. See `http://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html` for more debugging options.

This requirement is mandatory for every plugin. Once a plugin is loaded into GCC, and before the plugin begins its operation, the compiler checks for the existence of this symbol in the global scope. If it exists, then the plugin can run. Otherwise, the compiler emits a fatal error, and the execution is aborted:

```
cc1: fatal error: plugin path/to/plugin/name.so is not licensed
  under a GPL-compatible licensed
path/to/plugin/name.so: undefined symbol: plugin_is_GPL_compatible
compilation terminated
```

- **Plugin headers**: The first header that a plugin has to include is `gcc-plugin.h`, which contains functions and structures that are essential for the plugin. After this header, we can include any of the headers located in the plugin directory [4]. The files in there are the GCC API accessible to your plugin. For example, if we are dealing with GIMPLE nodes, we need to include `gimple.h`, or if we have to access to GENERIC nodes and trees, we need to include `tree.h`.

- **Plugin information**: Every plugin should specify some information about itself, in case users ask for help using `gcc -v` or `gcc -help`. We can specify this information using the structure `plugin_info` defined in `gcc-plugin.h`:

```
1 /* Information about the plugin */
2 static struct plugin_info myplugin_info =
3 {
4   .version = "001",
5   .help = "Work in progress",
6 };
```

- **GCC version checking**: Each plugin is written using the API defined by a particular version of GCC. This API could not be the same between different versions, and therefore, the plugin could not work with a different version of GCC. For this reason, plugins need to declare which version of GCC they work for. This information is defined using the structure `plugin_gcc_version`:

```
1 /* This plugin works for GCC version 4.6.2 */
2 static struct plugin_gcc_version myplugin_ver =
3 {
4   .basever = "4.6.2",
5   .datestamp = "20111026";
6   .devphase = "";
7   .revision = "";
8 };
```

[4] To see where the "plugin" directory is located, execute `gcc -print-file-name=plugin`.

For a particular version of GCC, in file `plugin-version.h` located in the plugin directory of GCC, we can find the information that we have to write in each field of this structure. As a plugin may not run correctly with a different version of GCC, we need to validate during the plugin initialization if the plugin is loaded in the correct version of GCC. There are two ways to check this. The first one is less restrictive, and only consists in a string comparison between the fields of two structures; one defined by the user, and other defined in `plugin-version.h`:

```
int plugin_init (struct plugin_name_args *info,
                 struct plugin_gcc_version *version)
...
  /* Validate the plugin with the correct version of GCC.  */
  if (strncmp(version->basever, myplugin_ver.basever, strlen("4.6.2")
      ))
    return -1; /* Incorrect version of GCC */
```

The second way to validate our plugin is using the function `plugin_default_version_check()`, which is a more strict check. This function compares both structures field by field, and therefore, it is mandatory that not only the GCC version, but also the datestamp and the other fields match. In order to use it, we need to include the header `plugin-version.h`.

```
#include <plugin-version.h>
...
int plugin_init (struct plugin_name_args *info,
                 struct plugin_gcc_version *version)
...
  if (!plugin_default_version_check (version, &gcc_version))
    return 1;
```

- **Plugin initialization**: The first function that is called after the plugin is loaded is `plugin_init()`, and it is the responsible for doing the initialization and registering all the callbacks required.

```
/* Return 0 on success or error code on failure.  */
int
plugin_init(struct plugin_name_args *info, // Argument information
            struct plugin_gcc_version *ver) // Version info of GCC
```

This function has to return 0 if the initialization has been done correctly, and otherwise, it has to return a non-zero value.

In the initialization, we establish in which moment the plugin is activated. A plugin is activated by the compiler at specific events as defined in `gcc-plugin.h`.

In those events in which we could be interested, the plugin should call the function `register_callback()` specifying the name of the event and address of the callback function that will handle that particular event. Using `register_callback()` we can set in which point the plugin is called in the succession of passes, or set the information about the plugin:

```
struct register_pass_info pass;

/* Information about the pass implemented,
   from the pass descriptor.  */
pass.pass = &myplugin_pass.pass;

/* Tell GCC that we want to be called after the first SSA pass.  */
pass.reference_pass_name = "ssa";
pass.ref_pass_instance_number = 1;
pass.pos_op = PASS_POS_INSERT_AFTER;

register_callback("myplugin",PLUGIN_PASS_MANAGER_SETUP,NULL, &pass);

/* Tell GCC some information about us, just for use in
   --help and --version.  */
register_callback("myplugin", PLUGIN_INFO, NULL, &myplugin_info);
```

The definition of each event is located in the file `plugin.def` of the GCC plugin directory. In file `tree-pass.h` we can find the name of each compiler pass, and the three positions where we can add our plugin pass:

- `PASS_POS_INSERT_AFTER`: The plugin pass is inserted after the reference pass.
- `PASS_POS_INSERT_BEFORE`: The plugin pass is inserted before the reference pass.
- `PASS_POS_REPLACE`: The plugin pass replaces the reference pass.

- **Pass descriptor**: Each pass of the compiler is described using a structure, whose fields are described in `tree-pass.h`. We do not have to specify a value for each field, and thus, a simple pass descriptor could be the following:

```
static struct gimple_opt_pass myplugin_pass =
{
  .pass.type = GIMPLE_PASS,      /* Type of pass */
  .pass.name = "myplugin",       /* Plugin name */
  .pass.gate = myplugin_gate,    /* Always retuns true */
  .pass.execute = myplugin_exec, /* Pass handler/callback */

};
```

However, if we want to set the pass properties, TODO flags, or other fields, the following is the best and more compact way to specify them:

```
static struct gimple_opt_pass myplugin_pass =
{
  {
  GIMPLE_PASS,
  "myplugin",                           /* name */
  myplugin_gate,                        /* gate */
  myplugin_exec,                        /* execute */
  NULL,                                 /* sub */
  NULL,                                 /* next */
  0,                                    /* static_pass_number */
  0,                                    /* tv_id */
  PROP_cfg | PROP_ssa,                  /* properties_required */
  0,                                    /* properties_provided */
  0,                                    /* properties_destroyed */
  0,                                    /* todo_flags_start */
  TODO_dump_func
    | TODO_verify_ssa
    | TODO_update_ssa
    | TODO_verify_stmts                 /* todo_flags_finish */
  }
};
```

- **Gate function**: This is the function that is executed before the pass implemented by the plugin. It could be used to enable/disable the plugin. If the return value is `true`, the "execute function" is executed; otherwise, its execution is skipped.

```
static bool myplugin_gate(void)
{
  return true;
}
```

- **Execute function**: This is the function called by the plugin and implements the plugin pass. The return value could contain `TODOs` to execute in addition to those in `TODO_flags_finish`. Usually, the return value is '0'.

```
static unsigned myplugin_exec(void)
{
  (...your plugin...)
  return 0;
}
```

A complete view of the plugin structure can be found in Subsection B.2.3.

B.2.1 Compiling and executing plugins

Once a plugin has been written, the next step is its compilation. The GCC project itself makes some indications about how a plugin has to be compiled [5]. If the plugin only has a single source file, it may be built with the following line:

```
$ gcc -I'gcc -print-file-name=plugin'/include -fPIC -shared -O2 plugin.c -o
    plugin.so
```

If the plugin source code involves more than one file, we can use a GNU Makefile script to compile them. The following GNU Makefile script shows how to build a simple plugin:

```
1 GCC=gcc
2 PLUGIN_SOURCE_FILES= plugin1.c plugin2.c
3 PLUGIN_OBJECT_FILES= $(patsubst %.c,%.o,$(PLUGIN_SOURCE_FILES))
4 GCCPLUGINS_DIR:= $(shell $(GCC) -print-file-name=plugin)
5 CFLAGS+= -I$(GCCPLUGINS_DIR)/include -fPIC -O2
6
7 plugin.so: $(PLUGIN_OBJECT_FILES)
8         $(GCC) -shared $^ -o $@
```

In order to execute a plugin, it is necessary that the version of GCC be 4.5 or superior, and that GCC have been compiled with the plugin support enabled. To check if our version of GCC has the support for plugins, we need to execute GCC with the following option:

```
$ gcc -print-file-name=plugin
```

If the operative version of GCC has the plugin support, this command returns the path to the directory where all the necessary header files to execute a plugin are located. Otherwise it returns the word "plugin".

Plugins are loaded by the compiler and invoked at pre-determined locations during the compilation process, e.g. between other compiler passes. To load the plugin into GCC, we need to run GCC with the option `-fplugin=/path/to/plugin/name.so`, or directly specify the name of the plugin with a shorter option `-fplugin=name`, if the plugin is located in the plugin directory.

The option `-fplugin-arg-name-key[=value]` enables to pass arguments to the plugin, where `name` is the name of the plugin, `key` is the name of a particular argument, and `value` is the value for this argument.

We can also load several plugins in sequence, for example, to add different new passes to the compiler.

[5] http://gcc.gnu.org/onlinedocs/gccint/Plugins.html

B.2.2 Plugin events

A plugin is activated by the compiler at specific events. The list of all considered events is located in file `plugin.def` of the GCC plugin directory. This list are then included by the file `gcc-plugin.h`. As we have seen before, these events allow us to activate our plugin at a particular moment in the compiler process. The way to register our plugin to a specific event is using the function `register_callback()`. One of the arguments of this function is the event in which our plugin will be triggered:

- `PLUGIN_PASS_MANAGER_SETUP`: It is the most common event. It enables to hook the pass implemented by the plugin into the pass manager. To that purpose, first we need to define the pass.
- `PLUGIN_INFO`: This is the second most common event. It enables to define the information about the plugin, which is printed with the `--help` and `--version` options.
- `PLUGIN_FINISH_TYPE`: To hook the pass after finishing parsing a type.
- `PLUGIN_FINISH_DECL`: To hook the pass after finishing parsing a declaration.
- `PLUGIN_START_UNIT`: The pass is hooked before processing a translation unit.
- `PLUGIN_FINISH_UNIT`: The pass is called after processing a translation unit.
- `PLUGIN_PRE_GENERICIZE`: It enables to see the low level AST in C and C++ frontends.
- `PLUGIN_FINISH`: The pass is called before GCC exits.
- `PLUGIN_NEW_PASS`: The pass implemented by the plugin is called when a pass is first instantiated.
- `PLUGIN_ATTRIBUTES`: The pass is called during attribute registration.
- `PLUGIN_PRAGMAS`: The pass is called during pragma registration.
- `PLUGIN_OVERRIDE_GATE`: It enables to override the pass gate decision for `current_pass`, which is the structure (defined in `tree-pass.h`) that stores the current optimization pass.
- `PLUGIN_PASS_EXECUTION`: The implemented pass is called before executing a particular pass.

- `PLUGIN_ALL_PASSES_START`: The pass is called before the first pass from `all_passes`, which is the structure (defined in `tree-pass.h`) that keeps the list of all passes (hooked to the list using the function `register_pass` in `passes.c`).
- `PLUGIN_ALL_PASSES_END`: The pass is called after the last pass from `all_passes`.
- `PLUGIN_ALL_IPA_PASSES_START`: The pass is called before the first Interprocedural Analysis (IPA) pass. In a similar way to `all_passes`, IPA passes are listed in `all_small_ipa_passes` and `all_regular_ipa_passes`.
- `PLUGIN_ALL_IPA_PASSES_END`: The pass is called after the last IPA pass.
- `PLUGIN_EARLY_GIMPLE_PASSES_START`: The pass is called before executing subpasses of a GIMPLE_PASS in `execute_ipa_pass_list()`, which is called by the Link Time Optimization (LTO) process [6].
- `PLUGIN_EARLY_GIMPLE_PASSES_END`: The pass is called after executing subpasses of a GIMPLE_PASS in `execute_ipa_pass_list()`.
- `PLUGIN_GGC_START`: The pass implemented by the plugin is called at the start of the GCC Garbage Collection (GGC)[7].
- `PLUGIN_GGC_MARKING`: It enables to extend the GGC marking.
- `PLUGIN_GGC_END`: The pass is called at the end of GGC.
- `PLUGIN_REGISTER_GGC_ROOTS`: It enables to register an extra GGC root table.
- `PLUGIN_REGISTER_GGC_CACHES`: It enables to register an extra GGC cache table.
- `PLUGIN_EVENT_FIRST_DYNAMIC`: This is a dummy event used for indexing callback array.

B.2.3 Plugin example

Below it is shown a source code of a complete operational example of a plugin:

```
1  /* Information about plugins and header that can be used
2  * could be find in the directory 'gcc -print-file-name=plugin' */
3
```

[6] See `http://gcc.gnu.org/onlinedocs/gccint/LTO.html` for more details.
[7] See `http://gcc.gnu.org/onlinedocs/gccint/Type-Information.html` for more information.

```
4  #include <gcc-plugin.h>
5  #include <coretypes.h>
6  #include <gimple.h>
7  #include <tree.h>
8  #include <tree-flow.h>
9  #include <tree-pass.h>
10 #include <stdio.h>
11
12 int plugin_is_GPL_compatible=1;
13
14 /* Information about the plugin */
15 static struct plugin_info myplugin_info =
16 {
17   .version = "001",
18   .help = "Work in progress",
19 };
20
21 /* This plugin works for GCC version 4.6.2 */
22 static struct plugin_gcc_version myplugin_ver =
23 {
24   .basever = "4.6.2",
25 };
26
27 /* The gate is a callback tripped just before exec is executed
28 * If the gate returns true, exec is executed,
29 * otherwise it is skipped. */
30 static bool myplugin_gate(void)
31 {
32   return true;
33 }
34
35
36 /* The callback that we have registered in the pass definition
37 * It is tripped when a compiler event is met */
38 static unsigned
39 myplugin_exec(void)
40 {
41
42   basic_block bb;
43   gimple stmt;
44   gimple_stmt_iterator gsi; /* GIMPLE statement iterator */
45
46   /* FOR_EACH_BB operates on a global variable in GCC which
47   * represents the current function being processed, cfun */
48   FOR_EACH_BB(bb)
49
50     /* We traverse each basic block visiting
51        each statement in the function */
52     for (gsi = gsi_start_bb(bb); !gsi_end_p(gsi); gsi_next(&gsi))
53     {
54       /* Get the statement */
55       stmt = gsi_stmt(gsi);
56       /* Do something with the stmt */
57       // ...your code...
58     }
59
```

```
60      printf("Plugin finished...\n");
61      return 0;
62    }
63
64    // Pass descriptor
65    static struct gimple_opt_pass myplugin_pass =
66    {
67      {
68        GIMPLE_PASS,
69        "myplugin",                           /* name */
70        myplugin_gate,                        /* gate */
71        myplugin_exec,                        /* execute */
72        NULL,                                 /* sub */
73        NULL,                                 /* next */
74        0,                                    /* static_pass_number */
75        0,                                    /* tv_id */
76        PROP_cfg | PROP_ssa,                  /* properties_required */
77        0,                                    /* properties_provided */
78        0,                                    /* properties_destroyed */
79        0,                                    /* todo_flags_start */
80        TODO_dump_func
81          | TODO_verify_ssa
82          | TODO_update_ssa
83          | TODO_verify_stmts                 /* todo_flags_finish */
84      }
85    };
86
87
88    /* Return 0 on success or error code on failure */
89    int
90    plugin_init(struct plugin_name_args *info, /* Argument information */
91                struct plugin_gcc_version *ver) /* Version info of GCC */
92    {
93      /* Used to tell the plugin-framework about where we want to be called in
94      * the set of all passes. This is located in tree-pass.h.  */
95      struct register_pass_info pass;
96      printf("Plugin initialized...\n");
97
98      /* Validate the plugin with the correct version of GCC */
99      if (strncmp(ver->basever, myplugin_ver.basever, strlen("4.6.2")))
100       return -1; /* Incorrect version of GCC */
101
102     /* Setup the info to register with GCC telling when we want to be
103     * called and to what GCC should call, when it's time to be called. */
104     pass.pass = &myplugin_pass.pass;
105
106     /* Get called after GCC has produced the SSA representation of the
107     * program. In this case, after the first SSA pass.  */
108     pass.reference_pass_name = "ssa";
109     pass.ref_pass_instance_number = 1;
110     pass.pos_op = PASS_POS_INSERT_AFTER;
111
112     /* Tell GCC that we want to be called after the first SSA pass */
113     register_callback("myplugin", PLUGIN_PASS_MANAGER_SETUP, NULL, &pass);
114
115     /* Tell GCC some information about us, just for use in --help
```

```
116             and ——version */
117     register_callback("myplugin", PLUGIN_INFO, NULL, &myplugin_info);
118
119     /* Successful initialization */
120     return 0;
121
122   }
```

B.3 GENERIC and GIMPLE representations

GCC handles several internal representations. In this section, we describe two of these representations: GENERIC and GIMPLE. The reason to describe these representations is because they are involved in the plugin development. This section describes the main details of these representations and the basic knowledge to address the plugin construction.

B.3.1 GENERIC

GENERIC is the common representation, language-independent, shared by all front ends. Each front end parses its corresponding source language, and emits the GENERIC representation at the end of the process. This representation is very similar to the Abstract Syntax Trees (ASTs), but without language specific constructs. Figure B.2 shows a graphical representation of a GENERIC tree. All the tree nodes are defined in `tree.def`.

Each parser can generate its own AST, but at the end of the process, it has to emit a GENERIC representation of the source code. Before GENERIC, parsers built

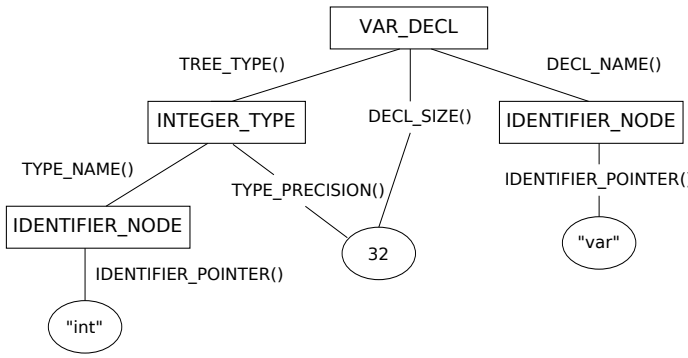

Figure B.2: Graphical GENERIC/TREE representation of `int var;`

up trees for a single statement, and then the compiler lowered these trees, which were abstract syntax trees, to RTL before moving on to the next statement. RTL fits well for low-level optimizations, but it has a lot of limitations for higher level optimizations. For example, RTL works with data types which are limited to machine words, and therefore, it is not possible to deal with structures or arrays as a single construct. Another limitation is the lack of trees for entire functions, requirement needed for most of the high-level optimizations. For this reason, among others, it was proposed to create new intermediate representations which allowed to handle this higher-level constructs, such as arrays or functions, enabling and making new optimizations easier, as inlining. From this purpose [121], GENERIC and GIMPLE arose in the version 4.0.0 of GCC, released in April 2005.

GENERIC tree nodes

A GENERIC representation uses `tree` as the data structure. A `tree` is defined in GCC as a pointer to a big union of structures –called `tree_node`–, and thus, tree nodes are dynamically typed. This set of structures can be found in `tree.h`, where the node types common to all languages are defined. Each language has some particular tree nodes. In the case of C and C++, these additional tree nodes are defined in `c-family/c-common.def`.

All variables and structure fields pointing to tree nodes have the type `tree`, which can address any type of node, even with different internal representations. GCC provides some functions and macros to access the data allocated in a particular tree node. These functions and macros are defined in `tree.h`, and can be found in the GCC internals.

B.3.2 GIMPLE

GIMPLE is a simplified version, a subset of GENERIC. GIMPLE is a three-address language with no high-level control flow structures, in which each statement does not contain more than three operands (except function calls or conditional expressions), control flow structures are combinations of conditional statements and goto operators, and there is no lexical scope. This restricted grammar facilitates the job to the different optimization passes. Figure 4.4 shows some of the differences between GENERIC and GIMPLE. Each GIMPLE statement is represented as a tuple, which contains the type of the statement, the result, the operator, and the operands. The result and the operands are still represented using trees, with the same syntax used in the GENERIC representation. Figure B.3 shows the internal raw representation of GIMPLE using tuples. This view can be obtained with the compilation option `-fdump-tree-all-raw`, while the normal view –like a source code– of GIMPLE is obtained with the option

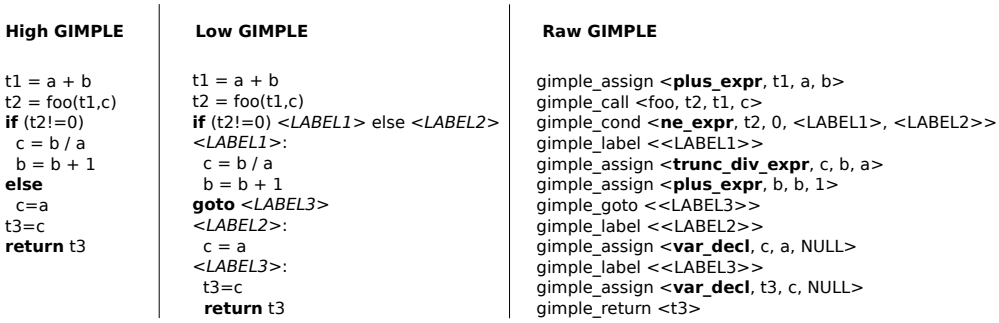

Figure B.3: Internal raw form of GIMPLE, with the 3-operands tuples

`-fdump-tree-all`.

GIMPLE lowers the control flow, where the program is transformed into a sequence of statements and conditional and unconditional jumps. This lowered control flow eases the transformation of GIMPLE into a SSA form. There exist two GIMPLE levels: *High GIMPLE*, generated in a first phase where cleanups and simplifications are performed, and *Low GIMPLE*, where the control flow are lowered. Differences between these two levels can be found in Figure 4.4.

Further information on how are named each GIMPLE node, and how to manipulating them, is found in the GCC Internals [71].

Traverse the GIMPLE representation

There are several ways to traverse the GIMPLE representation. In Sect. B.2.3 we have seen one of them, which implies traversing each `basic_block` that represents the current function being processed. This function is also accessible using the global variable `current_function_decl`. The following code shows how to traverse the GIMPLE representation:

```
1  static unsigned myplugin_exec(void)
2  {
3    /* Get the body of the current function declaration.  */
4    gimple_seq seq = gimple_body (current_function_decl);
5
6    /* Traverse the body of the current function.  */
7    walk_gimple_seq (seq, scan_stmt, scan_op, NULL);
8  }
9
10 /* This function is used to traverse each statement in the sequence.  */
11 static tree
12 scan_stmt( gimple_stmt_iterator *ptr_gsi,
13            bool *handled_ops_p,
```

```
14              struct walk_stmt_info *wi)
15  {
16    /* Get the current statement.  */
17    gimple stmt = gsi_stmt (*ptr_gsi);
18
19    /* Get the code of the statement to process it properly.  */
20    switch (gimple_code (stmt))
21    {
22      case GIMPLE_OMP_PARALLEL:
23      case GIMPLE_CALL:
24    ....
25    }
26    return NULL_TREE;
27
28  }
29
30  /* This function can be used to traverse all the operands
31      of each statement.  */
32  static tree
33  scan_op(tree *tp, int *walk_subtrees, void *data)
34  {
35    return NULL_TREE;
36  }
```

The function `walk_gimple_seq` enables us to traverse the sequence of statements of the current function. It has two parameters that are the names of two functions. The first one is used to access to each statement, whereas the second one is used to access, if necessary, the operands of each statement. Before processing each statement or operand, it is highly advisable to do a `switch` in order to process them according to its type and avoid errors.

Appendix C

Installation and User's Manuals

This appendix describes the installation process of BFCA and ATLaS, including all the instructions necessary to install the required software for these both systems, and also the user's manual.

C.1 BFCA: BonaFide C Analyzer

BFCA was proposed in Chap. 5. Although BFCA consists of four subsystems that could be used independently, it is specially designed to be used as a single system.

C.1.1 System Requirements

Bona Fide C Analyzer requires to have the following software and packages installed on your computer:

- JAVA 2 SDK, SE 1.6.x (or later).
- ANTLRv2.
- C development environment (C preprocessor, C standard library, C compiler. GCC is recommended).
- Intel C Compiler version 12 (installed in a compatible Linux system).

C.1.2 Installation Guide

Once you have installed each software specified below, you have to follow the next steps to install BFCA in your computer:

1. Unpack the tarball. This will create a new directory, named `bfca-1.0`, with the sources of Bona Fide C Analyzer.

```
$ tar xzfv bfca-1.0.tar.gz
```

2. In the `bfca-1.0` directory, there is an script named `pathvars.sh`, which modify the PATH and CLASSPATH environment variables. You have to modify the following line of the script in order to set the path of the directory in which you unpack the tarball.

```
INSTALL_DIR=$HOME/bfca-1.0
```

3. Set the environment variables for a terminal window adding the following line to `.bashrc` file:

```
source INSTALL_DIR/pathvars.sh
```

4. Checks `INSTALL_DIR/src/XMLCetus/build.sh` to verify if the following two lines are correct for your system.

```
JAVABIN="/usr/bin"                 #JAVA location
ANTLR="/usr/share/java/antlr.jar"  #antlr location
```

5. Install Bona Fide C Analyzer using `install.sh` script.

```
$ ./install.sh bin
```

6. Bona Fide C Analyzer contains a set of source codes (`samples` directory) that you can use to test the correct operation of the software. You can test it using the `test` option from the installation script, which also checks the correct installation of the software.

```
$ ./install.sh test
```

7. Once you have installed Bona Fide C Analyzer, you are ready to use it:

```
$ bfca --help
```

C.1.3 Command Options

Bona Fide C Analyzer has several options. Mandatory arguments to long options are mandatory for short options too.

−c, −−compile "FILE1.C FILE2.C ..." : specifies the list of source files to be analyzed (and compile if -m | −−makefile option is not specified).

Note: −c and −l options cannot be activated at the same time.

−l, −−list FILE : specifies a file which contains a list with the source files to be analyzed. It is not necessary to point out all the source files, but the correct operation of XMLCetus is not guaranteed, because of the original software of Cetus may not be able to build the Intermediate Representation of the code. There could be dependencies that force to indicate all the source files.

Note: $-c$ and −l options cannot be activated at the same time.

−f, −−flag "FLAG1 FLAG2 ..." : specifies the flags to compile the code.

Note: This option will be ignored with −m,−−makefile option".

−i, −−input "PARAM1 PARAM2 ..." : specifies the input parameters to execute the user's program.

−o, −−output : specifies the file where analysis results will be written.

−m, −−makefile : specifies that the compilation of the user's program depends on a Makefile. With this option it is necessary to point out the name of the executable created by the Makefile, using the −e, −−exec option.

Note: If the makefile operation depends on an argument, you can pass it following this option.

−e, −−exec RUNFILE : specifies the name of the executable resulting of the compilation of the user's program.

−x, −−xml "FILE.XML" : specifies the XML file that represents the source code that contains the target loop.

Note: It is mandatory to use this option together with -p, –parallelize

−p, −−parallelize "LOOP_LINENUMBER" : specifies the line number of the loop to be augmented with OpenMP + speculative clauses

Note: It is mandatory to use this option together with -x, –xml

`-h, --help` : display this help and exit.

`-v, --version` : output version information and exit.

C.1.4 Running Example

Bona Fide C Analyzer can be executed with two purposes: obtaining a characterization of the source code, and augmenting the source with OpenMP constructs.

Characterizing the source code

If the software that you want to analyze is compiled by using a Makefile, you have two alternatives:

1. To create a file with the list of source files that you want to analyze, and then pass this file name to Bona Fide C Analyzer.

   ```
   $ bfca -m -l list -e exec -o report.out -i "input_file1 input_file2"
   ```

   This command (1) compiles the source code, (2) generates an executable named `exec.exe`, (3) executes it with the specified input files, and finally (4) generates a report with the use of the variable in a file named `report.out`.

2. To indicate them by using the --compile option:

   ```
   $ bfca -m -c "source1.c source2.c source3.c" -e exec -o report.out
         -i "input_file1 input_file2"
   ```

On the other hand, if you do not have a Makefile, you can use Bona Fide C Analyzer with the -c and -f options:

```
$ bfca -m -c "source1.c source2.c" -f "Flag1 Flag2" -e exec -o report.out
      -i "input_file1 input_file2"
```

Augmenting the source code

Once a source code has been characterized, and BFCA has generated an XML version, you can execute the software with the following options to instrument a certain loop with OpenMP constructs.

```
$ bfca -x "source.xml" -p 66
```

C.2 ATLaS

The compilation-module of ATLaS was described in Chap. 4, in which we propose a GCC plugin to automatically apply all the transformations needed for speculatively parallelize a source code. In this section, we describe the installation process of the whole system ATLaS, including compilation-module and the TLS runtime library, as well as a running example for final users.

C.2.1 Content of the package

The package contains the following directories:

- `atlas`: This is the script that allows us to run ATLAS. The different options to execute this script are described in Sect. C.2.4.
- `doc/`: This directory contains the `PDF` version of this document.
- `gcc_updates/`: This directory contains the source files that modify the compiler GCC to support the new OpenMP `speculative` clause.
- `specprag/`: This directory contains the core of ATLAS, all the source code that implement the compiler and runtime modules of ATLAS.

C.2.2 System Requirements

ATLaS only requires the GCC compiler, preferably version 4.6.2, to be executed. The rest of packages required are involved with the compilation process of GCC and they will be described in the next section. Therefore, you need to install GCC version 4.6.2, with the plugin support enabled. To check this, you need to execute the following command and see the corresponding output.

```
$ <gcc_install_dir/>gcc -print-file-name=plugin
<gcc_install_dir/>/lib/gcc/<architecture_and_so>/4.6.2/plugin
```

This path indicates the directory where header files needed to execute plugins are located. If the plugin support is not enabled, you only will receive the word `plugin` in the output.

The compiler module of our system was developed using version 4.6.2 of GCC. Any version superior to 4.5, in which plugins were introduced, should work. However, we only ensure that the compiler module work with version 4.6.2.

C.2.3 Installation Guide

The installation process should be done in two steps. First, you have to download, compile and install GCC. Second, you need to replace original GCC files with source files that we provide you in the ATLaS package.

Install and compile GCC 4.6.2

First, you have to download version 4.6.2 of GCC from one of the official mirrors. For example, from the next URL:

```
ftp://www.mirrorservice.org/sites/ftp.gnu.org/gnu/gcc/gcc-4.6.2/gcc-4.6.2.tar.bz2
```

GCC requires that various tools and packages be available for use in the build procedure. The following packages are required:

- Another GCC compiler. This is necessary to compile the required version of GCC.
- GNAT (gnat-4.4)
- GMP (libgmp3-dev)
- MPFR (libmpfr-dev)
- MPC (libmpc-dev)
- GNU binutils
- libc6-dev (If you have a 64-bit system, install the 32-bit version of the library: libc6-dev-i386).
- libtool
- GAWK

A more detailed list of the prerequisites can be found in:

`http://gcc.gnu.org/install/prerequisites.html`

Once you have installed all the packages listed above, you need to prepare the building directory.

1. Uncompress the file downloaded into a directory called `srcdir`.

```
$ mkdir <gcc_build_dir>
$ mkdir <gcc_build_dir>/srcdir
$ mv gcc-4.6.2.tar.bz2 <gcc_build_dir>/srcdir
$ cd <gcc_build_dir>/srcdir
$ tar -xvjz gcc-4.6.2.tar.bz2
```

2. At the same level than `srcdir`, create a new directory called `objdir`.

```
$ mkdir objdir
```

3. Configure the compilation process of GCC with the following command. The `-prefix` option indicates where this version will be installed.

```
$ cd objdir
$ ../srcdir/configure --enable-shared --enable-threads=posix
  --enable-__cxa_atexit --enable-clocale=gnu
  --enable-languages=c --prefix=/opt/gcc-4.6.2
```

4. Starts the compilation. This process could take several hours.

```
$ make bootstrap
```

5. Install the compiler in the directory indicated in the configuration command.

```
$ make install
```

6. Once this process has finished, you have GCC version 4.6.2 installed in your computer. The next step is add to this version the support for the OpenMP `speculative` clause.

More compilation options and a detailed documentation of this process can be found in:

`http://gcc.gnu.org/install/configure.html`

Add support for the OpenMP `speculative` clause into GCC

Original implementation of GCC does not support Thread-Level Speculation (TLS). We have designed a new OpenMP clause, called `speculative`, to support TLS into GCC. For this purpose, we need to modify the following files of GCC:

```
$ cd gcc_updates
$ ls *
c-parser.c gimplify.c tree.c tree-nested.c
c-typeck.c omp-low.c tree.h tree-pretty-print.c

c-family:
c-omp.c c-pragma.h
```

Modifying GCC to add this new clause can be done in two simple steps:

1. Copy the files in directory `gcc_updates` into the directory that contains the original source files of GCC.

```
$ mv gcc_updates/* <gcc_build_dir>/srcdir/gcc
```

2. Recompile GCC, executing the following command in the directory `objdir`.

```
$ cd <gcc_build_dir>/objdir
$ make
```

At this point, you have installed in your computer a modified version of GCC that support the new OpenMP clause.

Using ATLaS with your software

Once you have installed our modified version of GCC, you need to accomplish a few more steps to use ATLaS with your application. You have to follow the next steps to install ATLaS (compiler and runtime modules):

1. Copy `atlas` script into the directory that contains your source code. This script drivers the compilation of your program with the TLS library.

```
$ cp atlas <your_application_dir>
```

2. Create a symbolic link to the directory `specprag` of the installation directory, using the same name than the original. This directory contains the compile and runtime modules of ATLaS.

```
$ ln -s <path_to_specprag> specprag
```

In this point, your are ready to start using ATLaS.

C.2.4 Command Options

ATLaS has several options. Mandatory arguments to long options are mandatory for short options too.

`-t, --threads INTEGER` : specifies the number of threads to execute the resulting binary file.

`-b, --block INTEGER` : specifies the size of each block of iterations.

`-p, --maxpointer INTEGER` : specifies the maximum number of elements which are speculative.

`-i, --maxiter INTEGER` : specifies the maximum number of iterations that a speculative loop can execute.

`-m --mask INTEGER` : specifies the size of the mask used.

`-c, --compile "FILE1.C FILE2.C ..."` : specifies the list of source files to be analyzed.

`-f, --flag "FLAG1 FLAG2 ..."` : specifies the flags to compile the code.

`-e, --exec RUNFILE` : specifies the name of the executable resulting of the compilation of the user's program.

`-d, --dump` : enables the dumping at various stages of processing the intermediate language tree to a file.

`-h, --help` : display this help and exit.

`-v, --version` : output version information and exit.

Six of this options are mandatory: `--threads`, `--block`, `--maxpointer`, `--maxiter`, `--mask`, and `--compile`. If you do not specify any of these options, ATLaS generates the following output:

```
1 You must specify the number of threads with '-t' or '--threads'
2 You must specify the size of the block of iterations with '-b' or '--block'
3 You must specify the maximum number of speculative elements with '-p' or
      '--maxpointer'
4 You must specify the maximum number of iterations with '-i' or '--maxiter'
5 You must specify the size of the mask with '-m' or '--mask'
6 You must specify a C file with '-c' or '--compile'
7
8 Usage: atlas -t "threads" -b "block" -p "maxpointer" -i "maxiter" -m "mask"
       -c "file1.c"
9 Example: atlas -t 4 -b 50 -p 10 -i 10000 -m 127 -c "example.c"
```

A template for a correct execution of ATLaS could be the following:

```
$ atlas --threads T --block B --maxpointer P --maxiter I --mask M -c
    example.c
```

where I is the maximum number of iterations that a speculative loop can execute in the pro- gram, T is the number of threads we want to run the program with, B is the size of the block of iterations, P is the maximum number of elements which are speculative, and M is the size of the mask used. These parameters are set by the programmer and they are not very tricky to set up, because they only need to know some easy features of the target loop to set maxiter and maxpointer. For example, a loop that speculatively reads from and write into an array of 1000 elements, and has 200 iterations sets the value of P to 1000, and I to 200. The other three parameters, the number of threads, the block size, and the mask size are variable and programmers can experiment with different values until obtaining the best performance to their programs.

C.2.5 Running Example

The following steps resume the process of parallelizing an application with ATLaS.

1. Analyze the loop to be parallelized, classifying their variables into private, shared or speculative. Any variable that could lead to a dependency violation should be classified as speculative.

2. Add the OpenMP directive `omp parallel for`, with the corresponding clauses `private`, `shared`, and the new `speculative` clause.

3. Add the function `specbegin(N)` before the parallel loop to specify the number of iteration of the loop. This function initializes the structures needed for the runtime module of ATLaS. In following versions of ATLaS, the addition of this function will not be further necessary.

```
1   #define MAX 100
2   #define NITER 30000
3   int i, k, aux, var1, var2;
4   int array[MAX];
5
6   specbegin (NITER);
7
8   #pragma omp parallel for default(none) schedule(static) \
9     private(i, k, aux) \
10    shared(array) \
11    speculative(var1, var2)
12  for ( i = 0 ; i < NITER ; i++ ) {
```

```
13
14      if (i == 3000) { k = var2; }
15      if (i == 600) { k = var1; }
16
17      for (k = 0; k < array[i % MAX] + NITER; k++) {
18
19        if (k >= 29900) { var2 = k + array[(i + k) % MAX]; }
20        if (k <= 200) { var1 = array[i % MAX]; }
21        aux = (k + NITER) % 100000;
22      }
23
24      if (i == NITER-1) { var2 = var1; }
25    }
```

4. Execute ATLaS with the right values for each argument. Details for each argument are found in the previous section.

```
$ ./atlas --threads 4 --block 500 --maxpointer 2 --maxiter 30000
          --mask 7 -c synthetic_sequential.c
```

ATLaS produces the following output for the code example shown above:

```
1  ****************************************
2  Analyzing 'main()'...
3  ****************************************
4
5  OpenMP pragma omp PARALLEL detected in line 69!
6    Another kind of clause
7    Private Clause: 'i'
8    Private Clause: 'k'
9    Private Clause: 'aux'
10   Shared clause: 'array'
11   Speculative clause: 'j'
12   Speculative clause: 'l'
13
14 Searching FOR directive associated with the PARALLEL directive...
15 Number of speculative variables = 2
16    Variable : j. Type: int.
17    Variable : l. Type: int.
18 Replacing original FOR loop for a speculative version...
19
20 Reading from speculative variable:  k = l;
21 Reading from speculative variable:  k = j;
22 Writing into speculative variable:  l = D.3239 + k;
23 Writing into speculative variable:  j = array[D.3242];
24 Reading from speculative variable:  l = j;
25 Writing into speculative variable:  l = D.3324;
26 Adding engine's functions pre-loop
27
28 Plugin finished in function 'main()'.
```

5. ATLaS generates a binary file functionally equivalent to the original application, but its execution will run in parallel, using the number of threads specified in the compilation.

   The execution of this file prints in first place a resume of the values selected for each option of ATLaS.

```
$ ./synthetic_sequential-auto
************************************************************
Speculative code executed with the following parameters:
Threads = 4
Maximum number of iterations = 30000
Block size = 500
Mask size = 7
Initial capacity reserved to pointers in a matrix = 2
************************************************************
...(application output)
```

Appendix D

Digital Support Contents

This appendix describes the contents of the digital support attached to this Ph.D. thesis. This is structured in several folders:

- **ATLaS:** It contains the source code of the ATLaS system, including the compile-time system developed in this Ph.D. thesis, and the TLS runtime library used.
- **auxiliaryTools:** It contains all the auxiliary tools and programs developed to reach the goals of this Ph.D. thesis.
- **BFCA:** It contains the source code of BFCA+.
- **dissertation:** It contains a PDF copy of this document.
- **papers:** It contains PDF versions of all the papers and contributions that this Ph.D. has produced.
- **tests-experiments:** It contains all the regression tests and the experiments run during the Ph.D. thesis.

Bibliography

[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] William R. Adrion. Research methodology in software engineering: Summary of the Dagstuhl workshop on future directions in software engineering. *SIGSOFT Software Engineering Notes*, 18(1):36–37, January 1993.

[3] Ademar Aguiar, Gabriel David, and Greg Badros. JavaML 2.0: Enriching the markup language for Java source code. In *Proceedings of XML Aplicações e Tecnologias Associadas*, XATA'04, pages 1–12, 2004.

[4] Raihan Al-Ekram and Kostas Kontogiannis. An XML-Based framework for language neutral program representation and generic analysis. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, CSMR'05, pages 42–51, 2005.

[5] Sergio Aldea. *Traducción automática de código secuencial a código especulativamente paralelo*. M.Sc. Computer Engineer, final project. Universidad de Valladolid, Departamento de Informática, June 2010.

[6] Sergio Aldea, Diego R. Llanos, and Arturo Gonzalez-Escribano. Using SPEC CPU2006 to evaluate the sequential and parallel code generated by commercial and open-source compilers. *The Journal of Supercomputing*, 59:486–498, June 2012.

[7] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 1st edition, October 2001.

[8] Mehdi Amini, Corinne Ancourt, Fabien Coelho, François Irigoin, Pierre Jouvelot, Ronan Keryell, Pierre Villalon, Béatrice Creusillet, and Serge Guelton. PIPS is not (just) polyhedral software. In *International Workshop on Polyhedral Compilation Techniques*, IMPACT'11, Chamonix, France, 2011.

[9] Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice Onanian McMahon, François-Xavier Pasquier, Grégoire Péan, and Pierre Villalon. Par4All: From convex array regions to heterogeneous computing. In *Second International Workshop on Polyhedral Compilation Techniques, within HiPEAC 2012*, IMPACT'12, 2012.

[10] Noritoshi Atsumi, Takashi Kobayashi, Shinichiro Yamamoto, and Kiyoshi Agusa. An XML C source code interchange format for CASE tools. In *Proceedings of the IEEE 35th Annual Computer Software and Applications Conference*, COMPSAC'11, pages 498–503, 2011.

[11] Greg J. Badros. JavaML: A markup language for java source code. In *Computer Networks: The International Journal of Computer and Telecommunications Networking*, pages 159–177, 2000.

[12] Hansang Bae, Leonardo Bachega, Chirag Dave, Sang-Ik Lee, Seyong Lee, Seung-Jai Mind, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A Source-to-Source compiler infrastructure for multicores. In *Proceedings of the 14th International Workshop on Compilers for Parallel Computing*, CPC'09.

[13] Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis, and Kunle Olukotun. The OpenTM transactional application programming interface. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT'07, pages 376–387, 2007.

[14] André Balsa. Linux benchmarking HOWTO. Last access: April 2014. `http://tldp.org/HOWTO/Benchmarking-HOWTO.html`, August 1997.

[15] Utpal K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, 1988.

[16] Joshua E. Barnes. TREE. Institute for Astronomy. University of Hawaii. Last access: April 2014. `http://www.ifa.hawaii.edu/~barnes/ftp/treecode/`, 1997.

[17] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, 2004.

[18] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path language (XPath) 2.0 (Second edition). W3C recommendation 14 december 2010. Last access: April 2014. `http://www.w3.org/TR/xpath20/`.

[19] Arthur J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions On Electronic Computers*, EC-15(5):757–763, 1966.

[20] Barna L. Bihari, Michael Wong, Amy Wang, Bronis R. de Supinski, and Wang Chen. A case for including transactions in OpenMP II: hardware transactional memory. In *Proceedings of the 8th International Workshop on OpenMP*, volume 7312 of *Lecture Notes in Computer Science, IWOMP'12*, pages 44–58, June 2012.

[21] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, 2008.

[22] Marat Boshernitsan and Susan L. Graham. Designing an XML-Based exchange format for harmonia. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, WCRE'00, pages 287–289, 2000.

[23] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 . W3C Recommendation 26 November 2008. Last access: April 2014. `http://www.w3.org/TR/xml/`.

[24] Robert C. Camp. *Benchmarking: The Search for Industry Best Practices that Lead to Superior Performance*. ASQC Quality Press, 1st edition, June 1989.

[25] Robert C. Camp. Benchmarking–a conversation with Robert C. Camp. The source. Interview by Joe Flower. *The Healthcare Forum Journal*, 36(1):30–36, February 1993.

[26] Robert C. Camp. Best practice benchmarking: the path to excellence. *CMA Magazine*, 72(8):10–15, 1998.

[27] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. HELIX: Automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 84–93, 2012.

[28] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[29] Cetus Project. Modified Artistic License. Last access: April 2014. `http://cetus.ecn.purdue.edu/Download/license_mod.html`.

[30] Cetus Project, Purdue University. Cetus, a Source-to-Source compiler infrastructure for C programs. Last access: April 2014. `http://cetus.ecn.purdue.edu`.

[31] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[32] Yin Chan, Ashok Sudarsanam, and Andrew Wolfe. The effect of compiler-flag tuning on SPEC benchmark performance. *SIGARCH Computer Architecture News*, 22(4):60–70, September 1994.

[33] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, 1 edition, October 2000.

[34] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA'05, pages 519–538, October 2005.

[35] Michael K. Chen and Kunle Olukotun. The Jrpm system for dynamically parallelizing java programs. In *Proceedings of the 30th International Symposium on Computer Architecture*, ISCA'03, pages 434–445, May 2003.

[36] Marcelo Cintra and Diego R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'03, pages 13–24, 2003.

[37] Marcelo Cintra and Diego R. Llanos. Design space exploration of a software speculative parallelization scheme. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):562–576, June 2005.

[38] Marcelo Cintra, José F. Martínez, and Josep Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA'00, pages 13–24, 2000.

[39] Kenneth L. Clarkson, Kurt Mehlhorn, and Raimund Seidel. Four results on randomized incremental constructions. *Computational Geometry*, 3(4):185–212, 1993.

[40] Albert Cohen, Ayal Zaks, Dorit Nuzman, Diego Novillo, Roberto Costa, Grigori Fursin, and Sebastian Pop. 2nd HIPEAC GCC Tutorial. Last access: April 2014. Slides in `http://www.hipeac.net/node/746`, January 2007.

[41] Michael. L. Collard. An infrastructure to support meta-differencing and refactoring of source code. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, ASE'03, pages 377–380, October 2003.

[42] Michael L. Collard. Addressing source code using srcML. In *IEEE International Workshop on Program Comprehension Working Session: Textual Views of Source Code to Support Comprehension*, IWPC'05, 2005.

[43] Michael L. Collard, Michael J. Decker, and Jonathan I. Maletic. Lightweight transformation and fact extraction with the srcML toolkit. In *Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM'11, pages 173–184, 2011.

[44] Michael L. Collard, Huzefa H. Kagdi, and Jonathan I. Maletic. An XML-Based lightweight C++ fact extractor. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC'03, pages 134–143, 2003.

[45] Michael L. Collard and Johnathan I. Maletic. Document-oriented source code transformation using XML. In *Proceedings of the First International Workshop on Software Evolution Transformation.*, SET'04, pages 11–14, 2004.

[46] Michael L. Collard, Jonathan I. Maletic, and Brian P. Robinson. A lightweight transformational approach to support large scale adaptive changes. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*, ICSM'10, pages 1–10, 2010.

[47] Peter Collingbourne and Paul Kelly. A compile-time infrastructure for GCC using Haskell. In *GCC Research Opportunities workshop, within the 4th International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC)*, GROW'09, 2009.

[48] James R. Cordy. Generalized selective XML markup of source code using agile parsing. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC'03, pages 144–153, 2003.

[49] Standard Performance Evaluation Corporation. SPEC CPU2000 to be retired in February 2007. Last access: April 2014. `http://www.spec.org/cpu2000/retired.html`.

[50] Standard Performance Evaluation Corporation. SPEC releases CPU2000 v1.3. Last access: April 2014. `http://www.spec.org/cpu2000/press/release_v1.3.html`.

[51] Standard Performance Evaluation Corporation. SPEC releases CPU2006 benchmarks. Last access: April 2014. `http://www.spec.org/cpu2006/press/V1.2release.html`.

[52] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, March 1998.

[53] Francis Dang, Hao Yu, and Lawrence Rauchwerger. The R-LRPD test: speculative parallelization of partially parallel loops. In *Proceedings of the 16th IEEE International Parallel and Distributed Processing Symposium*, IPDPS'02, pages 20–29, 2002.

[54] 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.

[55] Luc Devroye, Ernst P. Mücke, and Binhai Zhu. A note on point location in Delaunay triangulations of random points. *Algorithmica*, 22:477–482, 1998.

[56] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. Software behavior oriented parallelization. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'07, pages 223–234, June 2007.

[57] Jialin Dou and Marcelo Cintra. A compiler cost model for speculative parallelization. *ACM Transactions on Architecture and Code Optimization, TACO*, 4(2), June 2007.

[58] Zhao-Hui Du, Chu-Cheow Lim, Xiao-Feng Li, Chen Yang, Qingyu Zhao, and Tin-Fook Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI'04, pages 71–81, June 2004.

[59] Kamil Dudka, Petr Müller, Petr Peringer, and Tomáš Vojnar. Predator: A tool for verification of low-level list manipulation. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 627–629. 2013.

[60] Mark Duranton. The HiPEAC vision. Network of Excellence on High Performance and Embedded Architecture and Compilation (Grant Agreement ICT-217068), Deliverable 3.5. Last access: April 2014. `http://www.hipeac.net/system/files/hipeacvision.pdf`.

[61] Juergen Ebert, Bernt Kullbach, and Andreas Winter. GraX: An interchange format for reengineering tools. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, WCRE'99, pages 89–98, 1999.

[62] Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. New data structures to handle speculative parallelization at runtime. In *Proceedings of the 7th International Symposium on High-level Parallel Programming and Applications*, HLPP '14, 2014.

[63] Cesare Ferri, Andrea Marongiu, Benjamin Lipton, R. Iris Bahar, Tali Moreshet, Luca Benini, and Maurice Herlihy. SoC-TM: integrated HW/SW support for transactional memory programming on embedded MPSoCs. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS'11, pages 39–48, 2011.

[64] Gregor Fischer and Joachim Lusiardi. JAML - an XML-Representation of Java-Source code. Technical report, University of Würzburg, Würzburg, 2008.

[65] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher K. I. Williams, and Michael O'Boyle. Milepost GCC: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, 39(3):296–327, June 2011.

[66] Lin Gao, Lian Li, Jingling Xue, and Pen-Chung Yew. SEED: A statically greedy and dynamically adaptive approach for speculative loop execution. *IEEE Transactions on Computers*, 62(5):1004–1016, 2013.

[67] Saturnino Garcia, Donghwan Jeon, Christopher M. Louie, and Michael Bedford Taylor. The Kremlin oracle for sequential code parallelization. *IEEE Micro*, 32(4):42–53, 2012.

[68] Álvaro García-Yágüez, Diego Llanos, and Arturo Gonzalez-Escribano. Squashing alternatives for software-based speculative parallelization. *IEEE Transactions on Computers*, Early Access Online, 2013.

[69] Álvaro García-Yágüez, Diego R. Llanos, and Arturo Gonzalez-Escribano. Robust thread-level speculation. In *Proceedings of the 18th International Conference on High Performance Computing*, HIPC'11, pages 1–11, 2011.

[70] Taras Glek, David Mandelin, and Benjamin Smedberg. Dehydra and TreeHydra projects. Last access: April 2014. `https://developer.mozilla.org/en-US/`, 2010.

[71] GNU Project. GCC internals. Last access: April 2014. `http://gcc.gnu.org/`.

[72] GNU Project. GCC, the GNU Compiler Collection. Last access: April 2014. `http://gcc.gnu.org/`.

[73] GNU Project. Using the GNU Compiler Collection for GCC version 4.3.2. Last access: April 2014. `http://gcc.gnu.org/onlinedocs/gcc-4.3.2/gcc.pdf`.

[74] Katsuhiko Gondow and Hayato Kawashima. Towards ANSI C program slicing using XML. *Electronic Notes in Theoretical Computer Science*, 65(3):30–49, July 2002.

[75] William Gropp, Ewing L. Lusk, and Anthony Skjellum. *Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface*. The MIT Press, second edition edition, November 1999.

[76] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Grösslinger, and Louis-Noël Pouchet. Polly - Polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques*, IMPACT'11, 2011.

[77] Serge Guelton. *Building Source-to-Source Compilers for Heterogeneous Targets*. PhD thesis, Université européenne de Bretagne, Rennes, France, 2011.

[78] Manish Gupta and Rahul Nim. Techniques for speculative run-time parallelization of loops. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC'98, pages 1–12, 1998.

[79] Torben Hagerup. Allocating independent tasks to parallel processors: An experimental study. *Journal of Parallel and Distributed Computing*, 47(2):185–197, 1997.

[80] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'98, pages 58–69, October 1998.

[81] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[82] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, September 2006.

[83] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, ISCA'93, pages 289–300, 1993.

[84] Ben Hertzberg and Kunle Olukotun. Runtime automatic speculative parallelization. In *Proceedings of the Ninth IEEE/ACM International Symposium on Code Generation and Optimization*, CGO'11, pages 64–73, 2011.

[85] Richard C. Holt, Andreas Winter, and Andy Schürr. GXL: Toward a standard exchange format. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, WCRE'00, pages 162–171, 2000.

[86] IBM. Thread-level speculative execution for C/C++. IBM XL C/C++ for Blue Gene, 2012. Tech. report. Last access: April 2014. http://community.hartree.stfc.ac.uk/access/content/group/admin/HPC%20Training/BG_Q%20training%20course%20February%202013/Reference/Thread-Level%20Speculative%20Execution%20for%20C-C%2B%2B.pdf.

[87] 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.

[88] Intel Software Development Products. Quick reference guide to optimization with Intel compilers version 11. Last access: April 2014. https://umdrive.memphis.edu/g-hpc/public/resource_docs/Intel_Compiler_Quick_Reference_Guide.pdf.

[89] Nikolas Ioannou, Jeremy Singer, Salman Khan, Polychronis Xekalakis, Paraskevas Yiapanis, Adam Pocock, Gavin Brown, Mikel Lujan, Ian Watson, and Marcelo Cintra. Toward a more accurate understanding of the limits of the TLS execution paradigm. In *Proceedings of the 2010 IEEE International Symposium on Workload Characterization*, IISWC'10, pages 1–12, December 2010.

[90] Will Iverson. *Apache Jakarta Commons: Reusable Java (TM) Components (Bruce Perens' Open Source Series)*. Prentice Hall PTR, 2005. Last access: April 2014. http://commons.apache.org/collections.

[91] Haoqiang Jin, Gabriele Jost, Jerry Yan, Eduard Ayguade, Marc Gonzalez, and Xavier Martorell. Automatic multilevel parallelization using OpenMP. In *Proceedings of the Third European Workshop on OpenMP*, volume 11(2) of *Journal of Scientific Programming, EWOMP'11*, pages 177–190, April 2003.

[92] Stephen Johnson, Emyr Evans, Haoqiang Jin, and Constantinos Ierotheou. The ParaWise expert assistant - Widening accessibility to efficient and scalable tool generated OpenMP code. In *Proceedings of the Fifth International Conference on OpenMP Applications and Tools*, Lecture Notes in Computer Science, WOMPAT'04, pages 67–82, 2005.

[93] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Speculative thread decomposition through empirical optimization. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'07, pages 205–214, 2007.

[94] Michael Kay. Saxonica: XSLT and XQuery processing. Last access: April 2014. http://saxonica.com.

[95] Michael Kay. XSL transformations (XSLT) version 2.0. W3C recommendation 23 january 2007. Last access: April 2014. http://www.w3.org/TR/xslt20/.

[96] Arun Kejariwal, Xinmin Tian, Milind Girkar, Wei Li, Sergey Kozhukhov, Utpal Banerjee, Alexander Nicolau, Alexander V. Veidenbaum, and Constantine D. Polychronopoulos. Tight analysis of the performance potential of thread speculation using spec CPU 2006. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'07, pages 215–225, 2007.

[97] Arun Kejariwal, Xinmin Tian, Wei Li, Milind Girkar, Sergey Kozhukhov, Hideki Saito, Utpal Banerjee, Alexandru Nicolau, Alexander V. Veidenbaum, and Constantine D. Polychronopoulos. On the performance potential of different types of speculative thread-level parallelism. In *Proceedings of the 20th ACM/IEEE International Conference on Supercomputing*, ICS'06, pages 24–36, 2006.

[98] Arun Kejariwal, Alexander V. Veidenbaum, Alexander Nicolau, Xinmin Tian, Milind Girkar, Hideki Saito, and Utpal Banerjee. Comparative architectural characterization of SPEC CPU2000 and CPU2006 benchmarks on the intel® core™ 2 duo processor. In *Proceedings of the 2008 IEEE International Conference of Embedded Computer Systems: Architectures, Modeling, and Simulation*, IC-SAMOS'08, pages 132–141, 2008.

[99] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. SD3: A scalable approach to dynamic data-dependence profiling. In *Proceedings of the 43rd IEEE/ACM International Symposium on Microarchitecture*, MICRO'43, pages 535–546, 2010.

[100] Xiangyun Kong, David Klappholz, and Kleanthis Psarris. The I test: An improved dependence test for automatic parallelization and vectorization. *IEEE Transactions on Parallel Distributed Systems*, 2(3):342–349, July 1991.

[101] Clyde P. Kruskal and Alan Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, SE-11(10):1001–1016, 1985.

[102] Milind Kulkarni, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. Optimistic parallelism benefits from data partitioning. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'08, pages 233–243, 2008.

[103] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'07, pages 211–222, June 2007.

[104] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[105] James Larus and Christos Kozyrakis. Transactional memory. *Communications of the ACM*, 51(7):80–88, July 2008.

[106] Chris Lattner and Vikram Adve. LLVM: a compilation framework for lifelong program analysis transformation. CGO'04, pages 75–86, 2004.

[107] Shengmei Li, Lin Qiao, Zhizhong Tang, Buqi Cheng, and Xingyu Gao. Performance characterization of SPEC CPU2006 benchmarks on intel and AMD platform. In *Proceedings of the First International Workshop on Education Technology and Computer Science*, ETCS'09, pages 116–121, 2009.

[108] 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), Edmonton, Canada*, 2008.

[109] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: A TLS compiler that exploits program structure. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'06, pages 158–167, 2006.

[110] Diego R. Llanos. Thread-Level speculative parallelization. In P. Alberigo, G. Erbacci, and F. Garofalo, editors, *Science and Supercomputing in Europe, 2004 Annual Report*, pages 211–213. CINECA, Italy, 2005. ISBN 88-86037-15-5.

[111] Yangchun Luo, Venkatesan Packirisamy, Wei-Chung Hsu, Antonia Zhai, Nikhil Mungre, and Ankit Tarkas. Dynamic performance tuning for speculative threads. In *Proceedings of the 36th International Symposium on Computer Architecture*, ISPA'09, June 2009.

[112] Kazuaki Maeda. Experience of XML-Based source code representation with parsing actions. In *Proceedings of the Sixth conference on New Trends in Software Methodologies, Tools and Techniques*, SoMeT'07, pages 330–339, 2007.

[113] David Malcolm. GCC python plugin v0.12. Last access: April 2014. `https://fedorahosted.org/gcc-python-plugin/`, 2013.

[114] Jonathan I. Maletic, Michael L. Collard, and Huzefa Kagdi. Leveraging XML technologies in developing program analysis tools. In *Proceedings of Fourth International Workshop on Adoption-Centric Software Engineering*, ACSE'04, pages 80–85, McLean, Virginia, USA, May 2004.

[115] Evan Mamas and Kostas Kontogiannis. Towards portable source code representations using XML. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, WCRE'00, pages 172–182, 2000.

[116] Pedro Marcuello and Antonio González. Clustered speculative multithreaded processors. In *Proceedings of the 13th ACM/IEEE International Conference on Supercomputing*, ICS'99, pages 365–372, 1999.

[117] José F. Martínez and Josep Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'02, pages 18–29, October 2002.

[118] Katsuhisa Maruyama and Shinichiro Yamamoto. A CASE tool platform using an XML representation of java source code. In *Proceedings of the Fourth IEEE International*

Workshop on Source Code Analysis and Manipulation, SCAM'04, pages 158–167, 2004.

[119] Gregory McArthur, John Mylopoulos, and Siu Kee Keith Ng. An extensible tool for source code representation using XML. In *Proceedings of the Ninth Working Conference on Reverse Engineering*, WCRE'02, pages 199–208, 2002.

[120] Nabor C. Mendonça, Paulo Henrique M. Maia, Leonardo A. Fonseca, and Rossana M. C. Andrade. RefaX: A refactoring framework based on XML. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ISCM'04, pages 147–156, 2004.

[121] Jason Merrill. Generic and gimple: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers' Summit*, pages 171–179, Otawa, Canada, 2003.

[122] Miloš Milovanović, Roger Ferrer, Vladimir Gajinov, Osman S. Unsal, Adrian Cristal, Eduard Ayguadé, and Mateo Valero. Multithreaded software transactional memory and OpenMP. In *Proceedings of the 2007 workshop on MEmory performance: DEaling with Applications, systems and architecture*, MEDEA'07, pages 81–88, 2007.

[123] Miloš Milovanović, Roger Ferrer, Osman S. Unsal, Adrian Cristal, Xavier Martorell, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Transactional memory and OpenMP. In *Proceedings of the 3rd International Workshop on OpenMP*, volume 4935 of *Lecture Notes in Computer Science, IWOMP'07*, pages 37–53, 2007.

[124] Tipp Moseley, Daniel A. Connors, Dirk Grunwald, and Ramesh Peri. Identifying potential parallelism via loop-centric profiling. In *Proceedings of the 4th International Conference on Computing Frontiers*, CF'07, pages 143–152, 2007.

[125] Ernst P. Mücke, Isaac Saias, and Binhai Zhu. Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations. In *Proceedings of the 12th ACM Symposium on Computational Geometry*, SCG'96, pages 274–283, 1996.

[126] Robert Munafo. Reference machine times for SPEC CPU2006. Last access: April 2014. `http://www.mrob.com/pub/comp/benchmarks/spec.html`.

[127] Luigi Nardi, Fouad Badran, Pierre Fortin, and Sylvie Thiria. YAO: A generator of parallel code for variational data assimilation applications. In *Proceedings of the IEEE 14th International Conference on High Performance Computing and Communications*, HPCC'12, pages 224–232, June 2012.

[128] Diego Novillo. From source to binary: The inner workings of GCC. Last access: April 2014. `http://www.redhat.com/magazine/002dec04/features/gcc/`, December 2004.

[129] Diego Novillo. GCC an architectural overview, current status, and future directions. In *Proceedings of the Linux Symposium*, pages 185–200, Tokyo, Japan, September 2006.

[130] Diego Novillo. OpenMP and automatic parallelization in GCC. In *Proceedings of the 2006 GCC Developers' Summit*, pages 135–144, Ottawa, Canada, 2006.

[131] Diego Novillo. GCC internals (slides). In *International Symposium on Code Generation and Optimization*, CGO'07, March 2007.

[132] OpenMP Architecture Review Board. The OpenMP application program interface specification, version 3.1. `http://openmp.org`, July 2011.

[133] Venkatesan Packirisamy and Harish Barathvajasankar. OpenMP in multicore architectures. *University of Minnesota, Tech. Rep*, 2005.

[134] Venkatesan Packirisamy, Antonia Zhai, Wei-Chung Hsu, Pen-Chung Yew, and Tin-Fook Ngai. Exploring speculative parallelism in SPEC2006. In *Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS'09, pages 77–88, April 2009.

[135] Ken Polsson. Chronology of Workstation Computers. 1987-1990. Last access: April 2014. `http://www.islandnet.com/~kpolsson/workstat/work1987.htm`.

[136] Ken Polsson. Chronology of Workstation Computers. 1991-1992. Last access: April 2014. `http://www.islandnet.com/~kpolsson/workstat/work1991.htm`.

[137] James F. Power and Brian A. Malloy. Program annotation in XML: A Parse-Tree based approach. In *Proceedings of the Ninth Working Conference on Reverse Engineering*, WCRE'02, pages 190–198, 2002.

[138] Kleanthis Psarris and Konstantinos Kyriakopoulos. An experimental evaluation of data dependence analysis techniques. *IEEE Transactions on Parallel and Distributed Systems*, 15(3):196–213, March 2004.

[139] William Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, SC'91, pages 4–13, 1991.

[140] Hanson Prihantoro Putro and Inggriani Liem. XML representations of program code. In *Proceedings of the First International Conference on Electrical Engineering and Informatics*, ICEEI'11, pages 1–6, 2011.

[141] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'10, pages 65–76, 2010.

[142] Lawrence Rauchwerger and David Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, PLDI'95, pages 218–232, 1995.

[143] Sara Royuela, Alejandro Duran, Chunhua Liao, and Daniel J. Quinlan. Auto-scoping for OpenMP tasks. In *Proceedings of the Eighth International Workshop on OpenMP*, number 7312 in Lecture Notes in Computer Science, IWOMP'12, pages 29–43, January 2012.

[144] Alberto Salinas-Mendoza, Ulises Juarez-Martinez, Giner Alor-Hernandez, and Beatriz A. Olivares-Zepahua. Designing an XML-based representation for CaesarJ source code. In *Proceedings of the 2011 IEEE Electronics, Robotics and Automotive Mechanics Conference*, CERMA'11, pages 427–432, 2011.

[145] Dale Schouten, Xinmin Tian, Aart Bik, and Milind Girkar. Inside the Intel compiler. *Linux Journal*, 2003(106):4, 2003.

[146] Justin Seyster, Ketan Dixit, Xiaowan Huang, Radu Grosu, Klaus Havelund, Scott A. Smolka, Scott D. Stoller, and Erez Zadok. InterAspect: Aspect-oriented instrumentation with GCC. *Formal Methods in System Design*, 41(3):295–320, December 2012.

[147] Susan Elliott Sim, Steve Easterbrook, and Richard C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE'03, pages 74–83, 2003.

[148] Michael Spendolini. *Benchmarking*. Grupo Editorial Norma, 2005.

[149] Cloyce D. Spradling. SPEC CPU2006 benchmark tools. *SIGARCH Computer Architecture News*, 35(1):130–134, March 2007.

[150] Standard Performance Evaluation Corporation. Last access: April 2014. `http://www.spec.org/`.

[151] Basile Starynkevitch. MELT: A translated domain specific language embedded in the GCC compiler. In *Proceedings of the IFIP Working Conference on Domain-Specific Languages*, DSL'11, pages 118–142, 2011.

[152] Guy L. Steele Jr. Parallel programming and code selection in Fortress. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'06, page 1, 2006.

[153] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th International Symposium on Computer Architecture*, ISCA'00, pages 1–12, 2000.

[154] Yu Xia Sun, Huo Yan Chen, and T. H Tse. Lean implementations of software testing tools using XML representations of source codes. In *Proceedings of the 2008 International Conference on Computer Science and Software Engineering - Volume 02*, pages 708–711, 2008.

[155] Sun Microsystems. Sun Studio 12 Product Library Documentation. Last access: April 2014. `http://docs.oracle.com/cd/E19205-01/`.

[156] Julien Taillard, Frédéric Guyomarc'h, and Jean-Luc Dekeyser. A graphical framework for high performance computing using an MDE approach. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, PDP'08, pages 165–173, February 2008.

[157] Chen Tian, Min Feng, and Rajiv Gupta. Speculative parallelization using state separation and multiple value prediction. In *Proceedings of the 2010 international symposium on Memory management*, ISMM'10, pages 63–72, 2010.

[158] Doug Tidwell. *XSLT, 2nd Edition*. O'Reilly Media, 2 edition, June 2008.

[159] Massimo Torquati, Marco Vanneschi, Mehdi Amini, Serge Guelton, Ronan Keryell, Vincent Lanore, François-Xavier Pasquier, Michel Barreteau, Rémi Barrère, Claudia-Teodora Petrisor, et al. An innovative compilation tool-chain for embedded multi-core architectures. In *Embedded World Conference (February 2012)*.

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

[161] Ten H. Tzen and Lionel M. Ni. Trapezoid self-scheduling: A pratical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, 1993.

[162] Christoph von Praun, Rajesh Bordawekar, and Calin Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 185–196, 2008.

[163] Michael Voss, Eric Chiu, Patrick Man Yan Chow, Catherine Wong, and Kevin Yuen. An evaluation of auto-scoping in OpenMP. In Barbara M. Chapman, editor, *Proceedings of the 5th International Workshop on OpenMP Applications and Tools*, number 3349 in Lecture Notes in Computer Science, WOMPAT'04, pages 98–109, January 2005.

[164] Christian Wagner, Tiziana Margaria, and Hans-Georg Pagendarm. Analysis and code model extraction for C/C++ source code. In *Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS'09, pages 110–119, 2009.

[165] Shengyue Wang, Xiaoru Dai, Kiran S. Yellajyosula, Antonia Zhai, and Pen-Chung Yew. Loop selection for thread-level speculation. In *Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing*, LCPC'05, pages 289–303, 2006.

[166] Emo Welzl. Smallest enclosing disks (balls and ellipsoids). In *New results and new trends in computer science*, volume 555 of *Lecture Notes in Computer Science*, pages 359–370. Springer-Verlag, 1991.

[167] John Whaley and Christos Kozyrakis. Heuristics for profile-driven method-level speculative parallelization. In *Proceedings of the International Conference on Parallel Processing Workshops*, ICPP'05, pages 147–156, June 2005.

[168] David A. Wheeler. Sloccount: Counting source lines of code. Last access: April 2014. `http://www.dwheeler.com/sloccount`.

[169] Michael Wong, Barna L. Bihari, Bronis R. de Supinski, Peng Wu, Maged Michael, Yan Liu, and Wang Chen. A case for including transactions in OpenMP. In *Proceedings*

of the Eighth International Workshop on OpenMP, volume 6132 of *Lecture Notes in Computer Science, IWOMP'10*, pages 149–160, 2010.

[170] Peng Wu, Arun Kejariwal, and Călin Caşcaval. Compiler-driven dependence profiling to guide program parallelization. In José Amaral, editor, *Languages and Compilers for Parallel Computing*, volume 5335 of *Lecture Notes in Computer Science*, pages 232–248. Springer Berlin / Heidelberg, 2008.

[171] Polychronis Xekalakis, Nikolas Ioannou, and Marcelo Cintra. Combining thread level speculation helper threads and runahead execution. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS'09, pages 410–420, 2009.

[172] Dong Ye, Joydeep Ray, Christophe Harle, and David Kaeli. Performance characterization of SPEC CPU2006 integer benchmarks on x86-64 architecture. In *Proceedings of the 2006 IEEE International Symposium on Workload Characterization*, IISWC'06, pages 120–127, 2006.

[173] Ye Zhang, Lawrence Rauchwerger, and Josep Torrellas. Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, HPCA'98, pages 162–173, February 1998.

[174] Ying Zou and Kostas Kontogiannis. A framework for migrating procedural code to Object-Oriented platforms. In *Proceedings of the Eighth Asia-Pacific on Software Engineering Conference*, APSEC'01, pages 390–399, 2001.