Tesis de Máster

Curso 2007–2008

Máster en Ingeniería de Computadores

Detección automática de paralelismo a nivel de función

Autor: **David Yuste Romero**

Directores: **Julio Sahuquillo Borrás**

José Francisco Duato Marín

Índice

Resumen

Debido a los avances tecnológicos, las tendencias actuales en diseño de arquitecturas del procesador se centran en ubicar múltiples núcleos en un mismo chip en lugar de continuar incrementando la complejidad de los tradicionales procesadores con un solo núcleo. Estos nuevos procesadores, además de explotar el paralelismo a nivel de instrucción (ILP) explotan el paralelismo a nivel de tarea (TLP).

Para explotar este tipo de paralelismo los programadores deberían implementar código paralelo. Desafortunadamente, en la actualidad la popularidad de la programación paralela es escasa, por tanto, los desarrolladores infraaprovechan el potencial hardware que ofrece la máquina. Otra alternativa válida es que los compiladores y/o el hardware intenten extraer el máximo paralelismo de los programas a fin de alimentar a los distintos núcleos y mejorar las prestaciones de la máquina.

La mayor parte de la investigación realizada en materia de paralelización automática se ha enfocado al paralelismo a nivel de bucle, a través de complejos análisis de dependencia entre iteraciones. No obstante, para garantizar la corrección del programa paralelo resultante, se ejecutan complejos algoritmos que deben ejecutarse junto al programa final. En otros casos se hace uso exhaustivo de especulación (es decir, predicción de datos y control). Desafortunadamente, todas estas técnicas incrementan considerablemente la sobrecarga en tiempo de ejecución.

Para evitar estos inconvenientes deben desarrollarse nuevas formas de detectar tareas de forma eficiente. Este proyecto propone un modo de explotar TLP a través de una técnica basada en compilación que trabaja a nivel de función. Esta técnica detecta tareas más largas que las halladas mediante técnicas como las que trabajan a nivel de bucle reduciendo además la sobrecarga en tiempo de ejecución.

Para la detección de paralelismo a nivel de función *se ha implementado un paso de análisis en el compilador GNU Compiler Collection (GCC) que genera una caracterización de los fuentes a compilar. Esta información se utiliza posteriormente para realizar un análisis interprocedural, que permite establecer relaciones de dependencia entre llamadas a función. Estas relaciones permiten detectar el paralelismo en el programa analizado.*

Con la finalidad de evaluar los resultados del análisis se ha implementado un simulador que estima la mejora obtenida al aplicar las técnicas de paralelización propuestas.

Capítulo 1

Introducción

Tradicionalmente, los procesadores superescalares han mejorado sus prestaciones mediante sofisticadas técnicas para explotar paralelismo a nivel de instrucción (ILP). La mayor parte de estos procesadores implementan ejecución fuera de orden, que permite el lanzamiento a ejecución en un mismo ciclo de varias instrucciones independientes, solapando de este modo su ejecución. Además, la ejecución especulativa (p. ej., predicción de saltos y datos) incrementa los beneficios de estas técnicas.

La propuesta de técnicas para incrementar el ILP tomó importancia tanto en la industria como en investigación. Esto fue debido a los avances tecnológicos que hicieron posible introducir más transistores en un mismo chip, permitiendo el diseño de procesadores más complejos y potentes.

1.1 Descripción de la problemática actual

En la actualidad, la evolución de las arquitecturas basadas en un solo procesador se está viendo limitada por restricciones tecnológicas. Por un lado, la elevada frecuencia de los nuevos procesadores acarrea problemas relacionados con la propagación de señales globales como la señal de reloj, que no

pueden alcanzar toda el área de silicio en un solo ciclo. Consecuentemente, la comunicación entre partes distantes del procesador requiere varios ciclos de reloj. Por otro lado, incrementar la frecuencia también conlleva un incremento de consumo eléctrico y calor disipado, haciendo costoso incrementar la velocidad del procesador.

Estas razones han hecho que las microarquitecturas avancen en una nueva dirección. En lugar de centrarse en mejorar un complejo procesador basado en un solo núcleo, los nuevos enfoques intentan mejorar las prestaciones a través de arquitecturas basadas en múltiples núcleos en un mismo chip, también llamados procesadores multi-núcleo. De esta forma se persigue mejorar las prestaciones habilitando varios núcleos para trabajar en paralelo. Al mismo tiempo, y debido a razones relacionadas con la energía y el calor disipado, cada núcleo tiene un diseño más sencillo que los utilizados por las arquitecturas anteriores.

Con la puesta en práctica de este nuevo enfoque arquitectónico, el ILP por sí mismo pierde protagonismo tanto en la industria como en investigación. El nuevo modelo de paralelismo a explotar es el paralelismo a nivel de tarea (TLP), que consiste en la ejecución solapada de instrucciones independientes procedentes de distintos contextos, es decir, hilos o tareas. Así, los programas con múltiples hilos explícitos pueden beneficiarse de los procesadores multi-núcleo, dado que cada hilo puede ejecutarse por separado. Sin embargo, los programas secuenciales existentes no pueden beneficiarse de estas arquitecturas sin ser paralelizados.

1.2 Paralelismo a nivel de tarea

El término tarea ha sido usado en referencia a diferentes conceptos (p. ej., iteración de bucle). En esta Tesina se usará para definir una secuencia de sentencias de un programa (incluyendo operaciones, instrucciones de control y llamadas a función) que mantienen una relación de dependencia entre ellos. Estos sentencias pueden ser contiguos o no, y su relación explícita, como es el caso de las dependencias de datos a través de parámetros de llamada a función y variables locales, o más sofisticadas, implicando variables globales y dinámicas, llamadas a librería o través de punteros a función.

La ejecución concurrente de varias tareas independientes constituye el TLP. Sin embargo, para poder explotar el TLP es necesario que estas tareas hayan sido explicitadas, ya sea por el programador a través de las herramientas proporcionadas por el lenguaje, o mediante técnicas automáticas basadas en software (técnicas en tiempo de compilación), en hardware (técnicas en tiempo de ejecución) o híbridas. En consecuencia, el desafío que se presenta consiste en proporcionar al programador herramientas que permitan explotar el TLP presente en sus programas al ser ejecutados en procesadores multinúcleo. Estas herramientas deben permitir dividir el programa en tareas independientes. Es de especial interés que estas técnicas sean escalables en cuanto al número de procesadores de la arquitectura subyacente, para un buen aprovechamiento del cada vez mayor número de núcleos en un mismo chip. Además deben ser transparentes para el desarrollador, posibilitando la paralelización de los programas secuenciales existentes sin necesidad de volver a reescribirlos.

Gurindar S. Sohi establece en [1] una taxonomía que permite clasificar las distintas formas de dividir un programa en tareas independientes. Así, se distingue entre las tareas guiadas por flujo de control o de saltos y las guiadas

por flujo de datos. Además, en ambos casos puede distinguirse entre tareas especulativas o no especulativas, según se utilicen mecanismos de predicción o no, para delimitar o lanzar a ejecución las tareas. A continuación se expone un análisis más detallado de cada uno de estos métodos de detección de tareas y algunos de los estudios más relevantes entorno a ellos.

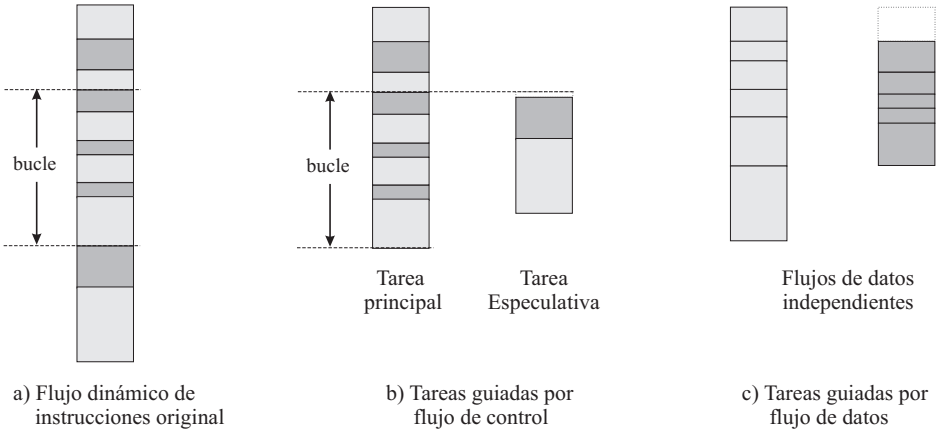


Figura 1.1: Detección automática de tareas en un programa secuencial (a) mediante dos métodos distintos de detección de tareas (b y c)

1.2.1 Tareas guiadas por flujo de control

Bajo este enfoque, el programa se divide a lo largo de las fronteras dadas por el flujo de control. El flujo dinámico de instrucciones, es decir, la secuencia de instrucciones que se obtiene al ejecutar el programa, se divide en segmentos contiguos que el sistema puede lanzar a ejecución concurrentemente como tareas. Éstas pueden ser, entre otras, bucles, iteraciones de bucle, o secuencias de bloques básicos que comienzan en puntos específicos como la instrucción siguiente a una llamada a función.

La principal dificultad en la detección de este tipo de tareas, estriba en determinar las circunstancias bajo las cuales pueden ser ejecutadas concur-

rentemente, y en garantizar la consistencia de memoria, es decir, que los accesos a memoria realizados por las mismas garantizan la corrección semántica del programa secuencial.

Las aproximaciones no especulativas son escasas y con un ámbito de aplicación limitado (p. ej., computación simbólica [2]). Esto es así porque en ausencia de mecanismos de recuperación debe garantizarse la independencia entre las tareas que van a ejecutarse concurrentemente. Sin embargo, en lenguajes imperativos es difícil determinar a priori si los accesos a memoria de una tarea no van a entrar en conflicto con los de otras.

Para relajar estas restricciones se hace uso de especulación. En [3] y [4] se presentan técnicas para explotar especulativamente TLP con tareas guiadas por flujo de control. Estas técnicas comienzan la ejecución no especulativa de la tarea principal. Al alcanzar ciertas instrucciones se crean tareas especulativas que comienzan en fragmentos de código que se espera que sean ejecutados próximamente. De esta forma, al alcanzar una instrucción de llamada a procedimiento se crearía una tarea especulativa con inicio en la instrucción siguiente a la llamada. Similarmente, al alcanzar la primera instrucción de un bucle, puede ejecutarse especulativamente el código siguiente al bucle.

Sin embargo, el valor de los datos de entrada de las tareas especulativas (p. ej., el valor devuelto por una llamada a función) no es conocido en el momento de su lanzamiento, sino que debe ser proporcionado por un predictor de valor. La precisión de estos predictores [5] a menudo compromete la eficacia de estas técnicas, debido a la sobrecarga implicada por los fallos de predicción.

Mejorando esta propuesta, el compilador *Mitosis* [6] genera código paralelo basado en esta misma metodología, aunque implementando una técnica eficiente de predicción de valores. Cada valor de entrada para la tarea pa-

ralela es calculado heurísticamente: la tarea contiene, además del código paralelizado, un prólogo encargado de precalcular los valores de entrada. Se trata de una simplificación del código originalmente encargado de producir estos valores, en la que los caminos de ejecución más improbables se eliminan, bajo la hipótesis de que no serán ejecutados.

Además, para determinar las variables que requieren ser predichas (las generadas por la tarea predecesora) se aplica un mecanismo similar, considerando únicamente las variables cuya probabilidad de ser escritas por la tarea predecesora y consumidas por la sucesora supera cierto umbral. Si esta suposición falla, se detecta durante el prológo de la tarea especulativa, produciéndose una cancelación anticipada de la misma. El principal inconveniente de esta técnica es la sobrecarga causada por el prólogo en las tareas especulativas.

También en el ámbito de tareas guiadas por flujo de control los investigadores han mostrado especial interés por la paralelización especulativa de bucles [7][8][9], que consiste en la ejecución concurrente de lotes de iteraciones contiguas, presuntamente independientes. El problema con el que se enfrentan estas técnicas, es la dificultad de alcanzar un tamaño de lote (número de iteraciones) que consiga un buen compromiso entre la sobrecarga ocasionada (más significativa cuanto más reducido es el tamaño de los lotes) y la penalización impuesta por los fallos de especulación (crecientes a medida que aumenta el tamaño de los lotes). En [10] y [11] se propone una estrategia para la planificación de estos lotes de iteraciones, aplicada a bucles irregulares como los que aparecen en los algoritmos incrementalmente aleatorizados, usados en Geometría Computacional y Optimización, donde el número de dependencias entre iteraciones tiende a disminuir a medida que el algoritmo avanza. Bajo esta técnica, inicialmente se utilizan lotes que con-

tienen un reducido número de iteraciones, al alcanzar cierto umbral en el que la probabilidad de dependencias entre iteraciones cae, se aumenta el tamaño de los lotes, reduciendo la sobrecarga. El problema de esta técnica es que su aplicación está restringida a algoritmos incrementalmente aleatorizados, y además el umbral en el que decaen las dependencias es dependiente del algoritmo en cuestión.

Pese a que la mayor parte de los esfuerzos en esta materia se han centrado en la paralelización de bucles, recientes estudios [12] ponen en duda los beneficios que pueden obtenerse con estos métodos. Éstos muestran que la cota superior en la aceleración que es posible alcanzar por medio de paralelismo de bucles especulativo es sólo de un 1%.

1.2.2 Tareas guiadas por flujo de datos

Para el caso de tareas guiadas por flujo de datos, el principal criterio de división viene dado por las dependencias de datos. Cada tarea consta de unas entradas (los datos que requiere) y genera una salidas (los datos que escribe). Cuando todas las entradas de una tarea están disponibles, se considera que la tarea está lista para ejecutarse.

A diferencia de las tareas guiadas por flujo de control, las instrucciones que componen este tipo de tareas pueden no ser contiguas. Las técnicas que persiguen explotar TLP bajo este enfoque deben detectar las instrucciones dependientes y agruparlas. En su mayoría, se trata de técnicas fuertemente orientadas hacia la compilación, basadas en exhaustivos análisis de dependencias [13] [14].

Existen también abundantes aproximaciones que trabajan a nivel de bucle como las realizadas por Saltz y colaboradores [15] [16]. En éstas, tras un análisis en tiempo de compilación, se modifica la estructura del bucle, inser-

tando código para preprocesar las relaciones entre iteraciones en tiempo de ejecución (*inspector*), y una versión modificada del cuerpo del bucle (*executor*). El análisis resultante de la ejecución de *inspector* permite determinar la relación de dependencia entre iteraciones y planificarlas paralelamente. Pese a tratarse de una excelente aproximación a paralelismo de tarea a nivel de bucle, que ha servido de base para numerosas investigaciones, su principal limitación es la sobrecarga debido a la ejecución secuencial del código *inspector*. De hecho, una gran cantidad de trabajos [17][18] se han centrado en reducir esta sobrecarga.

Un enfoque novedoso en la búsqueda de paralelismo persigue explotar paralelismo de datos más allá del ámbito de bucles. Estas técnicas pretenden explotar paralelismo a nivel de función. En [19] se propone un marco de trabajo para estimar el paralelismo potencial en programas. Las dependencias de datos son detectadas por medio de *profiling*, es decir, insertando instrucciones de instrumentación en el programa original y monitorizando su ejecución. Las dependencias de datos detectadas se utilizan para construir el *grafo de flujo de datos interprocedural* y el *grafo de datos compartidos*. El primero muestra qué funciones consumen datos escritos por otros. El segundo muestra la forma en que los datos son compartidos. Ambos son utilizados para seleccionar las construcciones paralelas más adecuadas (p. ej., maestro/esclavo, pila de trabajo, *pipeline*). Finalmente, esta información es utilizada para paralelizar el programa. Los inconvenientes de esta aproximación es que requiere realizar *profiling* para detectar las dependencias de datos. Dado que la ejecución concreta analizada depende de los datos de entrada, las dependencias detectadas son optimistas e inseguras.

1.3 Objetivos y estructura de la Tesina

La mayor parte de las técnicas desarrolladas en la línea de detección de tareas guiadas por flujo de datos se ven restringidas principalmente por dos limitaciones: el elevado coste computacional de la fase de análisis de dependencias y/o un ámbito de aplicación muy limitado dentro del programa (p. ej., bucles). Esta Tesina propone una estrategia que pretende relajar el coste computacional sin restringir su ámbito de aplicación a bucles o bloques básicos. Para tal fin, hemos considerado la función (más concretamente la llamada a función) como unidad mínima constituyente de la tarea.

En los lenguajes de programación imperativos y orientados a objetos, la función describe una transformación sobre unos datos de entrada generando unas salidas, o bien bajo un enfoque procedural, realiza una secuencia de acciones relacionadas entre sí. Por lo tanto, cabe esperar que las acciones realizadas por una llamada a función estén relacionadas. Bajo esta hipótesis se toma la determinación de buscar tareas en el marco de la llamada a función.

Al buscar paralelismo a nivel de llamada a función aumenta la granularidad del análisis, ya que no es necesario estudiar las relaciones entre todos los sentencias del programa. De esta forma, se reduce significativamente el tiempo de análisis en compilación.

Capítulo 2

Espacio de trabajo

En esta Tesina se describe una técnica de análisis de programas y compilación paralela que ha sido implementada en el compilador GCC (GNU Compilers Collection). En este capítulo se introducen los fundamentos de GCC que definen el espacio de trabajo en el que la técnica ha sido desarrollada. La arquitectura de GCC consta de tres etapas principales (ver figura 2.1), que se resumen a continuación:

1. *front end*. Toma como entrada código fuente escrito en cualquiera de los lenguajes soportados por el compilador y lo traduce a una representación génerica independiente del lenguaje. Esta representación recibe el nombre GENERIC. El código fuente, traducido a GENERIC se representa explícitamente en memoria mediante una estructura polifórmica en forma de árbol (tipo de datos *tree*). Sin embargo, esta gramática sigue siendo demasiado compleja para aplicar procesos de optimización automatizada. Para solucionarlo se introduce GIMPLE, una simplifiación de GENERIC con fuertes restricciones sintácticas.

2. *middle end*. Esta etapa de GCC constituye el grueso principal del compilador. Consta de una secuencia de pasos de optimización y transfor-

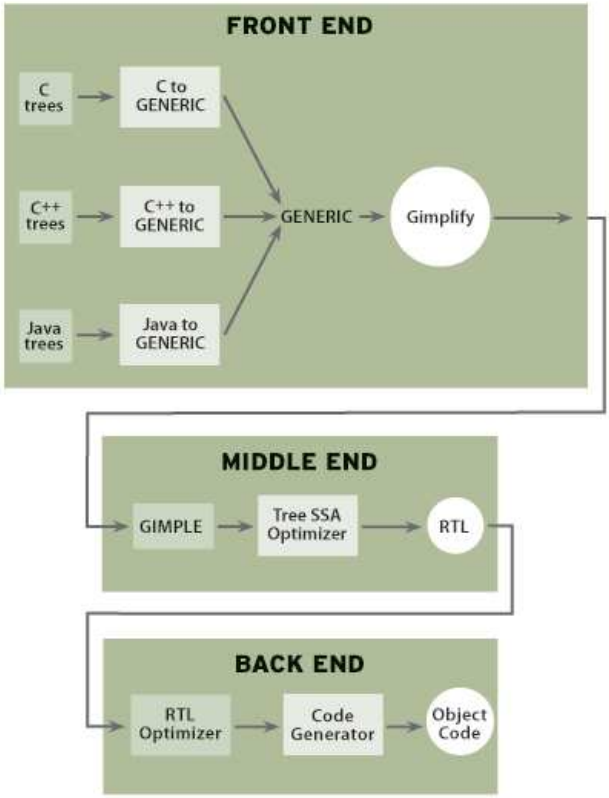


Figura 2.1: Arquitectura de GCC

mación que operan sobre el código representado como un árbol GIMPLE. Cada paso requiere que el árbol presente ciertas propiedades, y aporta o destruye otras (p. ej., forma SSA, grafo de control de flujo, etc.). La propia arquitectura de GCC se encarga de garantizar que las propiedades requeridas por un paso se garantizan invocando previamente a los pasos que las proporcionan.

3. *back end*. Tras aplicar los distintos pasos de optimización el árbol se expande y representa en RTL (Register Transfer Language). En este punto el código se especializa de acuerdo a las características específicas de la máquina objetivo.

2.1 Gramática GIMPLE

La gramática de un lenguaje imperativo de alto nivel a menudo aporta una complejidad excesiva e innecesaria para tareas de análisis, dada la enorme casuística en que pueden derivar sus expresiones. Por esta razón se usan gramáticas más sencillas como GIMPLE [20] (para más detalles ver Apéndice A). GIMPLE es actualmente la representación intermedia de alto nivel utilizada por GCC. Su gramática proporciona una representación independiente del lenguaje muy adecuada para realizar análisis y optimizaciones.

GIMPLE es una representación tres direcciones en la que cada sentencia, como máximo, toma dos valores de entrada y genera una salida. Excepcionalmente, un sentencia que realiza una llamada a función tiene asociada una lista de argumentos. Éstos serán siempre variables o constantes, ya que GIMPLE no permite expresiones en la lista de argumentos de la llamada. A diferencia de otras representaciones de código tres direcciones orientadas a la producción de código ensamblador, las variables en GIMPLE conservan

el tipado del lenguaje de alto nivel original, aportando de esta forma una gran cantidad de información al analizador que toma esta gramática como entrada.

Los sentencias de control de flujo también se simplifican en GIMPLE, quedando reducidos a expresiones *goto* explícitas, posiblemente en combinación con expresiones condicionales. Una expresión condicional es un sentencia que, dado un valor condición, ejecuta una instrucción dada en caso de evaluarse como verdadera u otra en caso contrario. Adicionalmente se conserva la expresión de control de flujo *switch*.

Así, una expresión *if-then-else* en GIMPLE se expresa como una expresión condicional que toma como variable condición el resultado de evaluar la condición asociada a la sentencia *if* (la variable temporal que contiene el resultado de su evaluación), y cuyas alternativas de ejecución son una instrucción *goto* hacia el cuerpo del bloque *then* y otra hacia el cuerpo del bloque *else*.

En resumen, las características de una gramática como la de GIMPLE facilitan la implementación de optimizadores y analizadores, reduciendo la casuística del lenguaje a la que éstos deben enfrentarse. En esta Tesina trabajaremos también con esta gramática, ya que recoge toda la información requerida para el análisis propuesto y reduce notablemente la complejidad del analizador.

2.2 Usos y Definiciones

Frecuentemente, a lo largo de esta Tesina, haremos referencia a *variables escritas* o *variables leídas* por sentencias o llamadas a función. Existen dos términos comúnmente aceptados en el ámbito de compiladores para denotar estas variables, se trata de *usos* y *definiciones*. Así, los *usos* de un sen-

tencia son las variables que dicho sentencia *usa* o lee. Análogamente, sus *definiciones* son las variables que *define* o escribe.

Cuando un sentencia *define* una variable crea una nueva versión de la misma, y los sentencias *siguientes* que usen esta variable usarán la versión definida por dicho sentencia hasta que la variable vuelva a ser *definida*. De esta forma, aparece el concepto de *alcance* de la *definición*. Para una variable A y un sentencia S_i que la define, el alcance de la *definición* de A realizada por el sentencia S_i es el conjunto de sentencias S_j que, en caso de usar A, podrían usar la versión definida por S_i. Por simplicidad, diremos que una *definición* alcanza cierto *uso*, cuando el sentencia en el que se da este *uso* pertenece al *alcance* de dicha *definición*.

Es importante destacar que en un análisis estático no se puede determinar con certeza el *alcance* de una *definición*, dado que éstas a menudo aparecen en caminos alternativos del *grafo de control de flujo* (*CFG*). Por lo tanto, estáticamente, un *uso* dado puede pertenecer al *alcance* de varias *definiciones*, aunque en tiempo de ejecución sólo sea alcanzado por una de ellas.

2.2.1 Forma SSA

Los conceptos descritos anteriormente son de especial relevancia en la forma *Static Single Assginment* (SSA) [21][22]. SSA es una representación en la cual cada variable es asignada una sola vez. Con cada asignación de una variable en la representación original se crea una nueva versión de la mis-ma en la representación SSA. De esta forma, cada *uso* de una variable depende exactamente de una *definición*.

Cuando convergen varios caminos del *CFG*, las definiciones sobre la misma variable efectuadas en ellos son agrupadas bajo el operador Φ. Un *uso*

del operador $\Phi(v1..vn)$, donde $v1..vn$ es un conjunto de variables (más concretamente, versiones de variables), debe interpretarse como un *uso* de alguna de las variables $vi \in v1..vn$, aunque no es posible determinar, estáticamente, cuál de ellas. La tabla 2.1 muestra la aplicación de la forma SSA a un ejemplo sencillo.

$B \leftarrow A$	$B.0 \leftarrow A.0$
if ... **then**	**if** ... **then**
$C \leftarrow B$	$C.0 \leftarrow B.0$
else	**else**
$C \leftarrow 0$	$C.1 \leftarrow 0$
end if	**end if**
	$C.2 \leftarrow \Phi(C.0, C.1)$
$A \leftarrow C$	$A.1 \leftarrow C.2$

Tabla 2.1: Ejemplo de aplicación de la forma SSA

La forma SSA, soportada por las versiones actuales de GCC, es muy útil para los optimizadores, ya que elimina falsas dependencias como las dependencias por nombre, y hace explícitas las cadenas de *definiciones* y *usos*, explicitando de esta forma, las cadenas de productores y consumidores de datos.

En GCC cualquier variable usada o definida por un sentencia recibe el nombre *operador*. Cada *operador* se corresponde con una versión de una variable, que en nomenclatura de GCC, se denomina *nombre SSA*. Además, éstos pueden ser *virtuales* o no. Así pues, un *uso virtual* es un uso que ocurre en un camino de ejecución condicional (por ejemplo, en el cuerpo de un sentencia if). Estáticamente no puede garantizarse que ese uso ocurra, sin embargo, queda constancia de que puede ocurrir mediante el correspondiente

operador virtual. Análogamente, las *definiciones virtuales* son definiciones que se dan en caminos de ejecución condicionales.

Capítulo 3

Técnica propuesta

En el cuerpo de una función suficientemente compleja a menudo aparecen llamadas a función que realizan trabajos no relacionados entre sí, es decir, trabajos independientes. En estos casos, la única razón para su ejecución secuencial es su disposición en el programa original, impuesta por lenguajes imperativos cuya capacidad expresiva no permite indicar que su orden de ejecución es arbitrario.

Estas llamadas se caracterizan por el hecho de que las variables que una llamada escribe no son necesarias para la ejecución de las otras y viceversa. Así, considerando dos llamadas a función $foo()$ y $bar()$, con $foo()$ precediendo a $bar()$ en el programa original; si ambas son independientes, no se establecen dependencias de control ni dependencias de datos entre ambas. Es decir, las variables escritas por $foo()$ no son necesarias para ejecutar las instrucciones de control que causan la ejecución de $bar()$, ni son leídas por las instrucciones ejecutadas debido a la llamada a $bar()$.

El objetivo de nuestra técnica es detectar estas llamadas a función y generar código parelo que las ejecute concurrentemente. Para alcanzar este fin, se han implementado dos pasos de optimización en GCC. El primero de ellos reúne información de las funciones del programa (en todos los ficheros

que los componen). El segundo considera esta información en el conjunto del programa y paraleliza las llamadas a función independientes. Este esquema puede verse en la figura 3.1

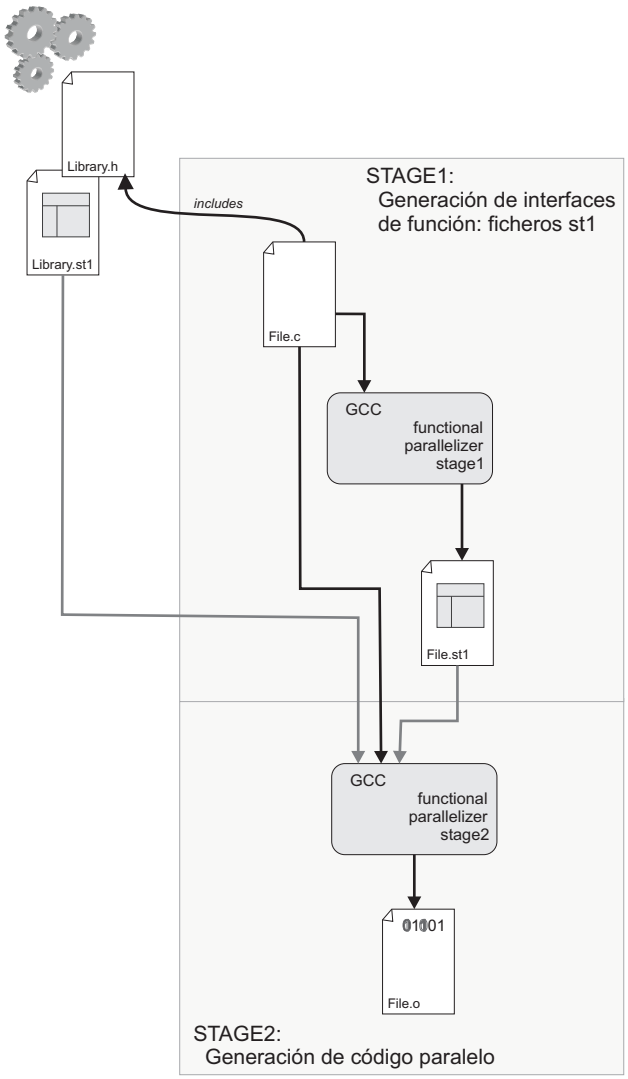


Figura 3.1: Diagrama general del proceso de análisis del programa y generación de código paralelo

3.1 Caracterización de funciones

La técnica propuesta persigue la identificación de llamadas a función cuya ejecución pueda realizarse concurrentemente. Para ello es preciso detectar las relaciones de dependencia entre las llamadas a función mediante un análisis interprocedural.

Sin embargo, realizar un análisis de dependencias interprocedural puede ser una tarea computacionalmente prohibitiva si se pretende tratar individualmente cada camino de ejecución que pueda darse en el programa. Además, un programa consta de múltiples ficheros fuente. El análisis interprocedural requiere información de lo que ocurre en cada uno de ellos. Para realizar este análisis de forma eficiente y ordenada, el primer paso de compilación aplicado por nuestra técnica recopila información de las funciones definidas en cada fichero. De cada una de ellas, se obtiene su interfaz: las variables que *usan* y *definen*, su contribución al grafo de llamadas y las asignaciones realizadas a punteros. Estos datos son almacenados en fichero. Concretamente, por cada fichero fuente *file.c* se genera un fichero *file.c.st1* conteniendo las interfaces de cada función. El formato de estos ficheros es una representación binaria optimizada para reconstruir eficientemente las estructuras de datos en GCC durante la segunda etapa de compilación. Esta información, como se verá más adelante, es esencial para paralelizar las llamadas independientes garantizando la corrección del programa. Veamos detalladamente este análisis aplicándolo al ejemplo mostrado en la tabla 3.1.

3.1.1 Identificación de usos y definiciones

El primer paso a realizar consiste en caracterizar cada función desde el punto de vista de sus interacciones con las variables del programa, es decir, identi-

```
01  int G1 = 1, G2 = 1, G3 = 1;
02
03
04  void foo(int * a))
05   {
06     *a = 2;
07    }
08
09  void bar(int * b))
10   {
11     g1 = *b;
12   }
13
14  void parallel()
15   {
16     foo (&g2);
17     bar (&g2);
18
19     foo (&g3);
20   }
21
22  void main(int argc, char * argv)
23   {
24     parallel ();
25     return g1 + g2;
26    }
```

Tabla 3.1: Código ejemplo: file.c

ficar sus *usos* y *definiciones*.

Cabe destacar que en este análisis preliminar, la función es caracterizada a partir de su definición *de forma abstracta*. De esta forma, el estado de las variables globales con las que interactúa (p. ej., sus *definiciones* previas) o el valor de sus parámetros, no son considerados. Esta información sólo podrá aproximarse al caracterizar cada instancia *concreta* de llamada a la función. Esta distinción entre la caracterización de las funciones en su definición y la caracterización de las instancias de llamada a función simplifica el análisis de dependencias, evitando analizar el cuerpo de las funciones un número innecesario de veces.

Como resultado de la caracterización de cada función se obtienen dos conjuntos que perfilan la interacción de la función con las variables del programa: un conjunto que contiene los *usos* de la función y otro que contiene las *definiciones*, es decir, las variables que lee y escribe respectivamente. Estos conjuntos reciben el nombre de *conjuntos de usos* y *conjuntos de definiciones* de la función. Para abreviar, nos referiremos a ambos como *conjuntos de la*

función.

Así, para cada sentencia perteneciente a una función f, sus *usos* son incluidos en el *conjunto de usos* de la función f, y análogamente sus *definiciones* son incluidas en el *conjunto de definiciones* de la función f. Aplicando esta sencilla regla a las funciones foo y *bar* del ejemplo de la tabla 3.1 se obtienen los *conjuntos de usos* y *definiciones* asociados a la definición de ambas funciones. El resultado puede verse en la tabla 3.2. La función foo escribe la variable $*a$ en la línea 6, por lo tanto esta expresión es añadida a su *conjunto de definiciones*. Por otra parte, la función *bar* define la variable $g1$ y usa la variable $*b$ en la línea 11; éstas serán incluídas en el *conjunto de definiciones* y el *conjunto de usos* de la función *bar* respectivamente. Análogamente se obtienen los conjuntos de las funciones *parallel* y *main*.

función	cjto. de usos	cjto. de definiciones
void foo(**int** * a))	{ }	{ *a }
void bar(**int** * b))	{ *b }	{ g1 }
void parallel()	{ }	{ }
void main(**int** argc, **char** * argv)	{ g1, g2 }	

Tabla 3.2: Usos y definiciones realizados por las funciones del programa

3.1.2 Llamadas a función

En el ejemplo anterior puede observarse que la función *parallel* realiza llamadas a las funciones foo y *bar* con distintos argumentos. Al realizar estas llamadas se producen *usos* y $definiciones$ derivados de su ejecución. Esta información es necesaria para nuestro análisis, sin embargo, en este punto de la compilación no se computarán los *usos* y $definiciones$ debidos a llamadas. En su lugar, se añadirá a la interfaz de la función las llamadas realizadas. De esta forma, cada función aporta su visión parcial del grafo de llamadas.

Analizando las funciones del programa de ejemplo se obtienen los datos mostrados en la tabla 3.3

función	subgrafo de llamadas
void foo(**int** * a))	
void bar(**int** * b))	
void parallel()	foo(&g2) bar(&g2) foo(&g3)
void main(**int** argc, **char** * argv)	parallel()

Tabla 3.3: Contribución de las funciones al grafo de llamadas

Cabe destacar que cada función referenciada en el fichero de interfaces (ya sea porque se define su interfaz o porque se realiza una llamada a dicha función) es acompañada de todos los datos necesarios para la reconstrucción de su declaración en GCC durante la segunda fase de análisis. Ésto incluye información sobre sus tipos de argumentos, valor devuelto, modificadores (p. ej., *constant*, *static*, etc.) y finalmente su localización: el fichero y número de línea en que han sido declarados. Esta última información será especialmente importante para nuestro análisis, como se verá en la siguiente sección.

3.1.3 Asignaciones a punteros

Para realizar un análisis de dependencias correcto y preciso, es necesario conocer información detallada sobre los punteros. Concretamente, se necesita conocer las variables a las que un puntero dado puede apuntar. Esta información puede formalizarse de la siguiente forma: dado un puntero P_i, el conjunto de variables potencialmente apuntadas por él viene dado por el conjunto $pointsto(P_i)$. Las reglas esenciales que definen este conjunto son las siguientes:

1. dada la asignación $P_i \leftarrow \&V$, $V \in pointsto(P_i)$
2. dada la asignación $P_i \leftarrow P_j$, $pointsto(P_j) \subset pointsto(P_i)$
3. dada la asignación $*P_i \leftarrow P_j$, $\forall P_k \in pointsto(P_i), pointsto(P_j) \subset pointsto(P_k)$

Aplicando la clausura transitiva al conjunto *pointsto*() de cada puntero, respecto a las propiedades expuestas arriba, se obtienen las variables apuntadas por él.

Si una función modifica punteros globales o contenidos en estructuras, se calcula el correspondiente conjunto *pointsto*() y se incorpora a la interfaz de la función.

3.2 Análisis interprocedural

Cuando se dispone de un fichero de interfaces por cada fuente del prorama a compilar, es posible continuar a la siguiente fase: el análisis interprocedural. Este análisis transcurre en la segunda etapa de compilación. Al invocar GCC con el argumento *-functional-parallelizer-st2*, se invoca nuestro segundo paso de análisis. La primera acción realizada es cargar el fichero de interfaces correspondiente al fichero fuente que está siendo compilado.

3.2.1 Expansión del grafo de llamadas

El conjunto de subgrafos de llamadas almacenado en el fichero de interfaces permite reconstruir el grafo de llamadas completo del fichero que está siendo compilado. Gracias a la información recolectada en el paso anterior, cada función es una caja negra que únicamente muestra los aspectos relevantes para su interacción con otras funciones.

El grafo de llamadas puede contener funciones definidas en ficheros distintos al que se está compilando, o incluso en librerías externas. El analizador detectará esta circunstancia al no encontrar una interfaz de función que concuerde con la llamada en cuestión. Sin embargo, dado que se almacena el nombre de fichero en el que se define cada una de las funciones referenciadas, el analizador puede intentar encontrar su correspondiente fichero de interfaces. En caso de encontrarlo, cargará las interfaces de las funciones necesarias para completar el grafo de llamadas. En caso contrario nos encontramos ante una función definida en una librería que no ha sido compilada con nuestro paralelizador. Aunque es deseable disponer de interfaces para todas las funciones que se utilizarán en el programa, existe una técnica que permite continuar con el análisis incluso si esta información no está disponible. Consiste en crear una interfaz artificial para representar la función. Dicha interfaz contiene un uso y una definición de la variable global *.GLOBAL*, utilizada por GCC para representar accesos a memoria *peligrosos*, que pueden interferir con cualquier otra lectura y escritura. Se trata de una solución conservadora que garantiza la corrección del ejecutable final.

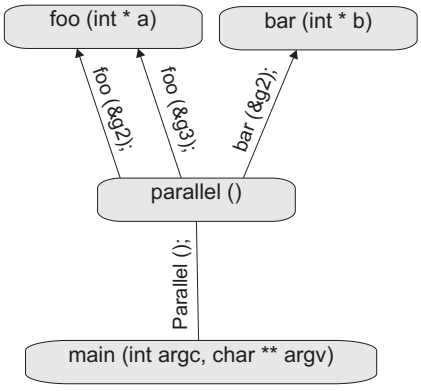


Figura 3.2: Grafo de llamadas del programa de ejemplo

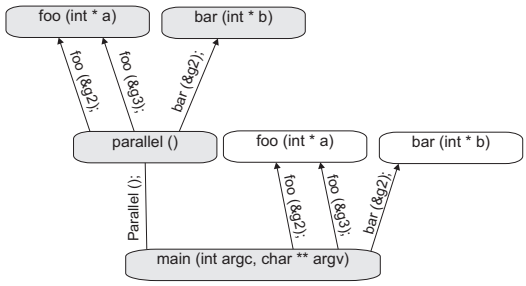


Figura 3.3: Grafo de llamadas ampliado del programa de ejemplo. Los nodos *main* y *bar* se incorporan al conjunto de funciones alcanzadas desde *main*

En este punto, el grafo de llamadas queda definido mediante un conjunto de vértices (las funciones) y aristas (las llamadas). A continuación se obtiene el *grafo de llamadas ampliado*. Partiendo del *grafo de llamadas* original, diremos que se cumple la relación $F_i callsF_j$ si existe una arista desde F_i hasta F_j. Basta con calcular la clausura transitiva de esta relación para obtener el *grafo de llamadas ampliado*. Es decir, si $F_i callsF_j$ y $F_j callsF_k$, se obtiene que $F_i callsF_k$. Una vez calculado el *grafo de llamadas ampliado* es posible consultar en tiempo constante el conjunto de funciones alcanzadas por una función dada.

Las figuras 3.2 y 3.3 muestran respectivamente el *grafo de llamadas* original y el *grafo de llamadas expandido* del programa de ejemplo.

3.2.2 Expansión de los conjuntos de usos y definiciones

Durante la primera etapa de compilación se crearon ficheros de interfaces en los que se indicaban los *usos* y *definiciones* de cada función. Sin embargo, la información completa sobre la interacción de una llamada a función con las variables del programa se obtiene al propagar los *usos* y *definiciones* de las funciones a través del grafo de llamadas. Concretamente, el conjunto de

usos expandido de una función F_i contiene la unión de su conjunto de usos inicial y los de cada función F_j, tal que existe una arista $F_i calls F_j$ en el *grafo de llamadas ampliado*. Análogamente se obtiene el conjunto de definiciones expandido.

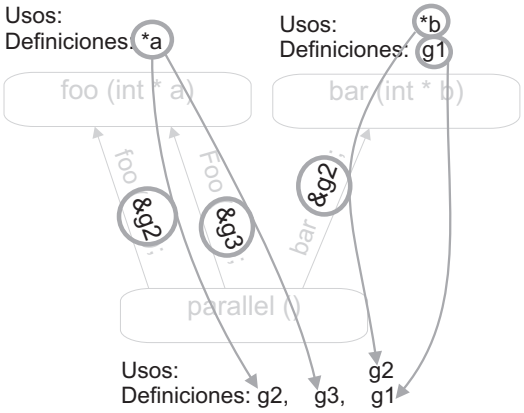


Figura 3.4: Expansión de usos y definiciones para la función *parallel*

Si prestamos atención a la función *parallel* del programa ejemplo, veremos que aunque no contiene usos ni definiciones (ver 3.2), realiza llamadas a las funciones foo y bar. Estas funciones sí que contienen variables en sus respectivos conjuntos de usos y definiciones. De acuerdo con lo expuesto en el párrafo anterior, estas variables se unen a los correspondientes conjuntos de la función. A modo de ejemplo, debido a la llamada $bar(\&g2)$ realizada por *parallel*, el conjunto de definiciones de *parallel* contiene la variable $g1$, definida por *bar*. Por otra parte, la función *bar* contiene la variable $*b$ en su conjunto de usos. Se trata del uso de una variable accedida a través de un puntero recibido como argumento. Cuando nos encontramos ante usos o definiciones de argumentos pasados por referencia, se procede a contextualizar el argumento en cuestión: se reemplaza la variable *argumento* por la variable

correspondiente variable pasada a la función en la llamada. Dado que estamos analizando la llamada $bar(\&g2)$, se debe reemplazar el argumento b por la expresión $\&g2$, obteniendo la expresión $*(\&g2)$, que una vez simplificado resulta $g2$. Por lo tanto, la variable $g2$ forma parte del conjunto de usos de la función *parallel*. Operando de esta forma para el resto llamadas de *parallel* se obtienen los usos y definiciones mostrados en la figura 3.4. Análogamente se obtienen los usos y definiciones del resto de funciones.

3.2.3 Incorporación de conjuntos de usos y definiciones en SSA

El objetivo de este paso es incorporar los usos y definiciones recopilados anteriormente en la representación SSA. Para ello se inserta en cada sentencia de llamada *operadores virtuales* de las variables expuestas a través de la llamada a función. Cabe destacar que la razón de que estos *operadores* sean virtuales es que la información proporcionada por el análisis interprocedural no garantiza que los correspondientes usos y definiciones ocurran en todos los caminos de ejecución.

GCC mantiene un potente sistema de renombrado de variables basado en la representación SSA. Una vez incorporadas las variables al sistema, cada asignación o definición de una variable crea una nueva versión de la misma, y cada lectura o uso referencia exactamente una versión de la variable. De esta forma es posible determinar con coste constante cuál es el sentencia productor de una versión de una variable dada. En ausencia de información interprocedural, GCC marca todos los sentencias de llamadas a función como posibles productores y consumidores de todas las variables globales (o de la variable artificial *.GLOBAL*), así como de los argumentos pasados por referencia. Esta conservadora práctica evita que los sentencias de llamada

a función sean reordenados poniendo en peligro la corrección del programa. Sin embargo, el análisis interprocedural descrito en esta Tesina proporciona información precisa sobre el comportamiento de las funciones.

En primer lugar, nuestro paso de análisis etiqueta cada sentencia de llamada a función, indicando a GCC que se dispone de información interprocedural para la llamada en cuestión. A continuación, se invoca el mecanismo de generación de operadores virtuales de GCC. Hemos modificado este mecanismo para que utilice la información interprocedural recopilada en etapas anteriores en lugar de aplicar las prácticas conservadoras mencionadas en el párrafo anterior. Así pues, por cada variable del conjunto de usos de la llamada a función se inserta un operador de uso virtual en el sentencia correspondiente. Análogamente se actúa con las definiciones. Finalmente, se notifica a GCC que ha habido cambios en los operadores virtuales, causando de esta forma la actualización de la representación SSA. Ésto consiste principalmente en detectar las nuevas definiciones de variables introducidas, asignándoles un número de versión disponible; y en detectar la versión correc-ta para cada uso insertado de acuerdo al alcance de las definiciones previas de las variables.

Aplicando este paso a la función *parallel* se obtiene la representación mostrada en la figura 3.5. La definición del primer argumento pasado por referencia en la llamada $foo(\&g2)$ provoca la creación de una nueva versión de la variable $g2$: $g2\_2$. Al tratarse de una definición virtual, se añade la versión anterior de la variable a la lista de usos virtuales (indicada entre los caracteres '<' y '>'). De esta forma se indica a los optimizadores que la nueva versión de la variable depende de una potencial nueva definición o del valor anterior de la variable. Dado que en GCC las versiones o *nombres*

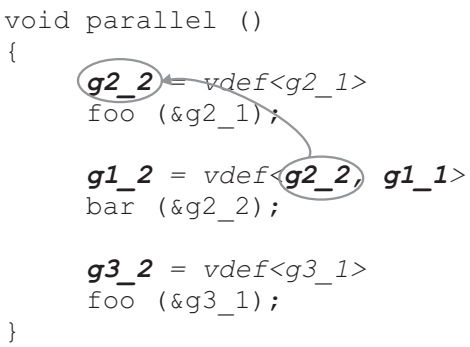


Figura 3.5: Función *parallel* en forma SSA con usos y definiciones interprocedurales incorporados. La notación empleada es la utilizada por GCC, donde las asignaciones entre operadores virtuales son anotaciones del código, en ningún caso código en sí mismo.

SSA son a todos los efectos variables locales, las variables globales constan siempre de una primera versión inicializada con el valor que toma la variable al inicio de la función. Por ello se hace referencia a la variable $g1\_1$ en lugar de simplemente $g1$.

La llamada $bar(\&g2)$ define $g1\_2$ y usa la última versión de $g2$ definida, $g2\_2$, cuyo productor es el sentencia anterior. Adicionalmente, y por las razones comentadas en el párrafo anterior, se añade un uso virtual del nombre SSA $g1\_1$. Finalmente, la llamada $foo(\&g3)$ crea una nueva versión de la variable $g3$. Puede observarse como, gracias a este mecanismo de renombrado, las cadenas de dependencias entre llamadas se representan de forma explícita y precisa.

3.3 Paralelización

Llegados a este punto, está todo dispuesto para detectar los recursos de paralelismo presentes en cada función y finalmente insertar el código necesario

para su ejecución concurrente.

3.3.1 Detección de recursos de paralelismo

En primer lugar, nuestro analizador particionará el código de cada función en bloques compuestos de sentencias. Los bloques se relacionan entre sí de acuerdo a las dependencias de datos y de control presentes en los sentencias que contienen. Se distinguen dos tipos de relaciones: las *relaciones de continencia* y las *relaciones de dependencia*. Las primeras vienen dadas por dependencias de control. Un bloque B_i contiene a otro bloque B_j si la ejecución de alguno de los sentencias en B_j depende de una condición evaluada en B_i. Por otro lado, las *relaciones de dependencia* están determinadas por dependencias de datos. Un bloque B_j depende de otro bloque B_i si alguna de las variables consumidas en B_j es producida por B_i.

Las estructura de datos que representa estos bloques es *fp_block*. Antes de comenzar con la explicación del algoritmo, cabe destacar las diferencias y similitudes entre las estructuras *basic_block* y *fp_block*. La primera es utilizada por GCC para representar un bloque básico y está presente en la representación del grafo de control de flujo. La segunda se utiliza únicamente en nuestro paso de paralelización. Es una estructura artificial que permite expresar relaciones de dependencia entre bloques de código. Pese a ser estructuras distintas mantienen una fuerte relación: un *fp_block* está compuesto por uno o más *basic_block* o uno o más *fp_block*. Además, todo *basic_block* pertence exactamente a un *fp_block*. Cada estructura *fp_block* consta de una lista de bloques que dependen de él por control (bloques contenidos) y otra de bloques que dependen de él por datos (bloques dependientes). Además, se almacena una referencia al bloque que lo contiene (contenedor) y/o el bloque del que depende (predecesor). Durante la construcción de un bloque, si se

detectan dependencias respecto a otros bloques, se anotan las correspondientes referencias pero no se actualizan las listas de los bloques en cuestión. Al finalizar la construcción, en caso de haberse detectado una dependencia se actualiza la lista de bloques dependientes del bloque correspondiente. En caso contrario, se actualiza la lista de bloques contenidos del bloque anotado como contenedor.

Para su construcción, se aplica al cuerpo de la función la siguiente rutina. Inicialmente se crea un bloque *fp_block* vacío que contendrá al resto de bloques de la función y se deposita en la cima de una pila de bloques. A continuación se procesarán secuencialmente y en orden de dominancia [1] los bloques básicos de la función que se encuetren en un mismo nivel de anidamiento[2]. Por cada bloque se iterara a través de sus sentencias y con cada uno se emprenderán las acciones descritas a continuación.

- Si el sentencia es iniciador de bloque, es decir, el primer sentencia de un nivel de anidamiento, un sentencia de llamada a función o una construcción de control (p. ej., if, *switch*, etc.): se finaliza el bloque en la cima de la pila y se extrae de la misma, en caso de compartir el nivel actual de anidamiento. Se crea un nuevo bloque que pasa a

[1]Se dice que un bloque A domina a otro bloque B si todos los caminos de ejecución que llevan a B pasan por A. Además A es el dominador principal de B si no hay otro bloque entre A y B que domina a B. Por definición todo bloque se domina a sí mismo. Si un conjunto de bloques $A_0, A_1..A_n$ están ordenados en orden de domiancia, se tiene que ningún A_j domina a otro $A_i, j>i$

[2]Dos bloques A y B, tal que A domina a B, se encuentran en un mismo nivel de anidamiento si existe una sucesión de saltos directos (incondicionales) que llevan desde A hasta B. A modo de ejemplo, en una función sencilla que contenga una llamada seguida en un primer bloque, seguida de una estructura if en otro bloque, tanto el bloque con la llamada como el bloque con el sentencia if compartirían el mismo nivel, mientras que los bloques destino de las ramas *then* y *else* estarían en un nivel más profundo.

ocupar la cima de la pila. El bloque anotado como contenedor es el situado inmediatamente debajo en la pila.

- Si el sentencia es una construcción de control: se itera a través de cada rama de ejecución condicional (p. ej., ramas *then* y *else* de la estructura if, bloques *case* de la estructura *switch*, etc.). Las instrucciones que éstas contienen se encuentran en un nivel de anidamiento más profundo, así que a cada rama se le aplica el mismo procedimiento que el aplicado al cuerpo de la función.

- El sentencia actual es añadido al bloque en la cima de la pila.

- Por cada uso realizado por el sentencia actual se consulta el sentencia que lo define. Si el bloque al que pertenece es más cercano que el predecesor del bloque actual, pasa a ser el nuevo predecesor.

Tras analizar todos los sentencias se finaliza y desapila el bloque actual y se vuelve al nivel de anidamiento anterior. En caso de quedar un único bloque en la pila, el bloque principal de la función, se da por terminada la construcción de bloques.

Para entender el funcionamiento del algoritmo encargado de particionar el código lo aplicaremos a la función *parallel* de nuestro ejemplo. En primer lugar, veamos en la figura 3.6 el aspecto de la función tal y como la recibe nuestro analizador. Puede observarse la presencia de dos bloques básicos artificiales creados por GCC: uno representa la entrada principal de la función y otro la salida. Estos bloques básicos existen siempre y no contienen sentencias.

La construcción comienza creando un bloque vacío (bloque 0) y colocándolo en la cima de la pila. Se avanza directamente hasta el bloque

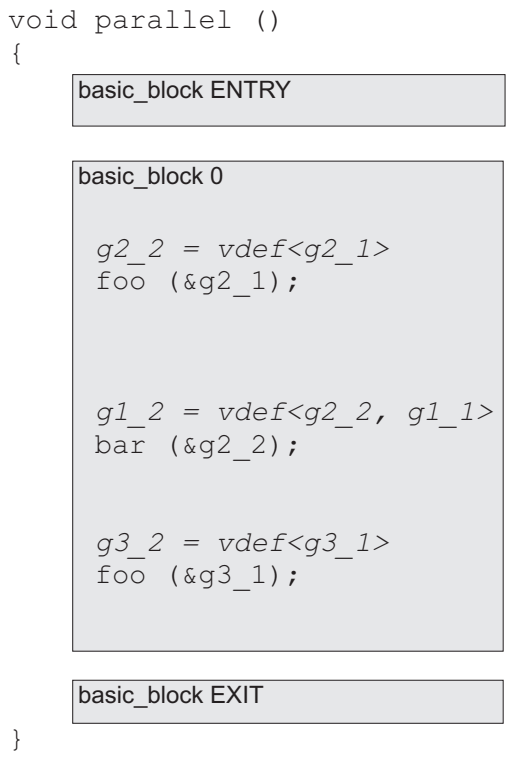


Figura 3.6: Representación de la función *parallel* en GCC

básico 0, ignorando el bloque de entrada artificial.

A continuación se procesa el primer sentencia. Es un sentencia iniciador de bloque (tanto por ser llamada a función como por ser el primer sentencia de su nivel), así que se crea un nuevo bloque. Puesto que el bloque 0 está en la cima de la pila y pertenece a un nivel de anidamiento menos profundo, se anota como predecesor del nuevo bloque. El nuevo bloque se coloca en la cima de la pila. El sentencia de llamada pasa a formar parte del bloque, y dado que ninguno de sus usos es definido por ningún sentencia no se establecen más relaciones.

Se avanza hacia el siguiente sentencia. De nuevo nos encontramos con un iniciador de bloque: una llamada a función. Dado que el bloque en la

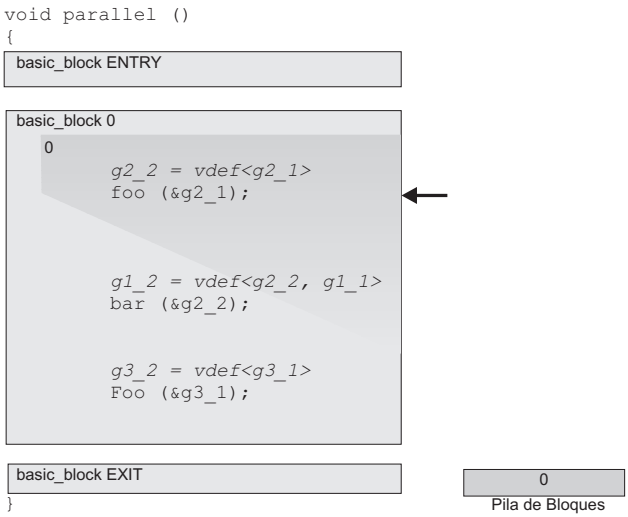


Figura 3.7: Construcción de bloques: paso 1

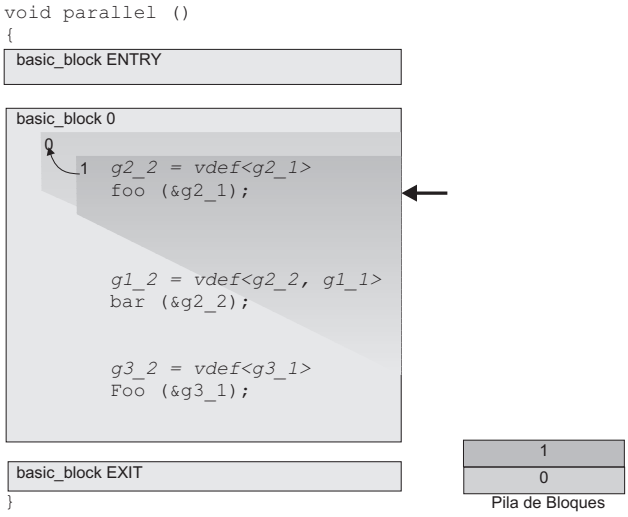


Figura 3.8: Construcción de bloques: paso 2

cima de la pila está en el mismo nivel que el nuevo bloque se finaliza el anterior. Puesto que el bloque 1 no tenía ningún predecesor establecido, pero sí un contenedor, se inserta en la lista de bloques contenidos del bloque 0. Seguidamente se crea el bloque 2, y se marca bloque 0 como contenedor. El sentencia de llamada pasa a formar parte del bloque 2. Uno de sus usos,

$G2\_2$, es definido por un sentencia que pertence al bloque 1, por lo tanto se marca este bloque como predecesor.

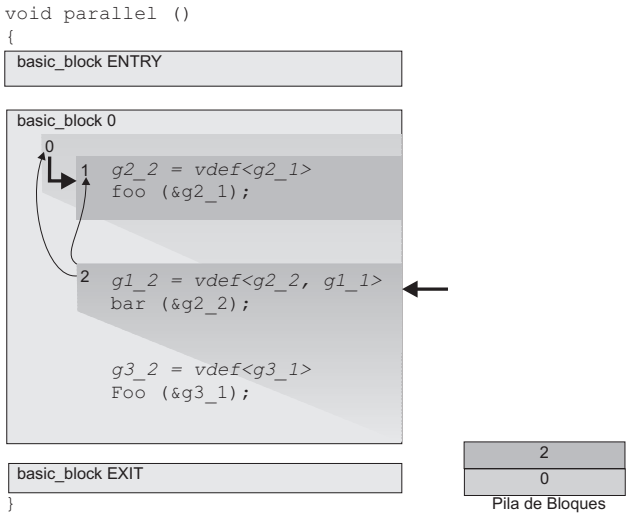


Figura 3.9: Construcción de bloques: paso 3

Seguimos avanzando hacia el siguiente sentencia. Una vez más se trata de un iniciador de bloque. Se finaliza el bloque 2. Puesto que tiene marcado como predecesor el bloque 1, es insertado en su lista de bloques dependientes. Se crea el bloque 3 y se establece el bloque 0 como contenedor. Tras insertar el sentencia en el bloque se analizan su único uso, $G3\_2$. Dado que ninguno de los sentencias analizados lo define, no se establece ningún predecesor para este bloque.

El sentencia anterior era el último del nivel actual de anidamiento, así que se cierra y desapila el bloque actual. En ausencia de predecesor, pasa a formar parte de la lista de bloques contenidos por el bloque 0. Dado que no hay sentencias para analizar en niveles de anidamiento más externos, se finaliza el bloque principal y concluye el algoritmo.

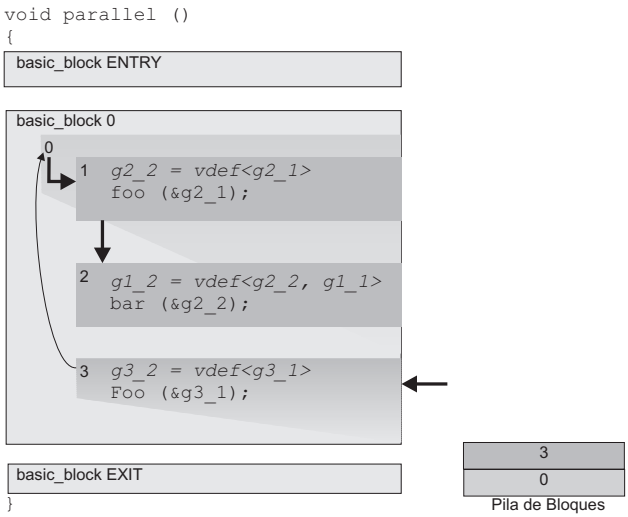


Figura 3.10: Construcción de bloques: paso 4

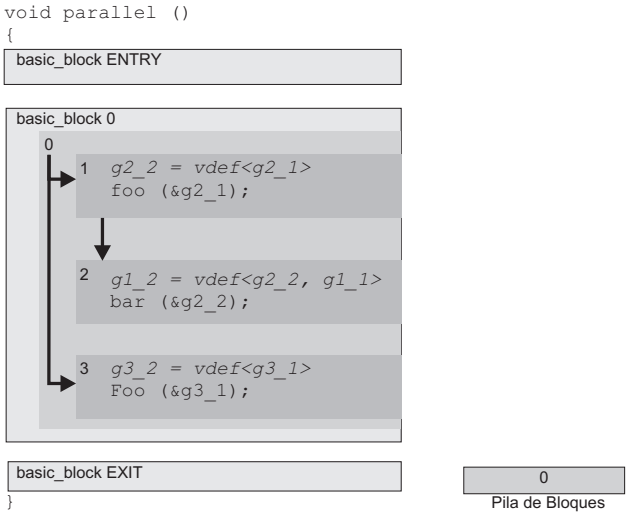


Figura 3.11: Construcción de bloques: paso 5

Como resultado se obtiene un árbol de bloques en el que el bloque 0 contiene a los bloques 1 y 3, y el bloque 1 tiene como predecesor al bloque 2. Esto significa que al inicio del bloque 0 podría comenzar la ejecución de los bloques 1 y 3. Por otra parte, tras la ejecución del bloque 1 puede comenzar a ejecutarse el bloque 2.

Antes de continuar con la paralelización del código se simplifica el árbol fusionando pares de bloques contiguos A y B, tales que A es predecesor de B y B es su único sucesor. Finalmente, se particionan los bloques básicos de grafo de control de flujo de forma que exista la correspondencia entre bloques básicos y bloques de dependencia expuesta al inicio de este apartado. La estructura resultante es la mostrada en la figura 3.12.

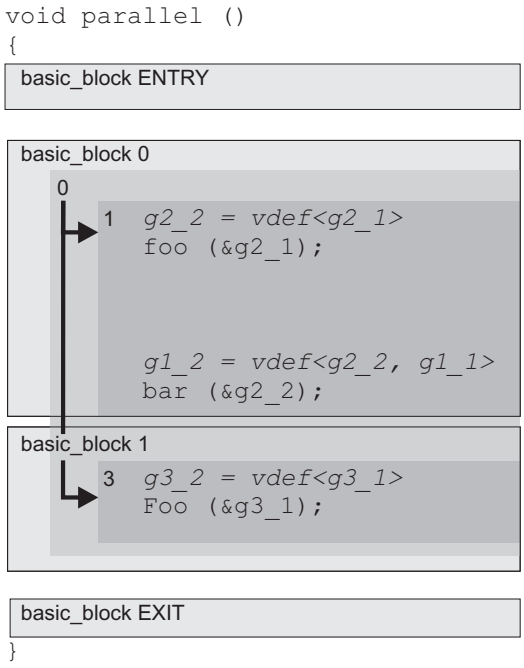


Figura 3.12: Construcción de bloques: resultado final

3.3.2 Inserción de primitivas OpenMP

Tras particionar el código en bloques se dispone de información precisa sobre los fragmentos de código que pueden ser ejecutados concurrentemente. El siguiente paso es insertar el código necesario para llevarlo a término. El paradigma de programación paralela más adecuado para nuestra técnica es

memoria compartida. La principal ventaja que encontramos con este paradigma es la posibilidad de explotar paralelismo de baja granularidad sin ser gravemente penalizados por la sobrecarga. Esto es así debido a la baja latencia de las operaciones de creación, sincronización y comunicación entre hilos. De entre las distintas plataformas disponibles para la creación de programas paralelos con memoria compartida hemos optado por OpenMP.

OpenMP oferta una amplia variedad de primitivas para la paralelización de código. Dada la naturaleza de nuestra técnica hemos seleccionado la primitiva *parallel sections*. Su sintaxis es la mostrada en la figura 3.13. La semántica asociada es la siguiente: cuando la ejecución del programa alcanza la directiva *omp sections*, el código contenido por los distintos bloques *section* es distribuido entre los hilos disponibles.

```
#pragma omp sections [clause[[,] clause] ...] new-line
  {
    [#pragma omp section new-line]
      structured-block
    [#pragma omp section new-line
      structured-block ]
    ...
  }
```

Figura 3.13: Directiva *ompsections*. Sintaxis

La forma en que utilizaremos esta directiva en nuestra técnica es la siguiente. Cuando un bloque contenga más de un bloque, se insertará a su inicio la directiva *omp sections*. Cada uno de los bloques contenidos estará precedido por una directiva *omp section*. Respecto a las relaciones de dependencia, cuando un bloque tenga más de un bloque en su lista de dependientes, se insertará una directiva *omp sections* al final del bloque, y acontinuación se dispondrán los bloques dependientes precedidos por la directiva *omp section*. Apliquémoslo a la función *parallel* de nuestro ejemplo. Dado que el bloque

0 contiene dos bloques, se inserta a su inicio la directiva *omp sections*. Los bloques contenidos constituyen las correspondientes secciones. El resultado puede verse a continuación:

```
void parallel ()
{
  #pragma omp sections

    #pragma omp section
    {
       g2_2 = vdef<g2_1>
       foo (&g2_1);

       g1_2 = vdef<g2_2, g1_1>
       bar (&g2_2);
    }

    #Pragma omp section
    {
       g3_2 = vdef<g3_1>
       Foo (&g3_1);
    }
```

Figura 3.14: Función *parallel* paralelizada mediante OpenMP

Sin embargo, aunque esta sintaxis es la proporcionada por el API de OpenMP para la programación de aplicaciones paralelas, nuestra técnica de paralelización actúa durante la compilación. En lugar de insertar directivas de OpenMP es necesario generar código paralelo tal y como GCC traduce las construcciones OpenMP. La implementación GNU de OpenMP (GOMP) proporciona un conjunto de funciones implementadas en la propia librería del compilador (*builtins*) para la generación de código paralelo utilizando OpenMP. La traducción de una construcción *omp sections* en GCC es la mostrada en la figura 3.15. Veamos en qué consiste. La función *GOMP_sections_start* recibe como argumento el número de secciones totales y devuelve a cada hilo un entero que identifica la primera sección que deberá

ejecutar. A continuación cada hilo salta, mediante la estructura *switch*, a la etiqueta correspondiente que le ha sido asignada y ejecuta el código perteneciente a la sección en cuestión. Al finalizar salta a *L_continue*, donde se produce la llamada a *GOMP_sections_next*. Ésta devolverá un nuevo identificador de sección si aún no se han ejecutado todas (en caso de haber más secciones que hilos), o 0 en caso contrario. Supongamos que ya se han ejecutado todas las secciones. Todos los hilos saltarán a la etiqueta *L_switch*, procesarán el *switch* y serán redirigido a la etiqueta 0, saltando finalmente a la etiqueta *L_reduction*. En este punto se produce la llamada a *GOMP_sections_end*, que finaliza la ejecución de la estructura *omp sections*.

```
        .section = GOMP_sections_start (N. SECCIONES);
L_switch:
        switch (.section)
        {
                case 0:
                        goto L_reduction;
                case 1:
                        SECCIÓN 1
                        goto L_continue;
                case 2:
                        SECCIÓN 2
                        goto L_continue;
                ...
                default:
                        abort ();
        }
L_continue:
        .section = GOMP_sections_next ();
        goto L_switch;
L_reduction:
        GOMP_sections_end ();
```

Figura 3.15: Traducción de la construcción *omp sections* en GCC

Por lo tanto, nuestra técnica en lugar de insertar directivas *OpenMP* transforma los recursos de paralelismo disponible en una estructura como la anterior. Para ello, los bloques básicos previos y posteriores a las secciones son divididos a fin de insertar las etiquetas y saltos adecuadamente[3]. También

[3]Las etiquetas deben situarse al inicio de un bloque básico, garantizando que ningún

se insertan las llamadas a las funciones $GOMP$ y se crea la variable artificial *.section* para almacenar el identificador de sección asignado a cada hilo. Tras aplicar esto a nuestra función de ejemplo, obtenemos el código mostrado en la figura 3.16. Este código es entregado a los pasos posteriores del compilador para ser optimizado y finalmente convertido en las instrucciones máquina que constituyen el programa paralelo generado mediante nuestra técnica.

salto tiene por destino un sentencia en mitad de un bloque básico. Además, todos los saltos deben ocurrir al final de un bloque básico

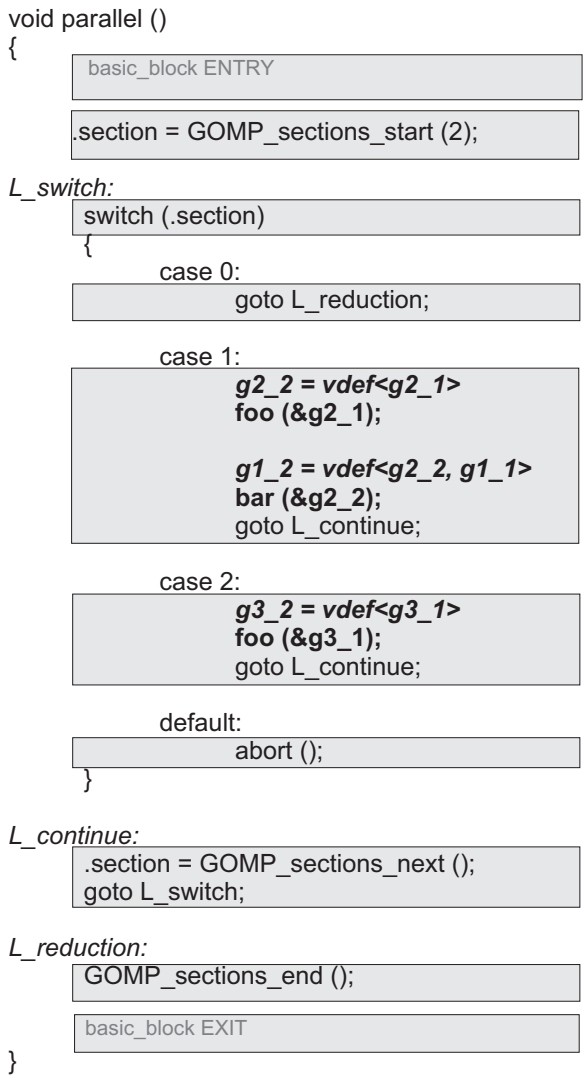


Figura 3.16: Código paralelo generado por nuestra técnica para la función $parallel$

Capítulo 4

Evaluación de la técnica

La implementación en GCC de la técnica descrita en esta Tesina ha sido el fruto de un año de aprendizaje y desarrollo. En la actualidad somos capaces de producir código paralelo real partiendo de un ejecutable secuencial. Sin embargo, aún falta concluir algunos aspectos críticos relacionados con el tratamiento de punteros que permitan aplicar nuestra técnica de forma generalizada. Por esta razón nuestra técnica no ha sido aplicada a *benchmarks* comerciales como las Spec2006, obteniendo medidas de prestaciones reales. Sin embargo, con anterioridad a la implementación en GCC evaluamos y publicamos [23][24] su potencial utilizando métodos de simulación para la medida de prestaciones. Los alentadores resultados que obtuvimos nos condujeron al desarrollo de esta implementación, con la finalidad de poder evaluar de forma precisa y flexible nuestra técnica.

El simulador desarrollado para la evaluación preliminar de la técnica toma como entrada un informe de dependencias resultantes de aplicar nuestro análisis a los programas sujetos a estudio y una traza de su ejecución secuencial. Con esta información se simula la ejecución paralela del programa, solapando el tiempo de ejecución de las llamadas independientes. La relación entre el tiempo de ejecución secuencial y la estimación del tiempo de

ejecución paralelo es un indicador de la capacidad de nuestra técnica para detectar paralelismo. En las siguientes secciones se explicará el proceso de obtención de las trazas y las bases de la simulación. Finalmente se ofrecen los resultados obtenidos.

4.1 Método de simulación

Para poder simular la ejecución paralela de los programas secuenciales tras aplicar la técnica propuesta es necesario conocer la cronología de las llamadas (el orden y duración de las mismas) en la ejecución secuencial. A fin de llevar a cabo tal propósito, el compilador inserta instrucciones de instrumentación en el ejecutable secuencial.

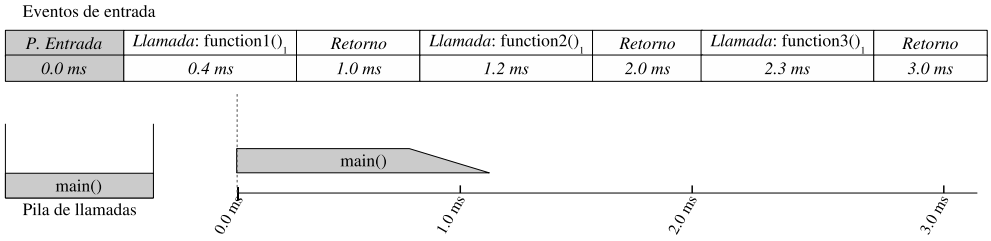
Para ello, un paso introducido en GCC detecta los sentencias de llamada a función e inserta antes y después de los mismos las instrucciones necesarias para identificar la llamada realizada y el instante en el que tiene lugar el evento. Para minimizar la sobrecarga debida a la propia instrumentación se realiza una corrección de los tiempos medidos, restando el tiempo empleado por estas rutinas.

Estas trazas permiten alimentar un simulador guiado por eventos. Se distinguen cuatro tipos de eventos. El evento *punto de entrada*, correspondiente al inicio de la función *main*; el evento *llamada*, generado por cada llamada a función; el evento *retorno*, generado por el retorno de cada llamada; y el evento eof, generado al alcanzar el fin del fichero de eventos. Cada evento tiene como atributo el instante en el que ocurre el evento. Además, los eventos de tipo llamada poseen el atributo *función*, que contiene el identificador de la función invocada y permite acceder a su tabla de dependencias.

Durante la simulación se mantiene una pila que refleja la pila de llamadas

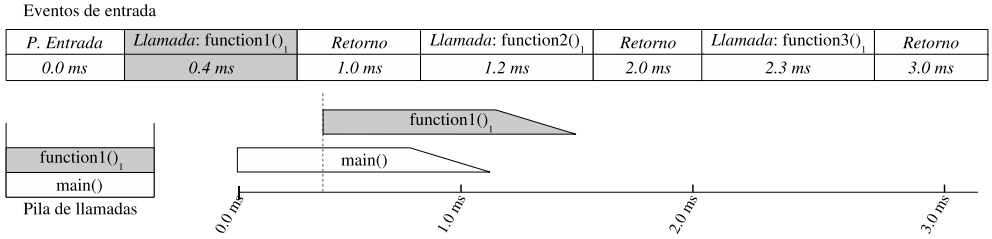
en la ejecución secuencial. Por cada llamada apilada se almacena una estructura llamada *rt_call* y un puntero a la tabla de dependencias de la función correspondiente a la llamada. La estructura *rt_call* almacena información descriptiva de la llamada (función invocada, función invocante e identificación de la llamada), información cronológica relativa a la ejecución secuencial de la llamada (instante de inicio y de fin), e información cronológica relativa a la simulación paralela (inicio y duración en la ejecución paralela, y otros datos requeridos para calcular esta información). Además almacena una lista que contiene las llamadas (punteros a estructuras *rt_call*) realizadas por ella.

Con la llegada del evento *punto de entrada*, correspondiente a la función *main*, la llamada *main* pasa a ser el *contexto activo* de la simulación. Se crea un nodo *rt_call* para *main*, con *cero* como instante de inicio secuencial y paralelo, y se obtiene su correspondiente tabla de dependencias. Finalmente se lee el siguiente evento.

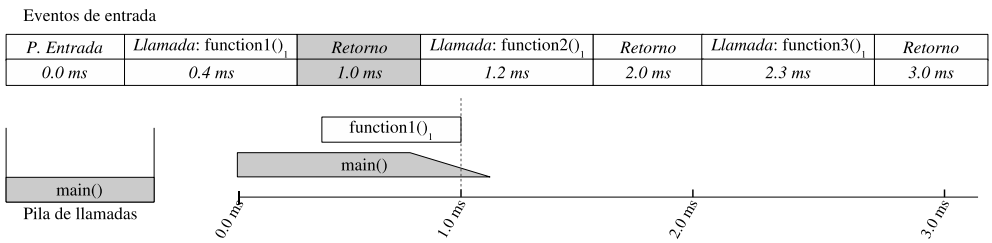


Al llegar un evento *llamada* se crea un nuevo nodo para la nueva llamada, que es almacenado en la lista de llamadas del *contexto activo* (*main*). Por tratarse de la primera llamada de *main*, su instante de inicio paralelo es el tiempo transcurrido desde el inicio de *main* hasta el instante en el que ocurre la llamada sumado al instante de inicio paralelo de *main*. Esto es así porque se asume una parte secuencial al inicio de cada función hasta producirse su primera llamada. Este fragmento de tiempo secuencial se almacena en el campo *tchain_start* de la estructura *rt_call* del *contexto activo*, ya que

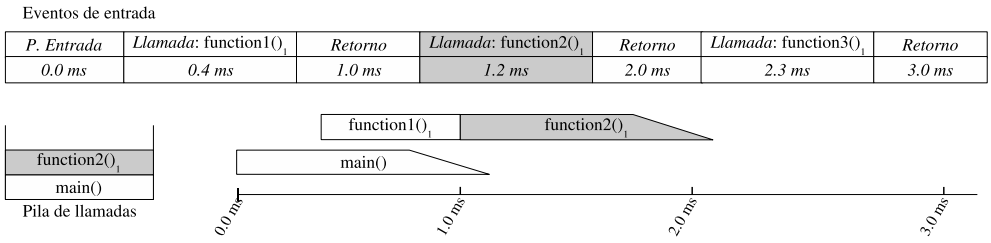
será requerido de nuevo si aparecen llamadas que no dependan de ninguna anterior. Finalmente, el *contexto activo* actual es apilado (se apila tanto su estructura *rt_call* como su tabla de dependencias) y la llamada recién llegada pasa a ser el *contexto activo*, así que la tabla de dependencias vigente pasa a ser la de esta función.



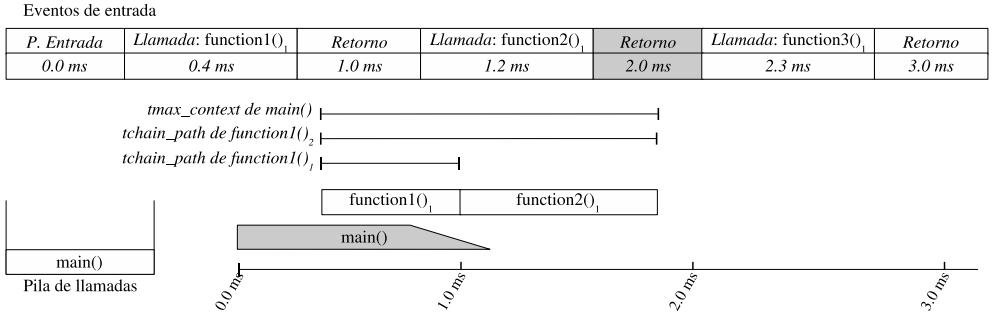
Si el siguiente evento en llegar es un evento *retorno* (el código de la llamada anterior no realiza llamadas), se procede a calcular el tiempo paralelo del *contexto activo* antes de ser eliminado. En este caso, su duración paralela coincide con la secuencial (dado que no ha realizado llamadas cuya ejecución pueda solaparse). En este punto se libera la lista de llamadas de la función que retorna (que estaba vacía), y se actualiza el tiempo paralelo de la función que la ha invocado (*main*), que se encuentra en la cima de la pila. En concreto se actualiza la fracción del tiempo de ejecución de la función debida a la ejecución solapada de llamadas (campo *tmax_context* de *rt_call*). En este caso, dado que sólo realiza una llamada, este tiempo coincide con la duración de la llamada actual. Finalmente, se desapila la llamada anterior (su estructura *rt_call* y su tabla de dependencias), con lo que el *contexto activo* vuelve a ser *main*, y se procesa el siguiente evento.



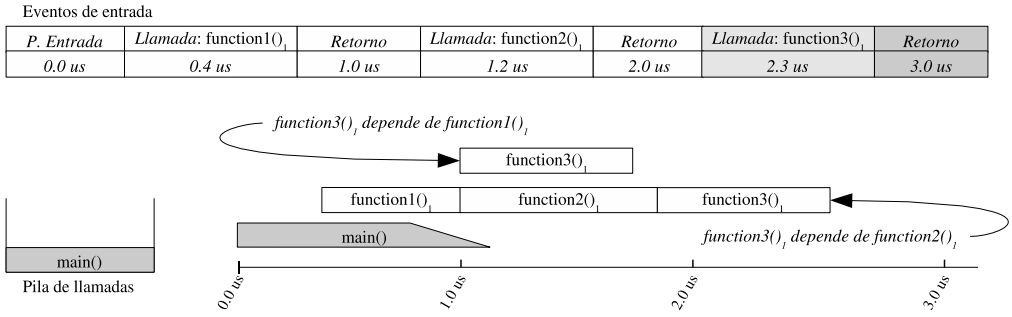
Supongamos que el siguiente evento es otra llamada. En este caso para calcular su instante de inicio se debe comprobar si depende de alguna llamada anterior (dado que la lista de llamadas del *contexto activo* no está vacía) consultando en la tabla de dependencias la relación entre la llamada actual y la previa. Si es así, se encadenará a la anterior (mediante un puntero desde la última llamada a su predecesor y no al revés) y el instante de inicio de la nueva llamada se establecerá como el instante de retorno de su predecesora. En caso de haber sido independiente de la llamada anterior, su instante de inicio coincide con el de la primera llamada realizada, que fue almacenado en el campo *tchain_start*. Como en el caso anterior, la nueva llamada pasa a ser el *contexto activo* y el anterior se apila de nuevo.



Con el evento *retorno* de esta llamada se actualiza el campo *tmax_context* de la llamada invocante. Como había un predecesor para la llamada *contexto activo*, *tmax_context* debe ser actualizado considerando la duración de la llamada actual y las anteriores. Para tal propósito, cada llamada almacena en el campo *tchain_path* de *rt_call* la duración de la cadena de llamadas a la que pertenece. De esta forma se evita recorrer la la cadena de predecesores de una llamada para determinar su duración. Por lo tanto, el valor actualizado de *tmax_context* de la llamada invocante es el máximo entre su valor actual (dado que este valor podría venir dado por una cadena de llamadas independiente a la que se está analizando) y el valor de *tchain_path* de la llamada que se está procesando.



Una nueva llamada tras el *retorno* de la actual comprobaría si depende de alguna de las previas, considerando todas las contenidas en la lista de llamadas de la función invocante. Dado que puede ser cualquiera de ellas (o ninguna), la cadena de llamadas dependientes es en realidad un árbol, y la componente *tmax_context* de la llamada de la que cuelga el árbol contiene su camino más largo (medido en tiempo).



El último evento que cabe esperar es el *retorno* de *main* o eof. Si se alcanza eof quedando llamadas en la pila (posiblemente porque se ha interrumpido la ejecución del programa secuencial), se planificarán eventos *retorno* artificiales hasta vaciar la pila.

Con el *retorno* de una función que ha realizado llamadas, como es el caso de *main*, se debe actualizar su duración paralela considerando el posible solapamiento de sus llamadas. Así, la duración paralela de estas funciones es la suma del tiempo secuencial previo a la primera llamada (*tchain_start*) y

el camino más largo de sus árboles de llamadas (puede haber varios en tanto que pueden haber varias llamadas independientes), que ha sido calculado a lo largo de la simulación y se almacena en *tmax_context*.

Sin embargo, el tiempo que hemos calculado es incompleto, dado que no considera el tiempo transcurrido entre llamadas. Este tiempo es la contribución de los sentencias del cuerpo de la función invocante. En una ejecución paralela real habrían sido reordenados en torno a las llamadas de las que dependen, solapándose la ejecución de los mismos en la medida en que se solapan las llamadas cuyas definiciones consumen. Sin embargo, simular esta característica es impracticable (el tiempo de ejecución del simulador sería excesivo), por lo tanto se simula un modelo pesimista y otro optimista que acotarán superior e inferiormente la mejora obtenida mediante la técnica propuesta. Estos modelos se estudian a continuación.

4.1.1 Modelos de aproximación para la simulación de sentencias básicos

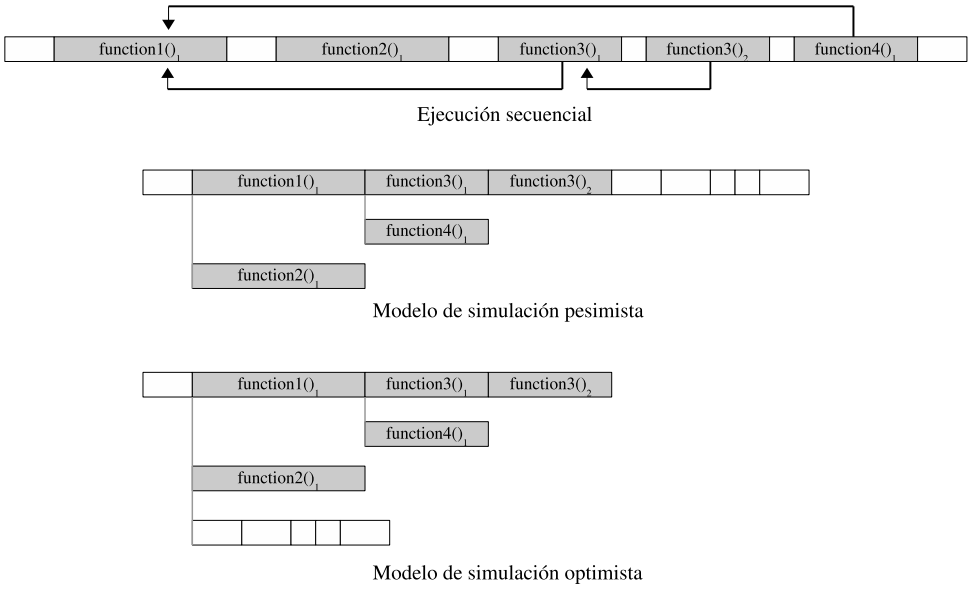
El tiempo empleado por la función para la ejecución de sentencias básicos (es decir, aquellos que no son llamadas a función), debe ser considerado en el cálculo del tiempo final de cada función. Dado que durante la simulación no es factible determinar a qué tarea (secuencia de llamadas) pertenecen estos sentencias, no es posible calcular de forma precisa su contribución al tiempo de la ejecución paralela del programa. Por lo tanto dicha contribución se acota entre un modelo pesimista y otro optimista.

El modelo pesimista asume que todos los sentencias básicos de una función se ejecutan secuencialmente en el camino más largo del árbol de llamadas. De esta forma, su contribución temporal se traduce en la suma de esta fracción del tiempo secuencial al tiempo paralelo que calculamos en el apartado an-

terior (tiempo hasta el inicio de la primera llamada más longitud del camino más largo del árbol). Una forma sencilla de calcular esta fracción es restar al tiempo de ejecución secuencial de la función el tiempo de ejecución secuencial de las llamadas que realiza y el tiempo hasta la primera llamada, que es asumido secuencial bajo cualquier modelo.

El modelo optimista asume que la ejecución de estos sentencias se solapa completamente con la ejecución de las llamadas invocadas por la función, de forma que su cálculo final del tiempo de la ejecución paralela de la función consiste en sumar al tiempo de inicio de la primera llamada, el máximo entre el camino más largo del árbol de llamadas y la fracción del tiempo de ejecución secuencial empleada en la ejecución de sentencias de la función.

La siguiente figura muestra un ejemplo de ambos modelos de simulación para la ejecución secuencial de una función que contiene 5 llamadas cuyas dependencias son las indicadas mediante flechas.



La duración de la ejecución paralela real (potencial, dado que no consideramos sobrecarga por sincronización y creación de tareas), oscilará entre

el tiempo de ejecución del modelo pesimista y el del optimista.

4.2 Resultados

La aceleración o *speedup* resultante de contrastar el tiempo de ejecución paralelo respecto al secuencial es un buen indicador de la capacidad de detección de paralelismo de la técnica implementada.

Nuestra técnica ha sido aplicada a un subconjunto de los benchmarks en C y Fortran del paquete Spec2006, y algunos benchmarks del paquete Mediabench. Por cada benchmark se ha realizado dos simulaciones, una bajo el modelo de simulación pesimista y otra bajo el modelo optimista, obteniendo una cota inferior y superior del *speedup* alcanzable respectivamente. La figura 4.1 muestra los resultados de este análisis.

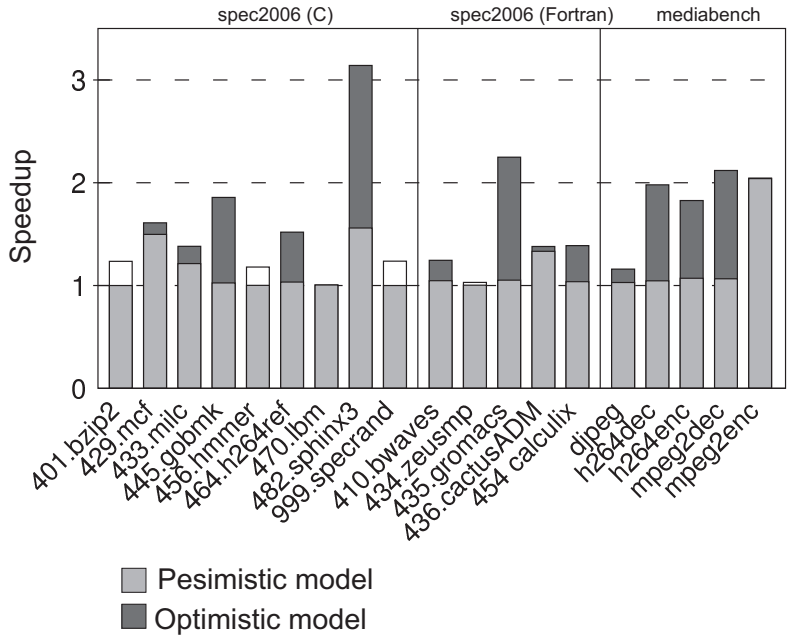


Figura 4.1: Speedup simulado obtenido al explotar *function level parallelism*

Antes de comentar los resultados cabe mencionar cómo deben ser interpretados. Dadas las especificaciones del modelo de ejecución pesimista, al obtener un *speedup* de *uno* en algún benchmark, es decir, su tiempo de ejecución secuencial coincide con el paralelo, debe entenderse que no se ha podido solapar la ejecución de ninguna función. Dado que ambos modelos de simulación parten de las mismas tablas de dependencia, en estos casos tampoco es posible solapar la ejecución de ninguna función bajo el modelo optimista. Sin embargo, dado que el modelo optimista asume que los sentencias básicos del cuerpo de las funciones se solapan con las llamadas que efectúan, aparece un *speedup* artificial. En estos casos, este *speedup* es siempre inalcanzable, dado que ninguna función se ejecuta como tarea separada y por lo tanto no hay sentencias básicos dependientes de estas llamadas que se ejecuten en su misma tarea. Los *speedups* correspondientes a estos casos han sido mostrados en blanco para no inducir a confusión.

De acuerdo a los resultados, los mejores beneficios aparecen en mpeg2enc donde el tiempo de ejecución se reduce por un factor de 2,1. En términos generales, los benchmarks Spec2006 muestran los mejores resultados al aplicar la técnica (p. ej., 429.mcf, 482.sphinx3 y 445.gobmk).

Capítulo 5

Conclusiones

Este proyecto ha introducido el Paralelismo a Nivel de Función o FLP (siglas del inglés *Function Level Parallelism*) como propuesta para explotar paralelismo a nivel de tarea. Se trata de una técnica basada en compilación que realiza un análisis estático del código para identificar qué llamadas a función pueden lanzarse concurrentemente. Esta técnica ha sido puesta en práctica en un compilador real, llegando a generar código paralelo de forma completamente automatizada.

Además, a falta de resultados definitivos, se ha desarrollado un simulador para poder estimar el potencial de nuestra propuesta. Los primeros resultados experimentales muestran que a través de la explotación de FLP mediante las técnicas expuestas, en algunas aplicaciones, se puede alcanzar una mejora de prestaciones que llega hasta el 200%.

FLP abre un amplio horizonte de posibilidades. Son muchas las mejoras que puede realizarse para continuar mejorando las prestaciones conseguidas por la técnica. Hasta el momento se ha explotado exclusivamente el paralelismo estáticamente garantizable. Sin embargo, en el futuro será posible extender nuestra técnica a medidas dinámicas que analicen dependencias en tiempo de ejecución, basándose en los valores tomados por los punteros y en

los caminos de ejecución tomados en cada estructura condicional.

También resulta de especial interés extender la aplicación de la técnica a lenguajes orientadas a objetos. Pensamos que estos lenguajes de programación manifiestan un elevado FLP intrínseco, debido a las propiedades del objeto, tales como la encapsulación o la comunicación por paso de mensajes.

Por otra parte, así como actualmente la función se considera atómica, y las llamadas dependientes se sincronizan para ejecutarse secuencialmente, un paso futuro será romper esta atomicidad y reducir las esperas, haciendo que la función dependiente espere únicamente hasta que los datos de los que depende pasen a estar disponibles.

Finalmente, queremos remarcar que la técnica propuesta es ortogonal con algunas técnicas existentes. En este sentido pensamos que el análisis conjunto de nuestra técnica junto con técnicas de paralelización de blucles puede dar lugar a importantes recursos de paralelismo a nivel de tarea con los que alimentar las actuales arquitecturas multinúcleo.

Apéndice A

Gramática GIMPLE

function : FUNCTION_DECL
DECL_SAVED_TREE → **compound-stmt**

compound-stmt : STATEMENT_LIST
members → **stmt**

stmt : **block**
| **if-stmt**
| **switch-stmt**
| **goto-stmt**
| **return-stmt**
| **resx-stmt**
| **label-stmt**
| **try-stmt**
| **modify-stmt**
| **call-stmt**

block : BIND_EXPR
BIND_EXPR_VARS → chain of DECLs
BIND_EXPR_BLOCK → BLOCK
BIND_EXPR_BODY → **compound-stmt**

if-stmt : COND_EXPR
op0 → **condition**
op1 → **compound-stmt**
op2 → **compound-stmt**

switch-stmt : SWITCH_EXPR
op0 → **val**
op1 → NULL
op2 → TREE_VEC of CASE_LABEL_EXPRs

goto-stmt : GOTO_EXPR
op0 → LABEL_DECL — **val**

return-stmt : RETURN_EXPR
op0 → **return-value**

return-value : NULL
| RESULT_DECL
| MODIFY_EXPR
op0 → RESULT_DECL
op1 → **lhs**

resx-stmt : RESX_EXPR

label-stmt : LABEL_EXPR
op0 → LABEL_DECL

try-stmt : TRY_CATCH_EXPR
op0 → **compound-stmt**
op1 → **handler**
| TRY_FINALLY_EXPR
op0 → **compound-stmt**

op1 → **compound-stmt**

handler : **catch-seq**
| EH_FILTER_EXPR
| **compound-stmt**

catch-seq : STATEMENT_LIST
members → CATCH_EXPR

modify-stmt : MODIFY_EXPR
op0 → **lhs**
op1 → **rhs**

call-stmt : CALL_EXPR
op0 → **val** — OBJ_TYPE_REF
op1 → **call-arg-list**

call-arg-list : TREE_LIST
members → **lhs** — CONST

addr-expr-arg : ID
| **compref**

addressable : **addr-expr-arg**
| **indirectref**

with-size-arg : **addressable**
| **call-stmt**

indirectref : INDIRECT_REF
op0 → **val**

lhs : **addressable**
| **bitfieldref**

| WITH_SIZE_EXPR
op0 → **with-size-arg**
op1 → **val**

min-lval : ID
| **indirectref**

bitfieldref : BIT_FIELD_REF
op0 → **inner-compref**
op1 → CONST
op2 → **var**

compref : **inner-compref**
| TARGET_MEM_REF
op0 → ID
op1 → **val**
op2 → **val**
op3 → CONST
op4 → CONST
| REALPART_EXPR
op0 → **inner-compref**
| IMAGPART_EXPR
op0 → **inner-compref**

inner-compref : **min-lval**
| COMPONENT_REF
op0 → **inner-compref**
op1 → FIELD_DECL
op2 → **val**
| ARRAY_REF
op0 → **inner-compref**
op1 → **val**
op2 → **val**
op3 → **val**

| ARRAY_RANGE_REF
op0 → **inner-compref**
op1 → **val**
op2 → **val**
op3 → **val**
| VIEW_CONVERT_EXPR
op0 → **inner-compref**

condition : **val**
| RELOP
op0 → **val**
op1 → **val**

val : ID
| CONST

rhs : **lhs**
| CONST
| **call-stmt**
| ADDR_EXPR
op0 → **addr-expr-arg**
| UNOP
op0 → **val**
| BINOP
op0 → **val**
op1 → **val**
| RELOP
op0 → **val**
op1 → **val**
| COND_EXPR
op0 → **condition**
op1 → **val**
op2 → **val**

Referencias

[1] G.S. Sohi. Speculative multithreaded processors. *Computer*, 34(4):66, 2001.

[2] Robert H. and JR. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on programming languages and systems*, 7(4):501, 1985.

[3] P Marcuello, J Tubella, and A González. *Value prediction for speculative multithreaded architectures*. 1999.

[4] J.T. Oplinger, D.L. Heine, and M.S. Lam. In search of speculative thread-level parallelism. *Parallel Architectures and Compilation Techniques, proceedings*, pages 303–313, 1999.

[5] Y. Sazeides and J.E. Smith. The predictability of data values. *Micro -Annual workshop then annual international symposium-*, 30:248–258, 1997.

[6] C. García, , C. Madriles, J. Sánchez, P. Marcuello, A. González, and D.M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. *SIGPLAN Not.*, 40(6):269–279, 2005.

[7] A. González-Escribano and D.R. Llanos. Speculative parallelization. *Computer*, 39(12):126–128, December 2006. ISSN 0018-9162.

[8] M. Cintra and D. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *PPoPP '03: Proc. of the 9th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 13–24, San Diego, California, USA, June 2003. ACM Press.

[9] F. Dang and L. Rauchwerger. The r-lrpd test: speculative parallelization of partially parallelloops. pages 20–29. Parallel and Distributed Processing Symposium, 2002.

[10] D.R. Llanos, D. Orden, and B. Palop. Meseta: a new scheduling strategy for speculative parallelization of randomized incremental algorithms. *ICPP 2005 Workshops*, 2005.

[11] D. R. Llanos, D. Orden, and B. Palop. New scheduling strategies for randomized incremental algorithms in the context of speculative parallelization. *IEEE Transactions on Computers*, 56(6):839–852, 2007.

[12] A. Kejariwal, X. Tian, M. Girkar, W. Li, S. Kozhukhov, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos. Tight analysis of the performance potential of thread speculation using spec cpu 2006. pages 215–225, 2007.

[13] M Gircar and C.D. Polychronopoulosi. Extracting task-level parallelism. *ACM Transactions on programming languages and systems*, 17(4):600, 1995.

[14] M. Girkar and C.D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, 3:166–178, 1992.

[15] J.H. Saltz and R. Mirchandaney. *The Preprocessed Doacross Loop*. 1990.

[16] J.H. Saltz. Run-time parallelization and scheduling of loops. *IEEE Transactions on computers*, 40(5):603, 1991.

[17] S.T. Leung and J. Zahorjan. Extending the domain and improving the execution performance of runtime parallelization. technical report. *Departament of Computer Science and Engineering, University of Washington*, 1992.

[18] S.T. Leung and J. Zahorjan. Improving the performance of runtime parallelization. *SIGPLAN notices*, 28(7):83, 1993.

[19] S. Rul, H. Vandierendonck, and K. Bosschere. Function level parallelism driven by data dependencies. *SIGARCH Comput. Archit. News*, 35(1):55–62, 2007.

[20] J. Merrill. Generic and gimple: A new tree representation for entire functions. *Proceedings of the 2003 GCC Summit*, May 2003.

[21] D. Novillo. Tree ssa. a new optimization infrastructure for gcc. *Red Hat Canada, Ltd.*, 2003.

[22] D. Novillo. Design and implementation of tree ssa. *Red Hat Canada, Ltd.*, 2004.

[23] D. Yuste, J. Sahuquillo, S. Petit, P. López, and J. Duato. Evaluating the performance potential of function level parallelism. *Proceedings of the 12th Workshop on Interaction between Compilers and Computer Architectures*, February 2008.

[24] D. Yuste, J. Sahuquillo, S. Petit, P. López, and J. Duato. Function level parallelism: a first approach. *XVIII Jornadas de Paralelismo, en el II Congreso Español de Informática (CEDI)*, 2007.