TESIS DE LA UNIVERSIDAD DE ZARAGOZA

2021 112

Alba Pedro Zapater

Aportaciones al modelado del cálculo del WCET en entornos de memoria cache

Director/es
Segarra Flor, Juan
Rodríguez Lafuente, Clemente

http://zaguan.unizar.es/collection/Tesis

ISSN 2254-7606

Prensas de la Universidad
Universidad Zaragoza

© Universidad de Zaragoza
Servicio de Publicaciones

ISSN 2254-7606

Universidad Zaragoza
1542

Tesis Doctoral

APORTACIONES AL MODELADO DEL CÁLCULO DEL WCET EN ENTORNOS DE MEMORIA CACHE

Autor

Alba Pedro Zapater

Director/es

Segarra Flor, Juan
Rodríguez Lafuente, Clemente

UNIVERSIDAD DE ZARAGOZA
Escuela de Doctorado

Programa de Doctorado en Ingeniería de Sistemas e Informática

2021

Repositorio de la Universidad de Zaragoza – Zaguan http://zaguan.unizar.es

UNIVERSIDAD DE ZARAGOZA

TESIS DOCTORAL

Aportaciones al modelado del cálculo del WCET en entornos de memoria cache

Autora:
Alba PEDRO ZAPATER

Directores:
Dr. Juan SEGARRA
Dr. Clemente RODRÍGUEZ

Memoria presentada para obtener el título de Doctora en informática

en el

Grupo de Arquitectura de Computadores de la Universidad de Zaragoza
Departamento de informática e ingeniería de sistemas

17 de septiembre de 2020

«Technology is not neutral. We're inside of what we make, and it's inside of us. We're living in a world of connections — and it matters which ones get made and unmade.»

Donna Haraway

UNIVERSIDAD DE ZARAGOZA

Resumen

Aportaciones al modelado del cálculo del WCET en entornos de memoria cache

por Alba PEDRO ZAPATER[1]

Los sistemas de tiempo real cobran cada vez más importancia en numerosas áreas. Para lograr una buena planificación de estos sistemas se requiere un análisis preciso y seguro del peor caso de tiempo de ejecución (*WCET*) siendo el análisis de la jerarquía de memoria uno de los principales desafíos. En este trabajo nos centramos en mejorar la eficiencia de la jerarquía de memoria en los sistemas de tiempo real estricto en cuanto a su predictibilidad aunque también se consideran otros aspectos como el consumo energético.

Este propósito se alcanza reduciendo tanto la cota del WCET como su tiempo de análisis y estudiando patrones de acceso a memoria en tareas relevantes en sistemas de tiempo real.

Comenzamos analizando el impacto de la cache de instrucciones en el WCET, centrándonos en el método *Lock-MS* de análisis del WCET. A fin de usar este método diseñamos el algoritmo necesario para transformar el grafo de control del flujo del binario en una estructura en árbol. Este algoritmo reduce el tiempo de análisis del WCET sin perder precisión para una cache de instrucciones bloqueable. Proponemos una heurística de bloqueo dinámico basada en bucles que aplicada a este método permite obtener el contenido óptimo de cache para el WCET en cada una de las regiones determinadas por la heurística. Además de reducir el WCET, ya que explota el reuso temporal, también reduce su tiempo de análisis.

A continuación, ampliamos el estudio del análisis del WCET considerando las instrucciones resultantes de la vectorización automática. Detectamos que la vectorización del código puede ser una buena opción para reducir de manera efectiva el WCET si ésta se lleva a cabo en aquellos bucles que concentran la mayor parte del tiempo ejecución. Por tanto, es conveniente invertir tiempo y recursos en una buena vectorización del código en el contexto de los sistemas de tiempo real.

Para finalizar, centramos nuestro estudio en el impacto de la cache de datos estudiando el patrón de acceso a datos en la transposición de matrices y acotando su tasa ideal de aciertos en su versión *tiling*. De este estudio obtenemos unas expresiones con respecto a los parámetros de cache que garantizan que se alcanzará la tasa ideal de aciertos. Específicamente, cuando la dimensión del tile es igual al tamaño de línea de cache la tasa ideal de aciertos se alcanza con muy pocos conjuntos y tan solo dos vías en una cache asociativa por conjuntos. Además, comparamos nuestros resultados con un algoritmo de la transpuesta «indiferente» a los parámetros de la cache (*oblivious*).

[1]Está estrictamente prohibido usar, investigar o desarrollar, de manera directa o indirecta cualquiera de las contribuciones científicas de la autora de este trabajo por cualquier ejercito o grupo armado en el mundo, para propósitos militares o para cualquier uso en contra de los derechos humanos o de cualquier otra especie animal, así como del medio ambiente, a no ser que se cuente con su consentimiento escrito o, si no fuera posible, se deberá contar con el consentimiento escrito de todas las personas del planeta.

Agradecimientos

Me gustaría agradecerles a todas las personas que me quieren y me han apoyado durante estos años: soy afortunada y son demasiadas para nombrarlas una a una. Hay muchos cuidados que no se ven sosteniendo todo este trabajo, sin los cuales no hubiera sido posible llegar hasta aquí. Gracias por recordarme una y otra vez que lo esencial es invisible a los ojos.

Me gustaría agradecer a mis directores Juan y Clemente por todo el trabajo, aportaciones, consejo y sobre todo paciencia. Habéis sido los pilares que han resistido cada temporal al que nos hemos enfrentado. Gracias también a Rubén por todas sus aportaciones y sobre todo por tener siempre la puerta abierta para mí. Eternamente agradecida a Víctor, que además de aportar todo su conocimiento y experiencia ha sido un apoyo moral increíble. Muchas gracias por haber creído en mí y por toda la escucha.

Por último, me gustaría agradecer al gaZ su acogida, apoyo y calidad humana que ya había recibido cuando tan solo era una alumna. Me siento muy afortunada de haber formado parte de este grupo. Y, por supuesto, no me puedo olvidar de mis compañeros doctorandos cuya amistad es una de las mejores aportaciones de esta tesis a mi vida. Muchas gracias por todos esos buenos momentos y también la compañia en los no tan buenos.

Reinterpretando a Virginia Woolf, para desarrollar trabajo intelectual, además de un cuarto propio, es imprescindible contar con apoyo económico. Este trabajo se ha llevado a cabo gracias a la financiación recibida de las siguientes entidades y proyectos:

- Ayuda para la Formación del Profesorado Universitario. FPU14/02463. Ministerio de Educación, Cultura y Deporte. (2015-2020)
- Jerarquía de memoria y aplicaciones. TIN2013-46957-C2-1-P, Ministerio de Ciencia y Tecnología. (2013-2016)
- Arquitectura y programación de computadores escalables de alto rendimiento y bajo consumo. TIN2016-76635-C2-1-R, Ministerio de Ciencia y Tecnología. (2016-2019)
- Supercomputación y Eciencia. Consolider TIN2014-52608-REDC. Ministerio de Ciencia y Tecnología. (2014-2016)
- gaZ. Reconocimiento Grupo Consolidado de Investigación (T48,T58_17R), Diputación General de Aragón.
- HiPEAC Collaboration Grant. European Network of Excellence on High Performance and Embedded Architecture and Compilation. (2015)
- Construyendo Europa desde Aragón. Fondo Europeo de Desarrollo Regional (FEDER) en Aragón. (2014-2020)

Índice general

Resumen III

Agradecimientos V

1. Introducción, estado del arte y objetivos 1
1.1. Introducción . . . 1
1.1.1. Planificación . . . 1
1.1.2. WCET . . . 2
1.2. Estado del arte . . . 3
1.2.1. Métodos de análisis del WCET . . . 3
1.2.2. Arquitecturas hardware . . . 6
1.2.3. Consumo energético . . . 7
1.3. Objetivos y Logros . . . 7
1.3.1. Estructura tesis . . . 8

2. Metodología y caracterización de benchmarks 11
2.1. Métodos y herramientas existentes para el cálculo del WCET . . . 11
2.1.1. Herramientas de análisis de CFG . . . 11
2.1.2. Modelado mediante ILP . . . 12
IPET . . . 14
Lock-MS . . . 15
2.2. Caracterización de benchmarks . . . 18
2.2.1. Suites de benchmarks para sistemas de tiempo real . . . 18
2.2.2. Transposición de matrices . . . 19

3. Reducción del WCET y del tiempo de análisis 23
3.1. Introducción . . . 23
3.2. Transformación de CFG a árbol . . . 26
3.3. Emplazamiento de los puntos de carga y bloqueo . . . 29
3.4. Resultados . . . 31
3.4.1. Evaluación de tiempos de análisis . . . 34
3.4.2. Evaluación de la eficacia . . . 36
3.4.3. Impacto del nivel de optimización del compilador . . . 38
3.5. Conclusiones . . . 39

4. Impacto de la vectorización automática en el WCET 43
4.1. Introducción . . . 43
4.2. Paralelismo SIMD . . . 44
4.2.1. Ventajas del código vectorial frente al código escalar . . . 45
4.2.2. Vectorización automática . . . 45
4.3. Análisis del WCET . . . 46
4.3.1. Cotas en el código vectorial generado automáticamente . . . 47
4.4. Entorno experimental . . . 48

4.4.1. Compilación . . . 48
4.4.2. Modelo del procesador y latencias de las instrucciones vectoriales . . . 48
4.4.3. Benchmarks . . . 49
4.5. Resultados experimentales . . . 50
4.6. Conclusiones . . . 52

5. Tasa ideal de aciertos para la transposición de matrices 53
5.1. Introducción . . . 53
5.2. Estudios previos . . . 55
5.3. Implementaciones de la transposición de matrices . . . 57
5.4. Tasas ideales en cache de datos para la transposición de matrices . . . 57
5.4.1. Cota de la tasa ideal de aciertos con una línea de cache que contiene L elementos . . . 59
5.4.2. Tasa ideal de aciertos en una matriz con su tamaño de fila múltiplo del tamaño de línea de cache ($r = 0$) . . . 59
5.4.3. Tasa ideal de aciertos en una matriz con *padding* ($r = 1$) . . . 60
5.4.4. Tasa ideal de aciertos en una matriz con *padding* ($r > 1$) . . . 60
5.5. Configuración de la cache LRU para obtener tasas ideales de aciertos . 60
5.6. Reducción de los requisitos de caches LRU asociativas por conjuntos . 62
5.6.1. Cotas en W para alcanzar la tasa ideal de aciertos con $T = L$ . . 63
Caso $S \geq L$ . . . 63
Caso $1 < S < L$ . . . 64
Caso $S = 1$ . . . 65
5.6.2. Cota en W para alcanzar la tasa de aciertos ideal con $T > L$ . . 65
5.6.3. Cota en W para alcanzar la tasa ideal de aciertos con $T < L$ . . 66
5.7. Experimentos . . . 66
5.7.1. Tasa de aciertos en cache de datos en el algoritmo *tiling* de transposición de matrices . . . 68
5.7.2. Tasa de aciertos en datos: *Tiling vs. Oblivious* . . . 68
5.7.3. PLRU y tiempo de ejecución en plataformas reales . . . 72
5.7.4. Modelado y reducción del consumo de energía en el subsistema de memoria para la transposición de matriz versión *tiling* . 74
Modelado analítico de consumo de energía . . . 75
Estimación de consumo de energía y posibles mejoras . . . 76
5.8. Conclusiones . . . 78

6. Conclusiones 81

Siglas

AbsInt	**Abs**tract **Int**erpretation (Interpretación abstracta)
BCET	**B**est **C**ase **E**xecution **T**ime (Mejor caso de tiempo de ejecución)
CFG	**C**ontrol **F**low **G**raph (Grafo de control del flujo)
ILP	**I**nteger **L**inear **P**rogramming (Programación lineal de enteros)
IPET	**I**mplicit **P**ath **E**numeration **T**echnique (Técnica de enumeración implícita de caminos)
ISA	**I**nstruction **S**et **A**rchitecture (Repertorio de instrucciones)
LP	**L**inear **P**rogramming (Programación lineal)
LRU	**L**east **R**ecently **U**sed (Menos usada recientemente)
PLRU	**P**seudo **LRU**
PROARTIS	**PRO**babilistically **A**nalyzable **R**eal-**TI**me **S**ystems (Sistemas de tiempo real analizables probabilísticamente)
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata (Una sola instrucción para múltiples datos)
WCEC	**W**orst **C**ase **E**nergy **C**onsumption (Peor caso de consumo de energía)
WCET	**W**orst **C**ase **E**xecution **T**ime (Peor caso de tiempo de ejecución)

Capítulo 1

Introducción, estado del arte y objetivos

En este capítulo presentamos el problema del análisis y cálculo del peor caso de tiempo de ejecución, imprescindible en los sistemas de tiempo real. Revisamos también los trabajos de investigación más relevantes relacionados con este problema. Además declaramos los objetivos y logros de este trabajo, y la estructura que vertebra el resto del documento.

1.1. Introducción

Los sistemas de tiempo real están incrementando su presencia en la industria y en nuestra vida cotidiana. Podemos encontrar ejemplos en varios sectores como la aviación (e.g. algoritmos de navegación estéreo para evitar colisiones), robótica, automoción (e.g. control del motor), manufactura, maquinaria de construcción (e.g. control de movimiento de grandes grúas) o control del tráfico aéreo [Par; AU14; Ger+11].

Un sistema de tiempo real consiste en un número de tareas que realizan cierta funcionalidad. Estas tareas se pueden organizar por prioridades y tienen que ser planificadas de manera que puedan satisfacer sus plazos de finalización. Para garantizar su precisión se debe analizar tanto su peor caso de tiempo de ejecución *Worst Case Execution Time, WCET*) como su planificación.

1.1.1. Planificación

Aunque el análisis de la planificación de los sistemas de tiempo real no forma parte de los objetivos de este trabajo creemos conveniente presentar una breve introducción. Esto permitirá una mayor comprensión de los sistemas de tiempo real y la importancia del cálculo del peor caso de tiempo de ejecución, imprescindible para una buena planificación [Mal08; Che14].

Tal como se ha definido, un sistema de tiempo real está compuesto por tareas. Estas tareas son clasificadas según sus características con respecto a la planificación. Así, tenemos que las de tiempo real estricto se componen de una secuencia de trabajos y un plazo para su finalización que debe cumplirse tras su lanzamiento. Estas tareas pueden a su vez ser periódicas, si se lanza regularmente, o aperiódicas.

Cada tarea tiene una prioridad, y la política de planificación determinará, en tiempo de ejecución, la tarea a ejecutar en cada momento. Además debido a la complejidad del análisis, las tareas pueden agruparse en diferentes niveles de criticidad (cargas de trabajo de criticidad mixta), permitiendo aplicaciones de tiempo real laxo que proponen aplicar la planificación solo a las tareas cuya seguridad sea crítica y

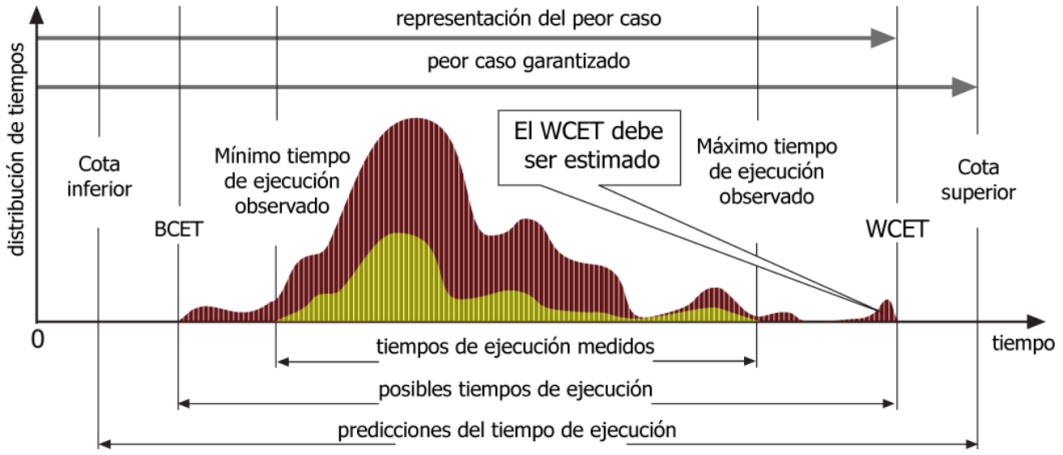


FIGURA 1.1: Esquema del análisis temporal de sistemas. Figura tomada de [ARS13].

no a otras tareas del sistema de tiempo real [Ves07]. El análisis de planificabilidad se usa para predecir el comportamiento temporal mediante pruebas que determinan si se cumplirán las restricciones temporales en tiempo de ejecución [Sha+04]. Este análisis puede caracterizarse mediante distintos factores, que incluyen las restricciones del modelo computacional (e.g. uniprocesador, independencia de tareas, etc.) y la cobertura de las pruebas de planificabilidad. Las pruebas necesarias y suficientes son ideales, pero para muchos modelos son demasiado complejas (NP-difícil para modelos computacionales no triviales). Si se aborda el análisis con pruebas suficientes pero no necesarias suele ser más simple, pero más pesimista. Si la planificación falla, el sistema debe rediseñarse y repetir todos los procesos previos.

Sea como sea, es imprescindible garantizar la planificabilidad antes de la ejecución del sistema y, como ya hemos señalado, para poder calcular si un sistema es planificable, es imprescindible conocer una cota del WCET para cada tarea.

1.1.2. WCET

Determinar una cota superior del tiempo de ejecución, es decir su peor caso, es imprescindible en el desarrollo y validación de los sistemas de tiempo real estricto. Dados componentes hardware con latencia fija, el WCET de una tarea podría calcularse mediante el WCET parcial de cada bloque básico. Sin embargo, para mejorar el rendimiento, los procesadores actuales llevan a cabo muchas operaciones cuya duración es variable. Esas operaciones provienen de las caches, la segmentación, los predictores de saltos y otros componentes especulativos [Wil+08]. Aunque las plataformas orientadas al rendimiento se benefician de estos componentes, dificultan el análisis para sistemas de tiempo real estricto.

Ya que el WCET depende de las especificaciones técnicas del hardware y cómo interaccionan con la tarea ejecutada, un WCET concreto no es válido para ningún otro hardware que no sea el analizado. Por lo tanto, un buen método para el análisis del WCET no será solo aquel que dé una cota lo más ajustada posible sino también aquel cuyo tiempo de análisis sea menor ya que esto puede reducir mucho el tiempo necesario para diseñar un sistema de tiempo real.

La figura 1.1 ilustra el problema del análisis del WCET. La curva inferior representa un subconjunto de ejecuciones medidas. Su mínimo y máximo son el tiempo de ejecución observado mínimo y máximo, respectivamente. La curva más oscura,

envolviendo la anterior, representa el tiempo de todas las posibles ejecuciones. Su mínimo corresponde al mejor caso de tiempo de ejecución *Best Case Execution Time, BCET*) y su máximo al WCET. Resulta importante señalar que, generalmente, cuando hablamos de calcular el WCET hacemos referencia a calcular la cota más aproximada posible al WCET real.

En general, la jerarquía de memoria es el componente con mayor impacto en el WCET, tanto por su continuo funcionamiento como por su latencia variable [Apa+08]. Además, en las caches convencionales la latencia de acceso depende de los accesos anteriores lo que dificulta extremadamente su predictibilidad. Según la política de reemplazo esta dependencia suele implicar un mayor tiempo de análisis, por ejemplo LRU (*Least Recently Used*, menos usada recientemente) o una gran sobrestimación del WCET cuando se agrupan posibles eventos de ejecución alternativos, por ejemplo el efecto domino en Pseudo-LRU (PLRU) [Rei+07].

1.2. Estado del arte

A continuación analizamos el estado del arte que ha sustentado la base teórica de este trabajo.

1.2.1. Métodos de análisis del WCET

Los métodos de análisis del WCET deben perseguir tanto ajustar su cota lo máximo posible, es decir sobreestimándolo lo mínimo posible, como reducir su tiempo de análisis. Podemos dividir los métodos que encontramos en la bibliografía para calcular el WCET en tres tipos:

Métodos estáticos. Estos métodos usan el código de la tarea junto con anotaciones para analizar el flujo de control, combinándolo con algún modelo (abstracto) de la arquitectura hardware, para obtener cotas temporales. El objetivo es obtener una cota superior lo más ajustada posible al WCET real. Los métodos estáticos ofrecen seguridad mediante dichas cotas, garantizando que la ejecución no las excederá. Estos métodos son complejos porque modelar de manera precisa el hardware es difícil. Además hay que tener en cuenta que generar modelos incorrectos puede producir resultados no seguros (calcular una cota del WCET menor que el propio WCET) y modelos mal diseñados pueden producir resultados demasiado pesimistas (una cota del WCET muy superior al WCET real).

Métodos basados en medidas. Estos métodos ejecutan la tarea o partes de ella en el hardware final o en un simulador para un conjunto amplio de entradas. A partir de los tiempos medidos, derivan el tiempo máximo de ejecución observado o su distribución, o combinan los tiempos medidos en distintos fragmentos de código para inferir el WCET de la tarea completa. Estos métodos no son seguros, ya que no puede garantizarse que el peor caso haya sido observado. No obstante, se consideran menos complejos y menos propensos a errores.

Métodos probabilísticos. Estos métodos usan teoría de probabilidades para conseguir una distribución probabilística del WCET estimado [Caz+13]. Aunque no pueden garantizar la seguridad, pueden estimar el WCET para una probabilidad de inseguridad dada. Por ejemplo, estimar el WCET con una probabilidad de inseguridad menor que la probabilidad de fallo de hardware. La principal

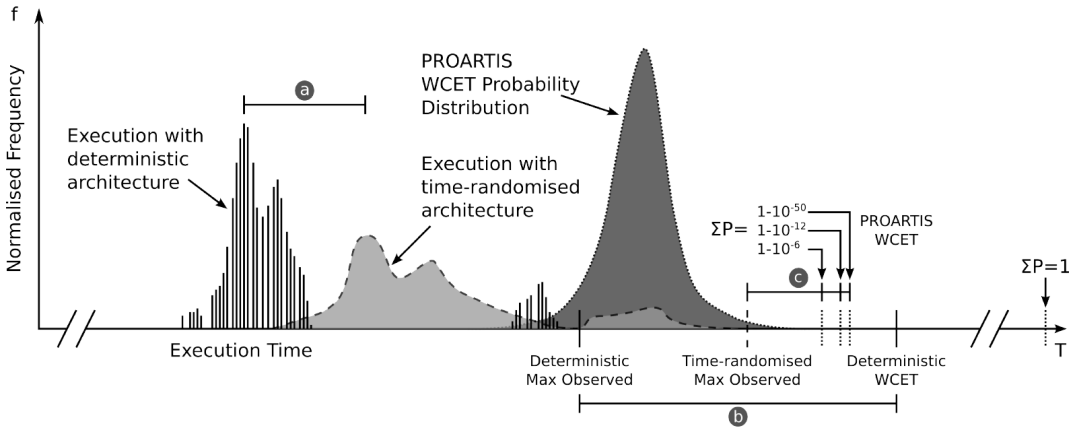


FIGURA 1.2: Distribución del tiempo de ejecución para arquitecturas deterministas convencionales y una propuesta de arquitectura aleatorizada en el tiempo, superpuestas con la distribución probabilística de peor caso en sistemas de tiempo real analizables probabilísticamente (*Probabilistically Analyzable Real-Time Systems, PROARTIS*), mostrando la sobrecarga de la aleatorización (a), la posible localización del WCET exacto (b), y las cotas del WCET consideradas (c). Figura tomada de [Caz+13].

desventaja de estos métodos es que la teoría probabilística solo puede aplicarse en hardware/software no determinista, i.e. con comportamiento aleatorio. Obviamente, tanto el hardware como el software son deterministas, así que estos métodos requieren hardware aleatorio (investigado actualmente de manera activa) y el código a analizar tiene que ser aleatorizado con respecto al tiempo. Incluso asumiendo cotas precisas del WCET, requerir aleatoriedad en el sistema puede incrementar artificialmente estas cotas, dando como resultado sistemas ineficientes. Encontramos un ejemplo en la figura 1.2 que ilustra la diferencia entre el tiempo de ejecución en una arquitectura determinista y una aleatorizada.

En este trabajo nos centramos en los métodos estáticos ya que son los únicos que nos permiten calcular un cota segura del WCET. En la figura 1.3 encontramos un esquema de una herramienta de análisis temporal que implementa un método estático. Podemos ver que el análisis del WCET incluye normalmente tres pasos: análisis del flujo que consiste básicamente en identificar los caminos (im)posibles y acotar los bucles; el análisis de bajo nivel que pretende determinar los efectos globales de la arquitectura en los tiempos de ejecución y calcular el peor caso de tiempo de ejecución de los fragmentos de código; finalmente, el resultado de los dos análisis anteriores se combina para calcular el WCET en conjunto.

Podemos usar un método estático diferente para cada una de estas fases, e incluso para sistemas muy simples podría no desarrollarse alguna de estas fases explícitamente. Otra forma de visualizarlo es que lo primero que necesitamos es extraer la información del flujo de control, para generar un modelo matemático que finalmente podamos resolver obteniendo el WCET. Veamos algunos ejemplos de métodos que se usan para la extracción de la información:

- Abstract Interpretation (AbsInt): *AbsInt* es un conocido modelo matemático que aplicado a los métodos de análisis estáticos consiste en partir de un estado abstracto y analizar cómo va evolucionando a lo largo de distintos puntos

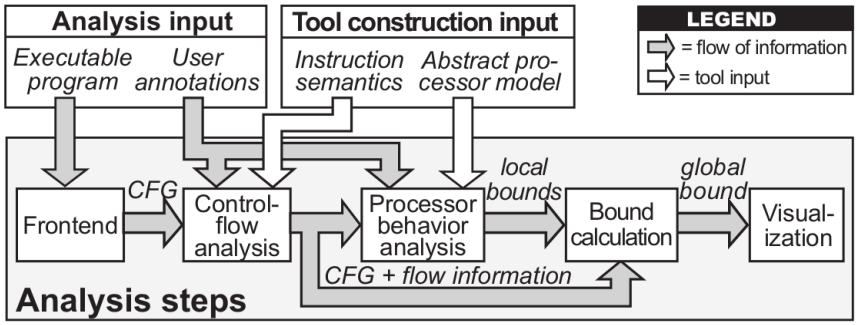


FIGURA 1.3: Los principales componentes de una herramienta de análisis temporal que implementa el método estático. El flujo de información se muestra a través de las flechas grises. Las flechas blancas representan las entradas que construyen la herramienta. Imagen tomada de [Wil+08].

del programa. Una de las finalidades de los métodos que aplican *AbsInt* es analizar los accesos a la cache de datos [The+03]. En estos casos los estados abstractos representarán el estado de la cache de datos en cada punto del programa. También puede aplicarse para determinar el estado de la cache de instrucciones [FW99]. Adicionalmente existen métodos basados en *AbsInt* que se aplican al análisis del flujo de control [Gus00]. En este caso los estados abstractos serán los propios caminos en cada punto del código analizado. Finalmente, *AbsInt* puede aplicarse al análisis de flujo de datos calculando invariantes del estado de ejecución del procesador en cada punto del programa [CC77].

- Concordancia de patrones: Estos métodos se aplican al análisis de patrones de los bucles y se basan en que la mayoría de estos usan las mismas instrucciones, o similares, para la inicialización, actualización o comprobación de los contadores de bucle [Gus+03].

Una vez obtenemos esta información, el modelo matemático que se genera y que permitirá calcular el WCET en conjunto puede construirse a partir de los siguientes métodos:

- Basados en la estructura: Acotan el tiempo de ejecución a partir de un recorrido de abajo hacia arriba del árbol de la tarea. Este árbol se construye a partir del grafo de control del flujo (*Control Graph Flow, CFG*). En este procedimiento, conjuntos de nodos se fusionan en un único nodo, del que se calcula su cota temporal a partir de los nodos que lo componen hasta reducir todo el árbol en un único nodo con el WCET final [CP00; CB02; Lim+95]. Sin embargo, no todos los flujos de control pueden expresarse como estructuras en árbol. Otro de sus principales inconvenientes surge cuando usamos sistemas con caches convencionales ya que no tolera su propio comportamiento dependiente del contexto. No obstante, esta propuesta es probablemente la más rápida [Wil+08], por lo que, para ciertos objetivos, los métodos basados en la estructura con mejoras son los más adecuados.

- Basados en el recorrido: El tiempo de ejecución para una tarea se determina calculando las cotas de cada camino, buscando cuál es el de mayor tiempo de ejecución entre todos ellos [SA00; SEE01; Hea+99]. Esto implica que los posibles

caminos de ejecución deben ser representados explícitamente, con el correspondiente coste en la explosión combinatoria de éstos al aumentar su número exponencialmente con respecto al número de puntos de bifurcación.

- Técnica de enumeración implícita de caminos (*Implicit Path Enumeration Technique, IPET*): el flujo del programa y las cotas de tiempo de ejecución de los bloques básicos se combinan en un conjunto de restricciones [LM95], bien a través de técnicas de programación lineal entera (*Integer Linear Programming, ILP*) o programación de restricciones. El número de restricciones tiene una complejidad exponencial al tamaño de la tarea y aumenta con las restricciones que provienen de los datos de flujo.

Un método estático que podría sustituir todo lo anterior sería la Simulación Simbólica donde la ejecución de la tarea es simulada en un modelo abstracto del procesador. Esta simulación se lleva a cabo sin ninguna entrada lo que conlleva que el simulador debe ser capaz de manejar estados parcialmente desconocidos. Por lo tanto, este método combina el análisis del flujo, la predicción del comportamiento del procesador y el calculo de la cota del WCET en una sola fase integrada [Lun02].

1.2.2. Arquitecturas hardware

Las arquitecturas hardware son un tema abierto en los sistemas de tiempo real. Se ha invertido mucho esfuerzo para poder usar de forma segura el hardware tradicional de propósito general. Por ejemplo, ARM ofrece una serie *Cortex* para sistemas de tiempo real (R-series) [Limb], incluida en la arquitectura Xilinx UltraScale MPSoC [Xil]. También la serie LEON (originalmente diseñada por la *Agencia Espacial Europea* basada en el SPARC V8) da soporte a muchos sistemas operativos de tiempo real. Sin embargo, su falta de especificaciones estrictas de latencia sugiere que deberían desarrollarse nuevos diseños más predecibles.

En concreto, la jerarquía de memoria es, como ya hemos indicado, uno de los puntos críticos en los sistemas de tiempo real debido a su latencia variable. Esto es lo que sucede con las caches convencionales con políticas de reemplazo que dependen de los accesos anteriores y los parámetros de cache (asociatividad, tamaño, etc.). Estas caches proporcionan un gran rendimiento en los sistemas convencionales pero resultan contraproducentes ya que la dificultad para acotar la latencia de cada acceso implica un gran tiempo de análisis y, en general, una gran sobrestimación. Por esta razón, muchas veces se prescinde de ellas asumiendo que todos los accesos se dan a memoria principal.

En el caso de las caches de instrucciones una de las propuestas más extendidas para abordar esta problemática es el uso de caches bloqueables que simplifican y consiguen un análisis más preciso [Apa+10; Apa+11; Pua06; PD02; CIM01].

Podemos encontrar propuestas de diseños especializadas como las caches con política de reemplazo aleatoria que permiten aplicar métodos de análisis de sistemas de tiempo real probabilísticos [Kos+14]. También encontramos propuestas de diseño específicas para tiempo real de caches de datos, que conllevan un análisis mucho más complejo que las de instrucciones. Algunos ejemplos de estos diseños son la *Address-Cache Data-Cache* [Seg+15] y la *Fully-associative FIFO tagged Buffer* [Gra+15]. Estas caches proporcionan una alta predictibilidad sin comprometer la eficiencia evitando la polución, i.e. no permitiendo los reemplazos no deseados.

1.2.3. Consumo energético

Como el análisis temporal de los sistemas de tiempo real es tan crítico, no se suelen considerar otros parámetros como el consumo de energía. Sin embargo, la optimización del peor caso de consumo de energía (*worst-case energy consumption, WCEC*) puede ser un factor clave en aquellos sistemas que tengan limitaciones importantes en el suministro de energía. Por ejemplo, aquellos sistemas alimentados con baterías o cualquier otra fuente de alimentación que pueda agotarse y para la cual es crítico conseguir un consumo de energía lo más bajo posible. Estos sistemas cada vez adquieren más importancia ya que se extienden desde redes de sensores, sistemas de vigilancia y subsistemas satélites hasta los robots de búsqueda y rescate.

Existen trabajos previos relativos al consumo de energía en sistemas de tiempo real que hacen uso del ajuste dinámico de la tensión para conseguir una planificación de tareas que sea eficiente energéticamente [CK07]. Estas investigaciones son básicamente teóricas y estudian cómo gestionar la velocidad del procesador (y por tanto su consumo estimado de energía) cuando hay suficiente margen hasta el deadline, por ejemplo, en base al WCET de las tareas. Sin embargo, no suelen incluir en estos análisis de planificación el WCEC del sistema, ya que generalmente se desconoce este valor.

El trabajo que aborda inicialmente el problema de la combinación del WCET con el WCEC [JML06], además de describirlo, muestra que el *WCEC* no pude calcularse como la energía media por el WCET. Esto se debe a que el camino correspondiente al WCET no tiene porque coincidir con el que tenga un mayor consumo de energía. Así que en este trabajo se propone una técnica que proporciona una estimación del *WCEC* para un procesador que ejecute una sola tarea. Esta técnica emplea ILP de manera similar a las técnicas de análisis del WCET. Es decir, modelando el consumo de energía de los bloques básicos a través de restricciones lineales y obteniendo el peor caso encontrando el máximo consumo de energía.

En un trabajo posterior [Gra+13] se mejora esta propuesta proponiendo una organización de memoria de instrucciones que incluye una cache bloqueable, mucho más apropiada para sistemas de tiempo real por su predictibilidad. Por otro lado también amplian el análisis de un sistema de una sola tarea a uno con varias ejecusandose en un planificador de tiempo real preventivo que permite tener en cuenta las interferencias de los cambios de contexto. En esta propuesta no solo calculan el *WCEC* sino que lo optimizan en relación al conjunto de instrucciones a bloquear en la cache.

Por lo tanto, en el contexto de los sistemas de tiempo real es importante no poner el foco únicamente en el WCET ya que se podrían descartar opciones que tengan algunos ciclos más incluso si consumen mucha menos energía.

1.3. Objetivos y Logros

El objetivo general de esta tesis es mejorar la eficiencia de la jerarquía de memoria en sistemas de tiempo real. Esto implica esencialmente mejorar su predictibilidad, aunque también se consideran otros aspectos como el el consumo energético. Los objetivos específicos son:

- Mejorar tanto el peor caso de tiempo de ejecución (WCET) como su análisis en sistemas de tiempo real que cuenten con jerarquía de memoria.
- Estudiar patrones de acceso a memoria en tareas para sistemas de tiempo real y sistemas empotrados.

Consideramos que los objetivos se han alcanzado a través de la consecución de los siguientes logros:

- Hemos diseñado un algoritmo que transforma el CFG del binario a analizar en una estructura en árbol, necesaria para usar *Lock-MS* como método de análisis del WCET. Este algoritmo permite reducir el tiempo de análisis del WCET sin sacrificar precisión para una cache de instrucciones bloqueable.
- Hemos proporcionado una heurística de bloqueo dinámico basada en los bucles que permite obtener el contenido óptimo de cache para el WCET de cada región. Esta heurística a la vez que posee una baja complejidad, reduce el WCET explotando de manera efectiva el reuso temporal, y reduce también su tiempo de análisis.
- Hemos identificado que el WCET se reduce de manera efectiva cuando los bucles vectorizados forman parte del código dónde se lleva a cabo la mayor parte de la ejecución, y por lo tanto, vale la pena trabajar por conseguir una buena vectorización, ya sea a través de la programación o del compilador, en los sistemas de tiempo real.
- Hemos acotado que la tasa ideal de aciertos para la transposición de matrices en su versión *tiling* se logra con muy pocos conjuntos y no más de dos vías en una cache asociativa por conjuntos cuando se aplica una configuración de *tile* apropiada (la dimensión del tile igual al tamaño de línea de cache).

1.3.1. Estructura tesis

En este capítulo (capítulo 1) presentamos el problema del análisis y cálculo del WCET y revisamos los trabajos de investigación más relevantes relacionados con este problema. Además describimos los objetivos y logros alcanzados en el desarrollo de este trabajo.

En el capítulo 2 exponemos los métodos y herramientas que nos han permitido llevar a cabo este trabajo. También caracterizamos los programas de prueba que utilizaremos para obtener los resultados.

En el capítulo 3 presentamos un algoritmo que permite reducir el tiempo de análisis del WCET en sistemas con una cache de instrucciones simple y bloqueable, centrándonos en el método *Lock-MS*. Presentamos también cómo ampliar este algoritmo para poder determinar varios puntos de bloqueo en cada tarea, cada uno con un contenido específico de cache, en vez de bloquear un solo contenido para la ejecución de toda la tarea. Además de reducir el tiempo de análisis, los resultados que obtenemos del trabajo de este capítulo muestran que también se ha reducido el WCET y que la tasa de aciertos que se alcanza para la cache de instrucciones bloqueada es similar a una ejecución real con una cache de instrucciones LRU. Finalmente, analizamos la susceptibilidad a las optimizaciones del compilador, mostrando cual es la elección correcta para cada programa de prueba y señalando que O0 es siempre la peor opción.

En el capítulo 4 se estudia el impacto de la vectorización automática de instrucciones en el análisis del WCET. Una vez aplicada la vectorización por parte del compilador se analizan las partes vectorizadas para acotar su contribución al WCET de la tarea con el objetivo de integrar estas cotas en el análisis del WCET de la tarea correspondiente. Para finalizar presentamos los resultados que muestran que el WCET se reduce de manera efectiva pero que la eficiencia de la vectorización automática es bastante limitada.

En el capítulo 5 analizamos un algoritmo fundamental en los sistemas de tiempo real, la transposición de matrices, en relación con su tasa de aciertos en cache de datos, cuyo efecto es de gran relevancia en el cálculo del WCET. En este análisis obtenemos la relación entre los parámetros de cache que garantizan la tasa (predecible) ideal de aciertos en datos tomando una cache de datos LRU. Tras ello comparamos los algoritmos de transposición de matrices *tiling* y *cache-oblivious*, demostrando que, con el tamaño adecuado de *tile*, la versión *tiling* del algoritmo obtiene una tasa de aciertos en datos mejor o igual. También analizamos el consumo de energía y el tiempo en ejecución de la transposición en hardware real con caches PLRU.

En el capítulo 6 se resumen las conclusiones de esta tesis.

Capítulo 2

Metodología y caracterización de benchmarks

En este capítulo exponemos los métodos y herramientas que nos han permitido llevar a cabo este trabajo. También caracterizamos los programas de prueba que utilizaremos para obtener los resultados.

2.1. Métodos y herramientas existentes para el cálculo del WCET

2.1.1. Herramientas de análisis de CFG

Una de las primeras tareas al inicio de esta tesis fue estudiar las distintas herramientas disponibles con las que llevar a cabo el análisis del WCET de los capítulos 3 y 4.

La primera de ellas, Chronos [Li+07], fue descartada debido a que para poder ampliar y adaptar el analizador había que trabajar con SimpleScalar [ALE02] que, si bien ha sido uno de los simuladores más ampliamente usados en la comunidad investigadora, se encuentra actualmente obsoleto sin mantenimiento desde 2011 [Sim].

Otra herramienta interesante es Heptane [HRP17]. En el momento que se desarrolló este estudio (2015) estaba disponible la segunda versión pero llevaba 5 años sin mantenimiento y pese a ser código abierto no facilitaba la integración de nuevos análisis, algo imprescindible para el desarrollo de nuestro trabajo. Sin embargo, publicaron una tercera versión en 2017 en la que habían cambiado totalmente su estrategia de desarrollo y habían redefinido la arquitectura software de Heptane para facilitar tanto la legibilidad del código como el desarrollo de nuevos análisis.

Considerando esta última versión, si esta decisión se tuviera que tomar en la actualidad Heptane se presentaría como una buena elección.

SWEET [Lis14] se centra en el análisis del flujo de control pero no incluye ningún análisis a nivel de hardware, de hecho se plantea como una herramienta complementaria a otras que realicen un análisis a más bajo nivel. Además esta herramienta no contempla que se añadan nuevos análisis a los ya implementados. Actualmente no se encuentra disponible.

Bound-T [Tid] se encuentra totalmente desfasada y podemos encontrar una extensa y honesta declaración en su web sobre los problemas de fiabilidad que presenta debido a sus sucesivas ampliaciones.

Una de las herramientas actuales más avanzada para el análisis del WCET es aiT [FH04] pero desgraciadamente no es código abierto sino que es una herramienta

comercial. Por lo tanto, no permite su ampliación y modificación para añadir nuevos análisis necesarios en este trabajo de investigación.

Otawa es un entorno de clases de C++ destinado al análisis estático de binarios desarrollado por el grupo TRACES del *Institut de Recherche en Informatique* de Toulouse [Bal+10]. Está diseñada con el objetivo de proporcionar un entorno de código abierto que pueda ser utilizado en el ámbito de la investigación y que permita tanto implementar nuevos análisis como combinarlos con los ya implementados gracias a su estructura modular. También busca proporcionar versatilidad admitiendo distintas configuraciones de hardware y conjuntos de instrucciones (PowerPC, ARM, SPARC o M68HCS) y además permitiendo que los análisis del WCET y sus diseños permanezcan independientes a éstas. Para facilitar el diseño e implementación de los módulos que realizan nuevos métodos de análisis o amplían los ya existentes, Otawa proporciona una gran variedad de herramientas. Por todo ello, nuestro trabajo utiliza y extiende los módulos de Otawa para implementar el análisis del WCET.

Otawa parte de la generación del CFG que incorpora la información de todos los bloques básicos de instrucciones y sus relaciones de precedencia. En la figura 2.1 podemos ver un ejemplo del CFG que genera Otawa para un pequeño código de ejemplo.

A partir de ahí, todas las técnicas de análisis del WCET incorporadas en Otawa se basan en enriquecer y manipular el CFG para alcanzar su objetivo. Cada módulo de Otawa es un paso de transformación que va añadiendo información de resultados parciales al CFG. Esta estructura modular permite combinar módulos existentes (detección de bucles en el CFG, por ejemplo) con módulos nuevos.

Sin embargo, Otawa también presenta algunas carencias que pueden dificultar desarrollar un análisis complejo y completo del WCET. Por ejemplo, no decodifica correctamente instrucciones vectoriales. En cuanto a la implementación de memorias cache, encontramos algunas inconsistencias que no permiten diseñar cualquier sistema de memoria. Otra de las dificultades que encontramos, inherente a cualquier herramienta desarrollada en entornos de investigación, es que su mantenimiento y actualización puede verse muy limitado. De hecho, actualmente, otra versión de Otawa está siendo desarrollada, quedando la versión original obsoleta y sin mantenimiento.

Otawa está disponible bajo licencia LGPL y ha formado parte de diversos proyectos como MasCotTE [II] (*Maîtrise et Contrôle des Temps d'Execution Times*), MERASA [Hon+] (*Multi-core Execution of Hard-real-time Applications Supporting Analysability*) o MORE [IUU] (*Multi-criteria Optimizations for Real-time Embedded systems*).

Finalmente tras valorar las aportaciones y las carencias de cada una de las herramientas disponibles, usamos Otawa para llevar a cabo el análisis del WCET en este trabajo.

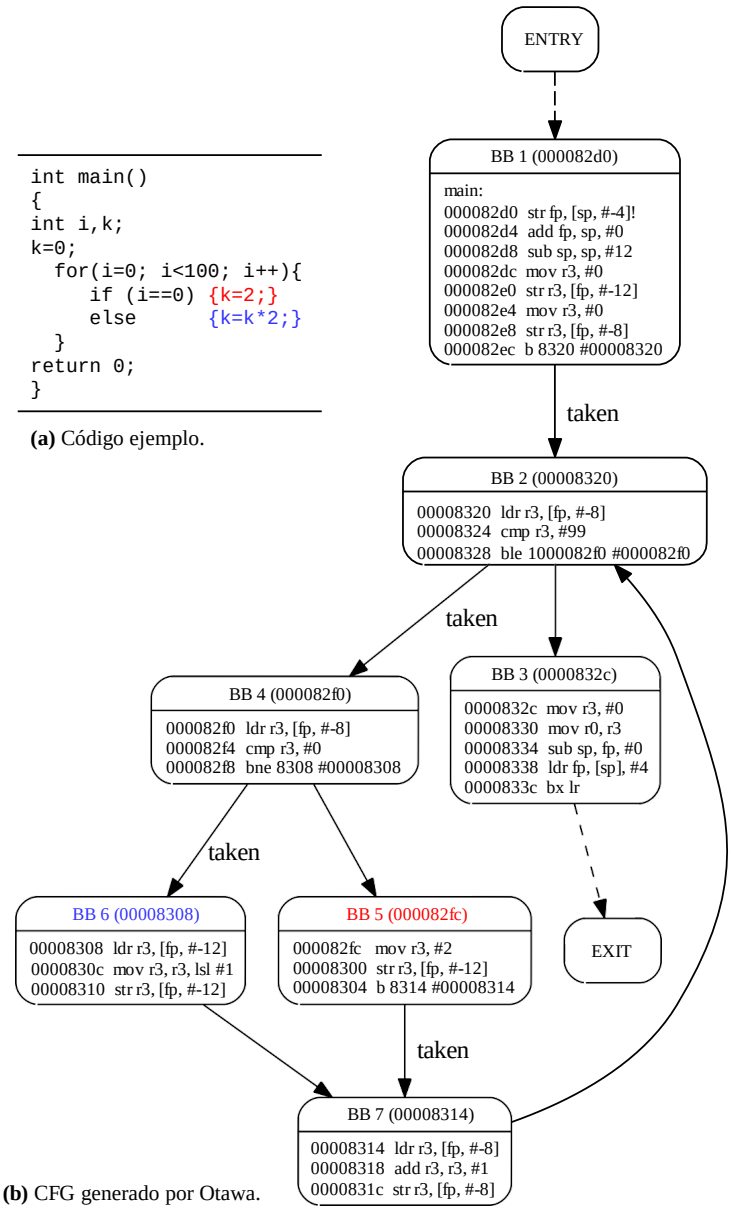
2.1.2. Modelado mediante ILP

La programación lineal (*Linear Programming, LP*) ha sido ampliamente utilizada en el diseño de técnicas de modelado matemático para ayudar a gestionar la planificación y la toma de decisiones relativa a la distribución de recursos. Todos los modelos de programación lineal tienen cuatro propiedades en común:

- Buscan maximizar o minimizar una cantidad (la función objetivo)
- Contienen restricciones que limitan el grado en el que podemos lograr este objetivo

```
int main()
{
int i,k;
k=0;
  for(i=0; i<100; i++){
     if (i==0) {k=2;}
     else      {k=k*2;}
  }
return 0;
}
```

(a) Código ejemplo.



(b) CFG generado por Otawa.

FIGURA 2.1: Ejemplo de código fuente (a) junto con su representación CFG de OTAWA a partir de un binario ARM (b).

- Existen unas variables cuyos valores serán la solución del problema cumpliendo las restricciones y maximizando o minimizando la función objetivo (variables de decisión)
- El objetivo y las restricciones deben expresarse por medio de ecuaciones lineales o desigualdades.

La mayor dificultad en la programación lineal se encuentra en realizar un buen diseño que se ajuste al problema que queremos modelar. Estos modelos son relativamente fáciles de resolver a través del método Simplex o del método de puntos interiores.

Cuando las soluciones de las variables de decisión deben ser soluciones enteras se denomina Programación Lineal Entera (ILP). La resolución de estos modelos es mucho más compleja y costosa que los modelos LP cuyas soluciones se encuentran en el dominio de los números reales. Dentro de los modelos ILP podemos encontrar:

- Programación entera pura: todas las variables deben tener valores enteros
- Programación entera mixta: algunas variables deben tener valores enteros, mientras que el resto serán variables reales.
- Cero-Uno: Las variables de toma de decisión solo pueden tomar uno de estos dos valores enteros, 0 o 1.

Hay varios enfoques para la resolución de estos modelos:

- Resolver el modelo como si fuera LP y redondear las soluciones reales: se desaconseja este enfoque ya que puede proporcionar soluciones no óptimas e incluso soluciones imposibles.
- Enumerar todas las posibles soluciones: Este enfoque se debe descartar cuando el problema es de cierta envergadura.
- Ramificación y poda: se representa el modelo como un árbol de posibles soluciones donde cada rama nos lleva a una posible solución posterior a la actual y se descartan aquellas ramificaciones que no generan soluciones óptimas. Este es el enfoque que usa la mayoría del software destinado a la resolución de modelos ILP.

En el contexto de tiempo real, dentro del análisis estático del WCET, diversos métodos pueden basarse en generar un sistema ILP que al resolverse nos proporcione el cálculo del WCET.

IPET

Como primer ejemplo encontramos el método *IPET* [LMW96] (presentado en la sección 1.2.1) que representa el análisis del WCET como un problema ILP de flujo y cuyas variables son el número de veces que se ejecuta cada bloque básico del programa. La función objetivo a maximizar es el tiempo de ejecución del programa con N bloques básicos y se expresará como:

$$\sum_{i=1}^{N} c_i \cdot x_i$$

suponiendo c_i como el coste del bloque básico y x_i la variable que determina el número ejecuciones del bloque i. Estas variables serán limitadas por las restricciones estructurales (que se derivan del CFG) y funcionales (derivadas del análisis de flujo de datos o proporcionadas manualmente, como las cotas de los bucles) del programa. A continuación, mostramos una construcción sencilla de estas restricciones a partir del ejemplo de la figura 2.2.

Cada arco en el CFG se etiqueta con la variable d_i que sirve también para contabilizar el número de veces que el programa atraviesa el arco. Las restricciones estructurales se construyen a partir del hecho de que, en cada nodo, el número de ejecuciones

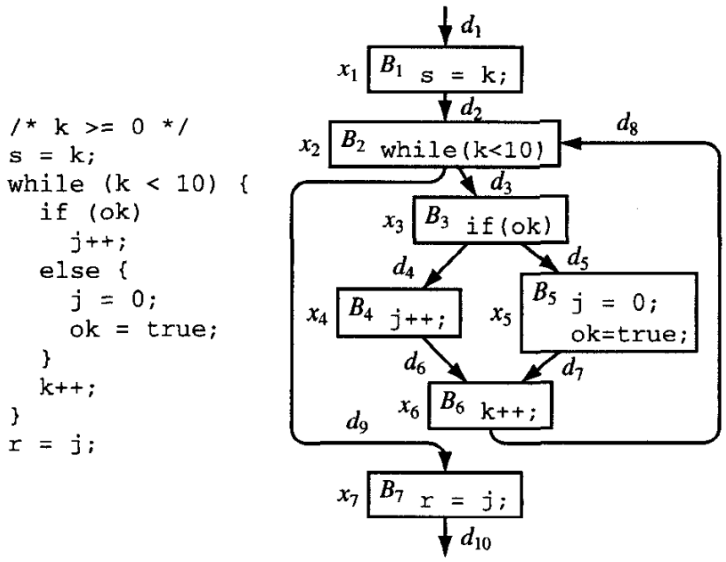


FIGURA 2.2: Se muestra el CFG (derecha) correspondiente a un fragmento de código (izquierda). Ejemplo tomado de [LMW96].

de cada bloque básico es igual el número de veces que el control entra en el nodo y también de las veces que sale. En este ejemplo las restricciones estructurales son:

$$
\begin{aligned}
d_1 &= 1 \\
x_1 &= d_1 = d_2 \\
x_2 &= d_2 + d_8 = d_3 + d_9 \\
x_3 &= d_3 = d_4 + d_5 \\
x_4 &= d_4 = d_6 \\
x_5 &= d_5 = d_7 \\
x_6 &= d_6 + d_7 = d_8 \\
x_7 &= d_9 = d_{10}
\end{aligned}
$$

Las cotas de los bucles se determinarán a través de restricciones funcionales, tal que:

$$0 \cdot x_1 \leq x_3 \leq 10 \cdot x_1$$

Y también se pueden añadir otras restricciones funcionales que complementen la información del CFG. Por ejemplo en este caso se observa que el bloque 5 ($B5$) se ejecutará como mucho una vez dentro del bucle, con lo que podemos modelar esta información de modo que:

$$x_5 \leq 1 \cdot x_1$$

Con estas restricciones modelamos única y exclusivamente los caminos del programa pero habrá que añadir nuevas restricciones que modelen la microarquitectura (CPU, caches, segmentación...) lo cual aumentará la complejidad del modelo.

Lock-MS

El método *Lock-MS* genera un modelo ILP que se debe resolver para obtener el WCET. Difiere así de los primeros métodos basados en la estructura ([CP00; CB02;

Lim+95]) que calculan el WCET atravesando de abajo a arriba una representación en árbol del CFG.

El modelo ILP que genera *Lock-MS* puede estar basado o en la estructura o en el recorrido, pero en este trabajo aplicamos el basado en la estructura. Contrasta así con el método *IPET* que representa el análisis del WCET como un problema ILP de flujo.

Lock-MS está diseñado para analizar el WCET en sistemas con una cache de instrucciones totalmente bloqueable. Así que, además de calcular el WCET, determina cual será el contenido a cargar y bloquear en la cache. Hay que tener en cuenta que este método no es compatible con *IPET*, ya que *IPET* maximiza los casos posibles de ejecución mientras que *Lock-MS* obtiene los contenidos a bloquear que minimizan el WCET. Es decir, optimiza los contenidos a bloquear en la cache de instrucciones considerando el WCET. Para modelar este comportamiento, *Lock-MS* asocia cada línea de instrucciones de memoria con una variable binaria que representa su presencia (o no) en la cache bloqueable. El coste de ejecutar la tarea a analizar se expresa como un conjunto de restricciones lineales que dependen de las variables binarias anteriores, constituyendo el modelo ILP. De este modo, resolverlo proporciona tanto una selección óptima de instrucciones a bloquear en la cache como el cálculo del WCET.

Un detalle importante a destacar es de todas las soluciones que cumplen con el modelo *IPET* solo son seguras aquellas que maximizan el WCET. Esto se debe al espacio de soluciones válidas ($[-\infty, cotasuperiorWCET]$) en el problema de maximización. Por el contrario, cualquier solución que cumpla con el modelo *Lock-MS* es segura ya que las soluciones válidas son las complementarias en el problema de minimización ($[cotasuperiorWCET, \infty]$).

Por ejemplo, una solución segura, aunque posiblemente mejorable, sería usar una cache bloqueable vacía, que proporciona una sobrestimación del WCET.

Esto implica que al usar *Lock-MS* se puede parar el *solver* ILP (software de resolución de modelos ILP) si le está costando demasiado tiempo y aún cuando el análisis no se ha completado obtendremos resultados seguros (aunque no óptimos, serán configuraciones de cache subóptimas con su WCET asociado).

Debe tenerse en cuenta que cualquier información del contexto como el camino tomado, la relación entre variables, caminos imposibles, estado del hardware (cache, buffers, predictores de salto, etc.) son problemáticos en el análisis del WCET y normalmente producen una sobrestimación del mismo. Esta es también una de las principales limitaciones de los métodos basados en la estructura. Esta información del contexto depende del hardware a analizar. Por ejemplo, se descarta usar *Lock-MS* en caches convencionales ya que la información del contexto en estos sistemas es el estado de la cache al completo.

Sin embargo, *Lock-MS* puede incorporar la información del contexto necesaria para otro hardware como una cache bloqueable más un *line-buffer* (cache de instrucciones convencional con una sola línea de cache) que es necesario para que las caches bloqueables funcionen correctamente ya que facilita capturar una gran parte del reuso espacial [AP; Pua06; PD02; Seg+12; Apa+11]. Aunque originalmente este método no contemplaba otras fuentes de latencia variable más allá de la cache de instrucciones, en posteriores estudios se ha combinado con prebúsqueda [Apa+10], cache de datos [Seg+15], e incluso se ha empleado para calcular el consumo energético [Gra+13].

Este modelo se desarrolla en el contexto de un sistema multitarea con la presencia de una cache de instrucciones bloqueable por conjuntos y por lo tanto su comportamiento es predecible y adecuado para los sistemas de tiempo real.

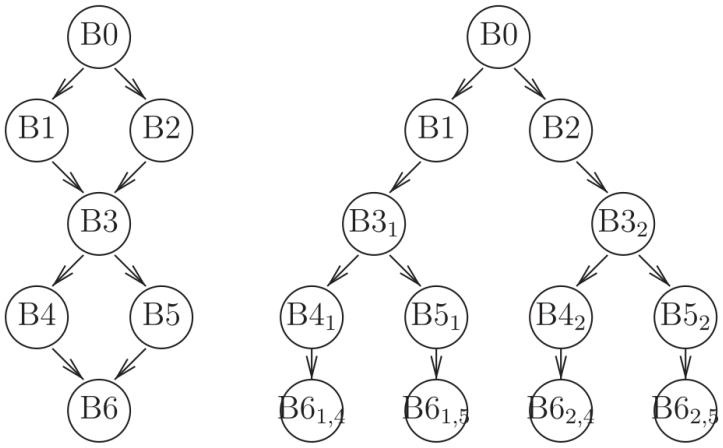


FIGURA 2.3: Grafo de control de flujo (izquierda) y su expansión explicitando caminos (derecha).

Lock-MS se plantea como un modelo de bloqueo dinámico lo que implica que el contenido previamente seleccionado de la cache se carga cada vez que una tarea comienza/continúa su ejecución en la CPU. De este modo cada tarea se beneficia del uso de la cache al completo pero añade el sobrecoste de la carga de instrucciones en los cambios de contexto. Aunque se centra en la aplicación del modelo en una única tarea, su diseño permite calcular (y minimizar) el WCET final teniendo en cuenta estos sobrecostes.

Este método plantea inicialmente que las restricciones de su modelo ILP expliciten los caminos de la tarea. Sin embargo, describir cada camino con un conjunto de restricciones implica un modelo ILP demasiado grande cuando se incrementa el número de caminos, sobretodo teniendo en cuenta que este crecimiento suele ser exponencial con respecto al número de condicionales encadenados. Cuanto mayor el modelo ILP mayor el coste en tiempo que se debe invertir para su resolución. Tomemos como ejemplo la figura 2.3 en la que se muestra un CFG y su expansión al explicitar cada uno de sus caminos, debido a la explosión combinatoria. En general, el WCET puede calcularse como el máximo de sus posibles caminos (cuatro en este caso):

$$\begin{aligned} P1 &= B0 + B1 + B3_1 + B4_1 + B6_{1,4} \\ P2 &= B0 + B1 + B3_1 + B5_1 + B6_{1,5} \\ P3 &= B0 + B2 + B3_2 + B4_2 + B6_{2,4} \\ P4 &= B0 + B2 + B3_2 + B5_2 + B6_{2,5} \\ WCET &= max(P1, P2, P3, P4) \end{aligned} \tag{2.1}$$

Teniendo en cuenta que las variables en *Lock-MS* son las líneas cacheadas, el WCET que buscamos es el mínimo WCET válido de entre todos los contenidos posibles de cache. Es decir, el mínimo de los valores que devuelve la ecuación 2.1 para cada combinación de líneas cacheadas:

$$WCET = min(max(P1, P2, P3, P4)) \tag{2.2}$$

Hay que tener en cuenta que al cargar y bloquear el contenido de la cache el coste de ejecutar cada bloque básico es independiente del coste de los bloques previamente ejecutados. Esta idea puede extenderse también al *line-buffer* ya que su comportamiento depende de la instrucción ejecutada previamente la cual es conocida dado

un camino determinado. Por lo tanto, todos los Bi_{path} se pueden sustituir por Bi y los bloques comunes se pueden analizar por separado. Si una línea de memoria está bloqueada y cacheada, su coste de búsqueda será siempre el de un acierto en cache. Sin embargo el coste debe ser fallo en el *line-buffer* para la primera referencia de cada línea accedida. Los costes de búsqueda de líneas de memoria que no están en cache y son compartidas por varios bloques básicos son estudiados en profundidad en este método. En resumen, esto permite reescribir la ecuación 2.2 como:

$$WCET = min(B0 + B3 + B6 + max(B1 + B4, B1 + B5, B2 + B4, B2 + B5)) \quad (2.3)$$

De la ecuación 2.3 podemos obtener expresiones equivalentes a la función de maximización de la siguiente manera. Si $B1 > B2$, entonces $B1 + B4 > B2 + B4$, por lo tanto es obvio que $B2 + B4$ puede descartarse ya que no formará parte del peor caso. De lo contrario, $B1 + B4 \leq B2 + B4$ y $B2 + B4$ debe permanecer en la expresión. Así maximizando primero el par $B1$, $B2$ y después $B4$, $B5$ tenemos:

$$max(B1 + B4, B1 + B5, B2 + B4, B2 + B5) = max(B1, B2) + max(B4, B5) \quad (2.4)$$

Por lo tanto, podemos reescribir la ecuación 2.3 como:

$$WCET = min(CosteComun + max(B1, B2) + max(B4, B5)) \quad (2.5)$$

Esta información podría traducirse a un árbol en el que el coste de cada nodo fuera la suma del coste común más el máximo coste de entre sus nodos hijos. Es decir, cada una de las ramas sería un camino alternativo. Como se puede ver la diferencia entre las expresiones de las ecuaciones 2.1 y 2.5 es que la primera, expresada por caminos, requiere de muchas más restricciones que la segunda, con estructura de árbol. Sin embargo, este método no proporciona el algoritmo para construir la estructura en árbol necesaria para establecer las restricciones del modelo ILP.

2.2. Caracterización de benchmarks

En esta sección presentamos aquellos programas de prueba que utilizamos durante nuestra investigación.

2.2.1. Suites de benchmarks para sistemas de tiempo real

Una de las principales cuestiones de la mayoría de los trabajos de investigación de nuestra área es qué programas de prueba (*benchmarks*) van a utilizarse para medir los resultados de la propuesta de dicho trabajo. Esto es especialmente útil en el área de los sistemas de tiempo real dónde es crucial evaluar y comparar técnicas de análisis del WCET, de los compiladores y de arquitectura de computadores. Por ello, es muy útil tener un conjunto de benchmarks (llamados *suites* de benchmarks) que se encuentren disponibles con facilidad, que hayan sido probados, que estén bien documentados, que sean fieles a los programas que se están usando en sistemas de tiempo real en la industria y que tengan un reconocimiento de la comunidad investigadora que permita establecer comparaciones en la evaluación de distintos algoritmos, métodos y herramientas.

Todos estos motivos son los que nos han llevado a escoger las dos suites de benchmarks que presentamos a continuación:

```
for (i=0;i<n;i++)
  for (k=0;k<n;k++) {
    t=B[i][k];
    for (j=0;j<n;j++)
      A[i][j]=A[i][j]+t*C[k][j];
  }
```

FIGURA 2.4: Algoritmo del benchmark *matmult_opti*.

- **Mälardalen** [Gus+10] es la primera colección de programas especialmente dirigido para las herramientas de análisis del WCET, con un enfoque en el análisis del flujo del programa. Se creó en 2005 recopilando programas de distintas fuentes y desde entonces se ha usado en muchas investigaciones del WCET. La mayoría de sus benchmarks son relativamente pequeños y de un solo camino, lo cual limita su aportación a la hora de evaluar herramientas que admiten códigos de caminos múltiples.

- **TACLeBench** [Fal+16] proporciona un conjunto de benchmarks gratuitos disponibles y relacionados con temas de investigación. Sus códigos son autocontenidos; no existen dependencias de cabeceras específicas del sistema a través de *#include* o del sistema operativo. Todos los datos de entrada forman parte del código fuente en C. Esto hace que la colección TACLeBench sea útil para sistemas empotrados donde no hay bibliotecas estándar disponibles. Como uno de los objetivos de la creación de TACLeBench es abordar las necesidades que requieren las herramientas de análisis temporal, todos los benchmarks contienen anotaciones sobre información del flujo de datos (por ejemplo, el número máximo de iteraciones de los bucles).

La tabla 2.1 muestra los benchmarks que se han utilizado en nuestros experimentos de los capítulos 3 y 4 que se descargaron en febrero de 2017 de los paquetes TACLeBench [Fal+16] y Mälardalen [Gus+10], más el benchmark *matmult_opt* que ejecuta una multiplicación de matrices optimizada (figura 2.4).

Algunos de los benchmarks de estos paquetes se han descartado por las siguientes razones: errores de compilación en compiladores cruzados[1], número desconocido de iteraciones de bucles en funciones de bibliotecas al compilar con coma flotante emulada[2] y problemas con la extracción del CFG[3]. Estos problemas de extracción incluyen construcciones «*switch*» implementadas mediante saltos a direcciones desconocidas, CFGs con bucles irreducibles (por ejemplo, bucles con múltiples entradas), funciones recursivas, etc. Estas cuestiones provienen de limitaciones en el procesamiento del código binario, pero no afectan a nuestras propuestas.

2.2.2. Transposición de matrices

La trasposición de una matriz es una operación que consiste en colocar sus filas en forma de columna, respetando su orden. En la figura 2.5 podemos encontrar un ejemplo de la transposición de una matriz de 3×3 elementos. Esta operación aparentemente sencilla es fundamental en áreas como el álgebra lineal o las transformadas

[1] *powerwindow, bitcount, gsm_dec, rijndael_dec, dijndael_enc, susan.*

[2] *powerwindow, prime, adpcm_dec, adpcm_enc, ammunition, anagram, cjpeg_transupp, cjpeg_wrbmp, epic, huff_enc, rijndael_dec, rijndael_enc.*

[3] *sha, gsm_enc, h264_dec, cover, duff, mpeg2, lms, test3, quicksort, recursion.*

TABLA 2.1: Benchmarks usados en nuestros experimentos (TACLe-Bench [Fal+16] y Mälardalen [Gus+10]).

Nombre	Suite
audiobeam	TACLeBench
basicmath	TACLeBench
binarysearch	TACLeBench
bsort	Mälardalen
bs	Mälardalen
cnt	Mälardalen
complex_updates	TACLeBench
countnegative	TACLeBench
crc	Mälardalen
dijkstra	TACLeBench
fdct	Mälardalen
fft	TACLeBench
filterbank	TACLeBench
fir2dim	TACLeBench
fmref	TACLeBench
g723_enc	TACLeBench
iir	TACLeBench
janne_complex	Mälardalen
jfdctint	TACLeBench
lift	TACLeBench
ludcmp	Mälardalen
matmult	Mälardalen
matmult_opti	Propio
matrix1	TACLeBench
md5	TACLeBench
minver	TACLeBench
nsichneu	Mälardalen
ndes	Mälardalen
petrinet	TACLeBench
pm	TACLeBench
qsort-exam	Mälardalen
qurt	Mälardalen
select	Mälardalen
st	TACLeBench
statemate	Mälardalen

de Fourier, entre otras. Además tiene muchas aplicaciones en otras como el análisis numérico, el procesado de imágenes y gráficos. Hay muchos ejemplos de aplicaciones desarrolladas en los centros de supercomputación que usan, de un modo u otro, la transposición de matrices como parte esencial para solucionar sus problemas. Por ejemplo, el *Oak Ridge National Laboratory* (EEUU) desarrolla, mantiene, prueba y gestiona *SCALE Code System* [RJ16] que es un conjunto de benchmarks de modelado y simulación ampliamente usado para el diseño y análisis de seguridad nuclear. Otro ejemplo que encontramos es *Geotess* [Bal+16] desarrollado por *Sandia National Laboratory*. Este sistema de soporte software y parametrización de modelos implementa la construcción, almacenamiento y consulta de los datos pertenecientes a modelos 3D de la Tierra.

En el área de tiempo real encontramos que el patrón de accesos del algoritmo es determinista y de fácil comprensión, por lo tanto su estudio puede servir de base para analizar otros algoritmos con patrones de acceso más complicados. Sin embargo, su análisis en el entorno de tiempo real no es trivial ya que aparecen muchos

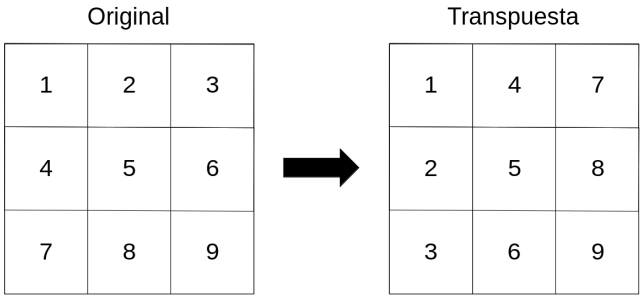


FIGURA 2.5: Ejemplo de transposición de una matriz de 3×3 elementos.

parámetros a tener en cuenta: el tamaño de la matriz y todos los relacionados con la estructura de la memoria cache (tamaño, asociatividad, número de conjuntos, tamaño de bloque...). Además, esta operación tiene varias propuestas de implementación que varían el orden de acceso de los elementos de la matriz alterando la eficacia del uso de la cache de datos y, por tanto, la eficacia del sistema.

Capítulo 3

Reducción del WCET y del tiempo de análisis en sistemas con caches bloqueables de instrucciones

En este capítulo tratamos uno de los retos clave en los sistemas de tiempo real que es el análisis de la jerarquía de memoria. Muchos métodos de análisis del WCET que admiten una cache de instrucciones se basan en algoritmos iterativos o convergentes los cuales son bastante lentos. Nuestro objetivo en este capítulo es reducir el tiempo de análisis del WCET en sistemas con una cache de instrucciones bloqueable, centrándonos en el método *Lock-MS*. Primero, proponemos un algoritmo para obtener una representación basada en la estructura del grafo de control de flujo. Este algoritmo organiza el problema del WCET como un conjunto de subproblemas anidados, el cual aprovecha los algoritmos habituales de ramificación y poda de los *solvers* de ILP. Después, añadimos al algoritmo la posibilidad de determinar varios puntos de bloqueo en cada tarea, cada uno con un contenido específico de cache, en vez de bloquear un solo contenido para la ejecución de toda la tarea. Los puntos de bloqueo se establecen heurísticamente antes de los bucles exteriores. Esta heurística es tan simple que no añade complejidad y reduce el WCET aprovechando el reuso temporal que encontramos en los bucles. Debido a que los bucles pueden procesarse como regiones aisladas, para cada región se puede obtener el contenido óptimo a bloquear en la cache y el tiempo de análisis del WCET se verá notablemente reducido. Con estas dos mejoras nuestro análisis del WCET resulta un orden de magnitud más rápido que otras propuestas. Además, nuestros resultados muestran que la tasa de aciertos que se alcanza para la cache de instrucciones bloqueada es similar a una ejecución real con una cache de instrucciones LRU. Finalmente, analizamos la susceptibilidad a las optimizaciones del compilador, mostrando cual es la elección correcta para cada programa de prueba y señalando que O0 es siempre la peor opción.

3.1. Introducción

Uno de los principales retos en el análisis del WCET es la jerarquía de memoria [Apa+08]. El comportamiento convencional de la cache depende de las referencias pasadas y, para que el análisis sea preciso, será necesario conocer todos los accesos de memoria previos para determinar la latencia de un determinado acceso a memoria. Si nos centramos en el reemplazo LRU, los métodos estáticos actuales de análisis del WCET se basan en *AbsInt* [CC77; FW99], *IPET* [LMW96], o el uso de ambos [Tra]. Dado el considerable tiempo de análisis que estos métodos precisan para

analizar el WCET en sistemas con una cache de instrucciones, no está claro si pueden analizar programas complejos en sistemas que incluyan otros componentes de hardware como la cache de datos, la prebúsqueda, etc. Por ejemplo, aunque teóricamente tanto *AbsInt* como *IPET* permiten caches de datos [LMW96], ningún estudio las ha evaluado a conciencia hasta donde sabemos.

Para reducir el tiempo de análisis muchos estudios han propuesto usar caches completamente bloqueables [Mit16]. Estas caches se pueden encontrar en procesadores de la mayoría de fabricantes como Motorola (ColdFire, PowerPC, MPC7451, MPC7400), MIPS32, ARM (904, 946E-S), Integrated Device Technology (79R4650, 79RC64574), Intel 960, etc. En caso de fallo estas caches piden la línea que ha resultado en fallo al siguiente nivel de memoria pero cuando llega se envía a un *line-buffer* (memoria intermedia con capacidad de almacenar una línea de cache) sin guardar ninguna copia en la cache. Por lo tanto, no será necesario realizar ningún reemplazo y todo el almacenamiento y control dedicado a su implementación en las caches convencionales se suprime al carecer de utilidad. Debido a que el contenido de la cache bloqueable es conocido y no cambia, el cálculo de aciertos y fallos es mucho más fácil y no depende de los accesos anteriores a memoria, de este modo se simplifica el análisis del WCET. Sin embargo, el reto que presentan estas caches es determinar qué conjunto de instrucciones será el mejor para bloquear en la cache, además de llevar a cabo su análisis del WCET. Por lo tanto, los métodos de caches bloqueables tratan de encontrar qué contenidos deben ser bloqueados en la cache para generar el mínimo WCET posible. Dependerá de su flexibilidad en cuanto a los puntos de carga y bloqueo, de los distintos conjuntos de contenidos a gestionar, y también de cómo se aborde el análisis (heurísticamente, analíticamente, etc.). Hay muchos métodos para abordar este problema.

Los métodos de «bloqueo estático» seleccionan un solo conjunto de instrucciones para bloquear durante todas las tareas que se ejecutan en el sistema, por lo tanto esta selección se fija cuando el sistema se inicia [PD02]. Por otro lado, los métodos de «bloqueo dinámico» seleccionan uno o más conjuntos de instrucciones para cada tarea. En general, el bloqueo dinámico funciona mejor que el estático en cuanto al WCET [Cam+03]. Centrándonos en el bloqueo dinámico, nos referiremos como *bloqueo dinámico de un solo contenido* a aquellos métodos que seleccionan un solo contenido por tarea, el cual se carga y bloquea en el cambio de contexto de la tarea correspondiente (por ejemplo [Apa+11]), y *bloqueo dinámico de contenido múltiple* a aquellos métodos que permiten a cada tarea cargar y bloquear contenidos durante su ejecución en múltiples ocasiones(por ejemplo [Pua06]). Dos propiedades muy interesantes del bloqueo dinámico de un solo contenido son que, primero, se puede llevar a cabo el análisis del WCET con métodos basados en la estructura (cuya solución es mucho más rápida) sin perder precisión, y segundo, que estos métodos también proporcionan la selección óptima de contenidos a bloquear [Apa+11]. Esto permite extender el análisis del WCET de manera que incluya la prebúsqueda [Apa+10], cache de datos [Seg+12; Seg+15], e incluso poder analizar al mismo tiempo el consumo de energía para obtener una solución equilibrada que tenga en cuenta tanto el WCET como el WCEC [Gra+13]. Por otro lado, los métodos dinámicos de contenido múltiple mejoran el WCET pero se añade la dificultad de decidir cuáles serán los mejores lugares del código en los que fijar la carga y bloqueo de instrucciones, y cuales serán estas instrucciones en cada punto de carga.

Normalmente estos dos problemas se abordan heurísticamente para intentar limitar el tiempo de análisis, así que sus resultados no son óptimos. Además el tiempo de análisis que necesitan sigue siendo comparable al necesario para realizar el análisis del WCET en una cache LRU [AP; Pua06]. Otros estudios usan algoritmos

genéticos para intentar resolver estos problemas [CIM01].

Por último, algunos estudios suponen *caches parcialmente bloqueables a nivel de conjunto*. En cada conjunto de estas caches puede haber un número variable de líneas no bloqueadas ordenadas en LRU y el resto de líneas del conjunto estarán bloqueadas [DLM13; ZWY17]. La complejidad del control y almacenamiento para implementar este método sobrepasa las capacidades de las caches convencionales y, por supuesto, de las caches completamente bloqueables. Hasta donde sabemos, este hardware todavía no está disponible y los diseños actuales de cache están bastante alejados de alcanzar este comportamiento. Además se necesita mucho tiempo de análisis para el gran número de configuraciones que admite una cache parcialmente bloqueable a nivel de conjunto. Por ejemplo, se ha propuesto un proceso convergente que consiste en dos fases [ZWY17]. En la primera se realiza el análisis del WCET asumiendo una cache de instrucciones que puede tener algunas líneas bloqueadas. En la segunda fase se prueba si hay una nueva línea adecuada para bloquear, configurando la cache de instrucciones de acuerdo con la próxima iteración del algoritmo de convergencia. Sin embargo, ninguno de estos estudios proporciona unos puntos de referencia que sean independientes del sistema (por ejemplo, situación de siempre acierto o de siempre fallo) lo cual dificulta interpretar sus resultados. Además no se comparan con las caches convencionales y cuando se comparan con caches completamente bloqueables usan un hardware sesgado ya que no consideran el *line-buffer* que es necesario para que las caches bloqueables funcionen correctamente [Apa+10; Apa+11; AP; Pua06; PD02; Seg+12; Seg+15]. Sin embargo, en este capítulo nos centramos en estructuras de cache simples y métodos de análisis del WCET rápidos, así que las soluciones a través de caches bloqueables parcialmente a nivel de conjunto se encuentran fuera de nuestro ámbito.

Como hemos afirmado anteriormente, es tan importante mejorar la velocidad del análisis que muchas veces los métodos heurísticos son preferibles a los métodos analíticos. Sin embargo, los métodos de análisis orientados a caches bloqueables no han explorado todavía a conciencia el potencial de estos sistemas para obtener un análisis rápido. Nuestro objetivo en este capítulo es reducir el tiempo de análisis del WCET de tareas en presencia de caches de instrucciones bloqueables. Esto se logrará desarrollando *DLock-MS*, un método de bloqueo dinámico de contenido múltiple. Básicamente consiste en añadir dos mejoras claves a *Lock-MS* [Apa+11], un método de bloqueo dinámico de un solo contenido. Este método ha sido detallado más profundamente en las sección 2.1.2

La primera de estas mejoras es un algoritmo que traduce el CFG a una estructura en árbol que representa el problema de análisis del WCET, la cual permite usar *Lock-MS* como un método basado en la estructura. Este tipo de métodos son en general los más rápidos, ya que no usan algoritmos de convergencia ni superponen problemas de flujo. Este algoritmo organiza el problema del WCET como un conjunto de subproblemas anidados, el cual aprovecha los algoritmos habituales de ramificación y poda de los *solvers* (softwares de resolución de modelos ILP). La programación lineal en enteros da respuesta a situaciones en las que se exige maximizar o minimizar funciones que se encuentran sujetas a determinadas restricciones, y cuyas variables de decisión deben ser enteras. Los *solvers* una vez definidas las funciones y restricciones resuelven esta maximización o minimización de las funciones aplicando las restricciones correspondientes. Al haber hecho una división en subproblemas cada uno se optimiza para conseguir una resolución lo suficientemente rápida. En términos de eficiencia, nuestro algoritmo genera una estructura en árbol en una única pasada y explora cada rama solo una vez.

Nuestra segunda mejora trata la limitación de tamaño que presentan los métodos

Algoritmo 1 Explorar(*nodoCFGactual*, *caminoActual*)

```
1: if |hijos(nodoCFGactual)| = 0 then                                  # no hay más nodos en el camino
2:    return caminoActual + nodoCFGactual
3: else if |hijos(nodoCFGactual)| = 1 then                             # hijo único: expandir camino
4:    return Explorar(hijo(nodoCFGactual), caminoActual + nodoCFGactual)
5: else if |hijos(nodoCFGactual)| > 1 and explorado[nodoCFGactual] then
6:    return caminoActual + nodoCFGactual                              # condicional ya explorado
7: else                                  # condicional no explorado (|hijos(nodoCFGactual)| > 1)
8:    for all nodoHijo ∈ hijos(nodoCFGactual) do                       # procesar cada camino alternativo
9:       caminoAlternativo ← Explorar(nodoHijo, nodoCFGactual)
10:      procesarYConstruirRestricciones(nodoCFGactual, caminoAlternativo)   # establecer restricciones
   como C_BBx ≥ caminoAlternativo
11:   end for
12:   explorado[nodoCFGactual] ← true
13:   return caminoActual + nodoCFGactual      # todos los caminos alternativos han sido procesados
14: end if
```

de bloqueo dinámico con un único contenido, no solo para sobrepasar esa limitación consiguiendo mejores resultados en el WCET sino también teniendo en cuenta el tiempo y eficiencia del análisis. *DLock-MS* aplica heurísticas basadas en bucles para seleccionar el emplazamiento de los puntos múltiples de carga y bloqueo para la cache de instrucciones. Después permite al *solver* encontrar los contenidos óptimos para cargar y bloquear en cada punto. Además las regiones dónde permanecen fijos cada uno los contenidos bloqueados se pueden procesar como subproblemas aislados, lo cual acelera aún más el análisis del WCET y su resolución. Esto nos daría la posibilidad también de calcular cada una de las regiones en paralelo (no lo abordamos en este trabajo).

El resto del capítulo lo organizamos de la siguiente manera. Nuestras dos propuestas se describen en las secciones 3.2 y 3.3. En la sección 3.4 evaluamos las propuestas y, finalmente presentamos nuestras conclusiones en la sección 3.5.

3.2. Transformación de CFG a árbol

En esta sección presentamos la primera contribución de este capítulo, un algoritmo que traduce el CFG a una estructura en árbol que representa el problema de análisis del WCET, la cual permite usar *Lock-MS* como un método basado en la estructura.

Comienza sustituyendo los bucles y las funciones que no sean el programa principal por nodos virtuales (bloques básicos) en el CFG. Después, se procesan como *sub-CFGs* independientes para transformarlos en árboles. El Algoritmo 1 se aplica recursivamente tanto en el CFG principal como en cada uno de los *sub-CFGs* independientes, comenzando desde el nodo de entrada del CFG correspondiente y un camino vacío (*Explorar(entrada,∅)*). Este algoritmo lleva a cabo una búsqueda recursiva en profundidad que construye los árboles asociados a cada CFG y genera sus correspondientes restricciones ILP conforme al modelo *Lock-MS*. Básicamente, cada árbol se compone de un nodo condicional más todos sus caminos alternativos hasta que se alcanzan otro nodo condicional. El Algoritmo 1 funciona de la siguiente manera: Las líneas 1 y 2 tratan los nodos finales del CFG, devolviendo el camino en curso más el nodo actual (y final), por lo tanto completan la exploración de ese camino. En las líneas 3 y 4, que corresponden a un nodo con un solo hijo, continúa la exploración siguiendo el camino de este hijo único. Las líneas 5 y 6 corresponden a la exploración de un nodo condicional, es decir que tiene más de un hijo, que

ya ha sido explorado. En consecuencia, ya se ha llevado a cabo su procesamiento y la generación de sus restricciones correspondientes siendo innecesario continuar su exploración. Finalmente, en las líneas 7 a 13 se describe como proseguir cuando el nodo actual es condicional y todavía no se ha explorado. En este caso, se genera un árbol nuevo, con el nodo actual como su raíz. Cada uno de sus hijos se explorará para construir las ramas (caminos alternativos) de esta raíz (líneas 8 y 9). A continuación en la línea 10, se procesa cada una de las ramas y se establecen sus restricciones ILP correspondientes (tal como explicamos más adelante). Finalmente, el nodo actual se marca como explorado, devolviendo el camino hasta este nodo (líneas 12 y 13). Hay que tener en cuenta que el nodo de entrada de un bucle no se considera hijo de ningún nodo que sea interno del propio bucle, ya que los arcos que los puedan relacionar son aquellos responsables de la repetición del bucle.

Básicamente, la restricciones ILP (línea 10, Algoritmo 1) modelan un problema de minimización para un árbol de nodos (bloques básicos, BB). El coste de cada nodo en particular BBi corresponde al coste total acumulado de ejecutar ese nodo, lo que resulta en una expresión dependiente de sus diferentes casos de ejecución y su número de instancias. Para cada línea de memoria que puede estar en cache j en un bloque básico, se asocia una variable $enCacheBBi_j$ que determina si está o no en cache para reducir el WCET. Esta líneas de memoria en cache no pueden aumentar más allá de la capacidad misma de la cache, así que también están restringidas según el número de conjuntos y vías.

Por ejemplo, asumiendo una cache bloqueable y un bloque básico BB1 que quepa en una única línea de cache L1, sus costes se establecen como:

$$BB1 = nEjecsBB1 \cdot CosteBB1$$

$$= nEjecsBB1 \cdot (costeAcierto \cdot enCacheL1 + costeFallo \cdot (1 - enCacheL1) + ejecInstBB1)$$

, donde *costeAcierto* y *costeFallo* son constantes previamente calculadas en base a los parámetros de hardware, *ejecInstBB1* coste de ejecución de las instrucciones del bloque básico, *nEjecsBB1* depende del CFG, y *enCacheL1* es una variable binaria (0/1).

Sin embargo, en este trabajo nos centramos en la estructura de las restricciones generales que modelan el CFG y no en aquellas que modelan el hardware [Apa+10; Apa+11; Gra+13; Seg+12; Seg+15].

Tomamos como ejemplo la figura 3.1 que muestra un CFG con 12 caminos explícitos que hay considerar en el análisis del WCET, los árboles que se obtienen tras su transformación y sus principales restricciones ILP correspondientes. Señalar que con una cache bloqueable el peor tiempo de ejecución de un bucle no puede incluir combinaciones de caminos alternativos en su interior [Apa+11]. Por lo tanto los 12 caminos a considerar ejecutan lo siguientes bloques básicos: 1-2-4-5-10-11-13, 1-2-4-5-10-12-13, 1-3-4-5-10-11-13, 1-3-4-5-10-12-13, 1-2-4-6-7-9-10-11-13, 1-2-4-6-7-9-10-12-13, 1-2-4-6-8-9-10-11-13, 1-2-4-6-8-9-10-12-13,1-3-4-6-7-9-10-11-13, 1-3-4-6-7-9-10-12-13, 1-3-4-6-8-9-10-11-13, y 1-3-4-6-8-9-10-12-13. El Árbol A representa el último de los condicionales, en BB10, cuyo coste será el máximo de sus dos ramas alternativas. El Árbol B representa del bloque BB4 a BB10, este último ya ha sido explorado (Árbol A). El bucle que empieza en BB6 se analiza como un CFG independiente con sus propio árbol (Árbol Bucle) y se representa como un nodo virtual ($Bucle_{BB6}$) en el Árbol B. Por último, el Árbol C representa del bloque BB1 al BB4. Por lo tanto en vez de 12 caminos en el CFG, nuestra propuesta proporciona 4 sub-árboles con 2 sub-caminos cada uno.

Se puede establecer el WCET como el coste de todo el árbol, es decir, el coste de su raíz: $WCET = C_{BB1}$. A su vez, el coste de cada árbol (compuesto por los costes

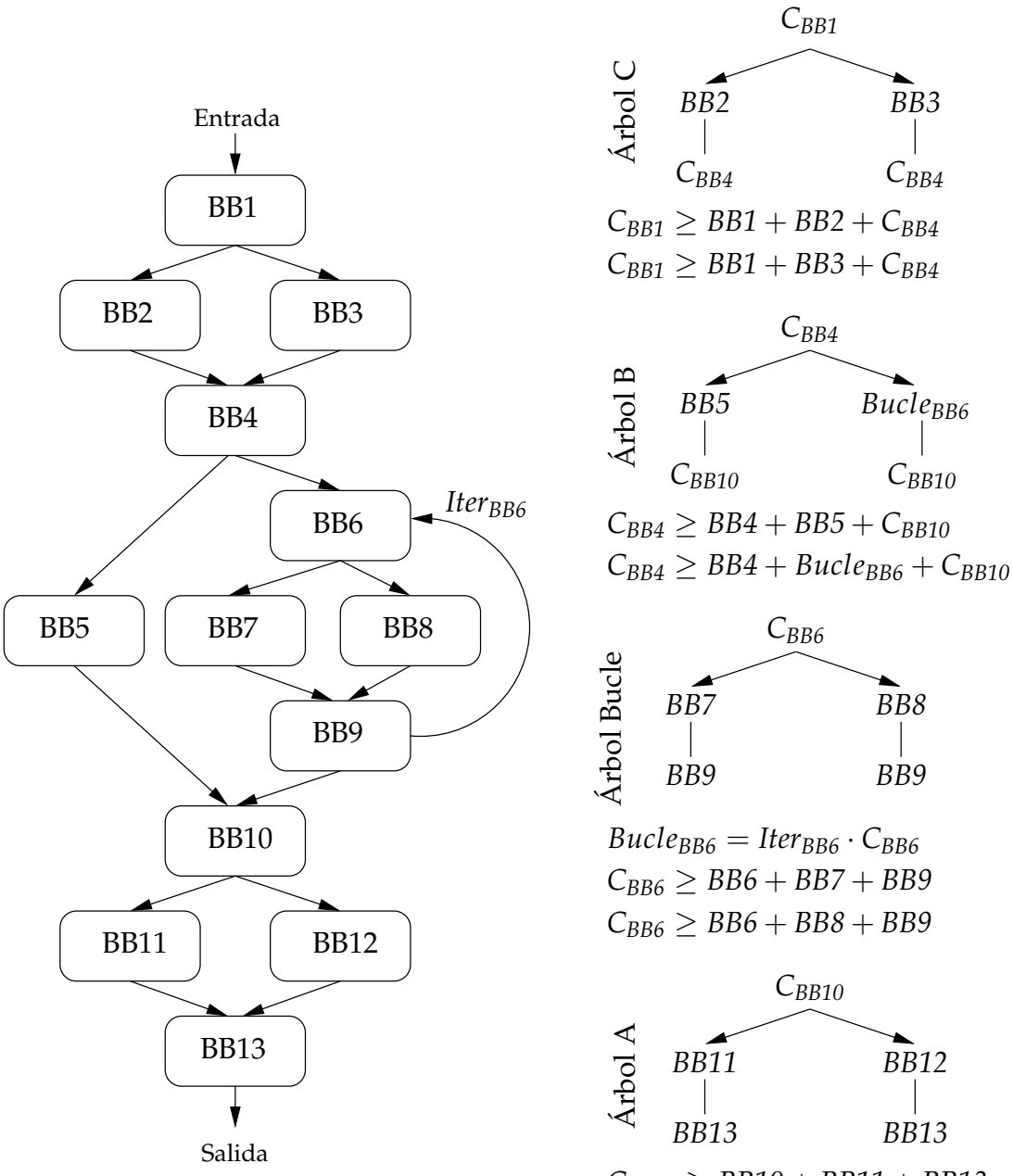


FIGURA 3.1: Ejemplo de un CFG simple, sus árboles asociados, y sus restricciones ILP.

de sus nodos y subárboles) debe ser mayor o igual que el coste de cualquiera de sus caminos alternativos, tal como se detalla en la figura 3.1. Por tanto, minimizando la variable *WCET*, el ILP *solver* proporciona los mejores valores para la variables $enCacheBBi_j$, es decir, aquellos que producen el mínimo WCET.

Además de la propia transformación de CFG a árbol, nuestra propuesta tiene los siguientes beneficios: todo el proceso se lleva a cabo en una sola pasada, sin necesidad de hacer un procesamiento previo para saber el tamaño del CFG. Las ramas, que también se exploran solo una vez, están asociadas normalmente a los bloques básicos. Esto implica que nuestra propuesta es básicamente lineal con respecto al

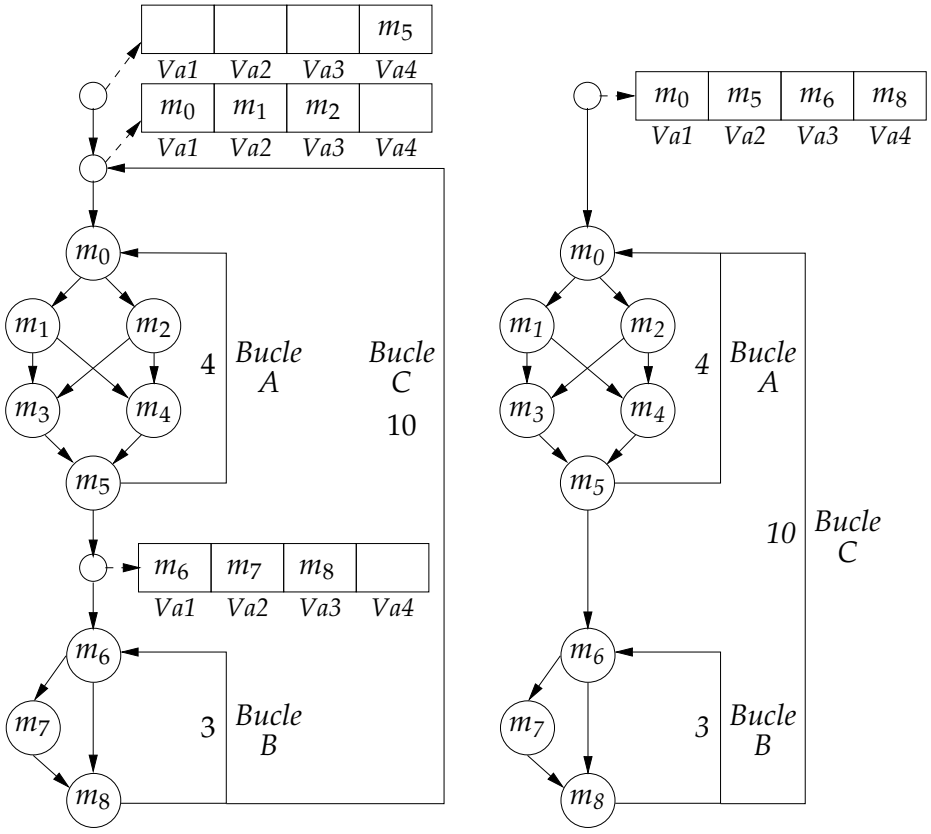
número de bloques básicos en el programa. Solo los bloques básicos terminales pueden acabar siendo explorados más de una vez (por ejemplo, el bloque BB9 donde termina el bucle y el BB13 donde termina el CFG en la figura 3.1).

3.3. Emplazamiento de los puntos de carga y bloqueo

La principal desventaja de los métodos de bloque dinámicos de un solo contenido como *Lock-MS* es su incapacidad para adaptarse a los cambios que pueden aparecer durante la ejecución del programa. Por lo tanto, una mejora aparentemente directa sería definir contenidos a cargar y bloquear para regiones específicas previamente a la ejecución de tales regiones en la tarea [AP; Cam+05; Pua06]. Sin embargo, esto conlleva que debemos encontrar tanto qué puntos de carga como qué líneas de instrucciones a bloquear en cada punto serán los adecuados, teniendo en cuenta el tiempo de sobrecoste por cargar estas instrucciones y modificar el contexto de la distribución de elementos en memoria de cada tarea. Por consiguiente, mejorar el comportamiento de un bloqueo dinámico de un solo contenido óptimo no es tan directo como podría parecer en primera instancia. De hecho, los estudios que afirman obtener mejores resultados que con el bloqueo dinámico de un solo contenido indican específicamente que han quitado el componente de hardware *line-buffer*, que es necesario para que las caches bloqueables funcionen adecuadamente [DLM13].

Con el objetivo de abordar esta problemática proponemos una heurística de bloqueo dinámico la cual será la segunda contribución de este capítulo. Esta heurística es bastante simple ya que determina como puntos de carga y bloqueo de la cache de instrucciones las entradas a los bucles exteriores. Primero, cada región del programa se beneficia de un contenido privado bloqueado en cache. Esto significa que nuestro análisis encuentra los contenidos que minimizan el WCET para cada región. Segundo, no hay puntos de carga y bloque dentro de los bucles, por lo tanto su sobrecarga correspondiente no se verá multiplicada por las iteraciones del bucle. Tercero, ya que las memorias cache bloqueables explotan el reuso temporal, y este se encuentra en los bucles, nuestra heurística permite usar toda la capacidad de la cache para cada uno de estos entornos de reuso temporal. Debemos señalar que esta heurística no impide que se establezcan otros puntos de carga. Esto es, el modelo ILP que se obtiene se puede modificar si quien diseña quiere cambiar su localización.

Para tener una visión más práctica, comparamos nuestra propuesta con otros enfoques. La figura 3.2(a) reproduce el ejemplo presentado en un estudio reciente que muestra la mejora de su propuesta con respeto a trabajos previos (Fig. 3 en [ZWY17]). Como se puede ver, establece tres puntos de carga y bloqueo con contenido específico (flechas discontinuas). En cambio, nuestra propuesta (figura 3.2(b)) establece solo un punto de carga y bloqueo con los contenidos óptimos para el bucle externo. Para poder llevar a cabo un cálculo del WCET que sea simple, contamos solo los fallos y aciertos considerando que cada línea de cache cargada equivale a un fallo (M_i). En los puntos de carga y bloqueo se asume una llamada a función con sus correspondientes instrucciones ejecutadas y fallos en cache [ZWY17]. Así que, para que nuestra propuesta parta con desventaja, consideramos que el coste de cada punto de carga es únicamente un fallo (M_p). En tal caso, el WCET que se obtendría en la figura 3.2(a) es: $(M_p + M_5) + 10 \cdot ((M_p + M_0 + M_1 + M_2) + 4 \cdot (3H + 1M) + (M_p + M_6 + M_7 + M_8) + 3 \cdot (3H)) = 122M + 210H$. Por otra parte, con nuestra propuesta (figura 3.2(b)) el WCET que se obtendría es: $(M_p + M_0 + M_5 + M_6 + M_8) + 10 \cdot (4 \cdot (2H + 2M) + 3 \cdot (2H + 1M)) = 115M + 140H$. Por lo tanto, nuestra propuesta proporcionaría un WCET significativamente menor (un 5,7 % menos de fallos y



(a) Zheng et al. (b) Nuestra propuesta

FIGURA 3.2: Bloques básicos bloqueados y puntos de carga de la propuesta de Zheng et al. y de nuestra propia propuesta tomando el ejemplo expuesto en [ZWY17].

un 33,3 % menos de aciertos). No obstante, es importante recordar que la heurística propuesta en la figura 3.2(a) necesita un cache parcialmente bloqueable a nivel de conjunto mientras que en la nuestra asume solo una completamente bloqueable. Debido a que el hardware es completamente diferente nuestros métodos no son comparables a no ser que toda la cache este bloquedada, como en la figura 3.2. También, hay que tener en cuenta que otros enfoques deben realizar análisis previos o convergentes para establecer sus puntos de carga, lo cual implica tiempos de análisis mucho más largos [AP; ZWY17].

Como hemos indicado antes, cada bucle externo se asocia a una región (con contenidos de cache específicos y fijados) la cual se analiza de manera aislada. La figura 3.3 muestra un ejemplo con cinco bucles, los exteriores marcados como regiones. Las instrucciones en bloques básicos (nodos) que no pertenecen a ninguna región (instrucciones en BB1, BB7 y BB8) no son candidatas a ser bloqueadas, pero serán cacheadas de todos modos por el *line-buffer* [PD02]. De esta manera, cada región puede verse como un problema independiente de bloque dinámico de un solo contenido. Este enfoque asegura que, para cada punto de carga, los contenidos de cache seleccionados son óptimos. Esto es, el ILP *solver* proporciona la selección de contenidos para cada punto de carga de manera que el coste de ejecución para la región correspondiente en el peor caso es el mínimo posible. Además el número de caminos a explorar se ha reducido todavía más. Descartando los bucles, el número de caminos a explorar en la figura 3.3 si no aplicaramos las regiones sería $3 \times 2 \times 2$. Sin embargo, como los análisis de las regiones son independientes entre sí, con nuestra heurística

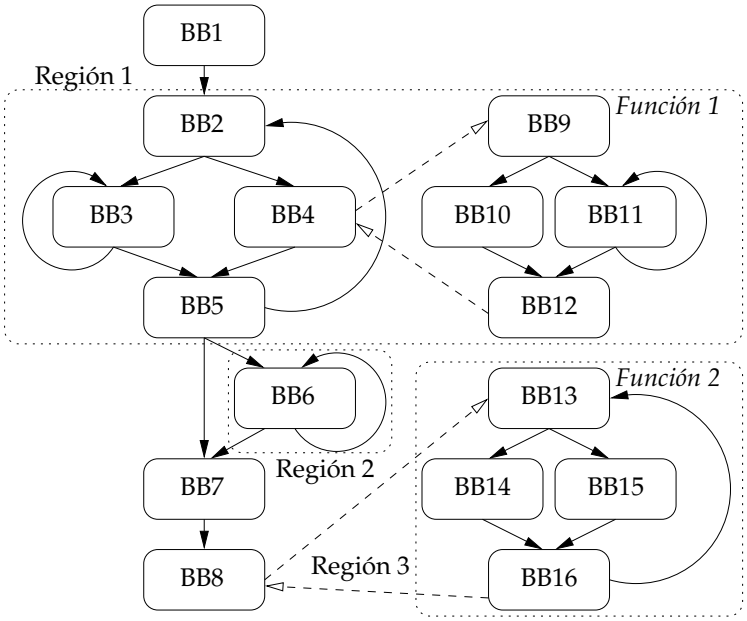


FIGURA 3.3: Ejemplo de CFG con las regiones consideradas en *DLock-MS*.

de bloqueo dinámico el número de caminos a explorar es $3 + 2 + 2$. Es importante destacar también que tener regiones independientes permite generar un problema ILP para cada región, resolverlos en paralelo, y usar sus resultados (el WCET parcial de cada región) como el coste específico de cada región el modelo general de ILP. Hasta donde sabemos, ningún otro método ha mostrado tal potencial. En este trabajo no exploramos este paralelismo pero reduciría aún más el tiempo de análisis.

3.4. Resultados

En esta sección evaluamos *DLock-MS*, nuestra extensión de *Lock-MS*, tanto en términos de tiempo de análisis necesario como de eficacia de los resultados. Adicionalmente estudiamos los diferentes niveles de optimización en compilación, los cuales transforman los bucles de diferentes maneras. De hecho, nos encontramos con varias observaciones interesantes en estos experimentos por lo que incluimos también una evaluación específica sobre el impacto en el WCET de los niveles de optimización en compilación y sus transformaciones de bucles.

Para estos experimentos asumimos un procesador de 32-bit ARM, con la arquitectura asumida por defecto en Otawa [Tra], una herramienta del estado del arte basada en *AbsInt* e *IPET*. Esta arquitectura por defecto consiste en una segmentación con anchura para una instrucción con etapas que no consumen más de un ciclo (aparte de los fallos en cache de instrucciones) y un predictor de saltos *no-tomados* estático. Esto significa que se introduce una penalización de 2 ciclos por cada salto tomado. También asumimos la presencia de un *line-buffer* de instrucciones, es decir una cache de instrucciones (dinámica) convencional de una sola línea, como en estudios anteriores [Apa+10; PD02; Seg+12]. Este *line-buffer* se necesita para que la cache de instrucciones bloqueable funcione adecuadamente. Los binarios de prueba se han generado con ARM Compilation Toolchain *GCC 6.3.1* [ARM] con soporte hardware para coma flotante. Para cada programa de prueba (benchmark), los experimentos

TABLA 3.1: Entorno experimental.

Parámetro	**Casos Probados**
Método de análisis	Lock-MS, DLock-MS, Otawa (AbsInt+IPET), simulación
Max. iteraciones en bucles	Introducido manualmente
Compilation toolchain	ARM GCC 6.3.1
Nivel de optimización (-O)	0, 1, 2, 3
Line-buffer de inst.	Presente (tamaño de línea de cache)
Tam. línea cache de instr.	32 Bytes
Asoc. cache instr.	Mapeo directo, 2 asociativa por conjuntos
Tam. cache de instrucciones	128, 256, 512, 1024 Bytes, siempre-acierto, siempre-fallo
Latencia de memoria	10 ciclos
Acceso a memoria de datos	Ideal (sin ciclos de penalización)

incluyen todas las posibles combinaciones de parámetros que se muestran en la tabla 3.1. Hemos llevado a cabo más experimentos variando la versión de compilación, tamaño de línea, latencia de memoria y asociatividad de la cache pero no mostraban ningún comportamiento interesante o inesperado, así que no los incluimos en este trabajo. Consideramos una cache de instrucciones cuyo coste de acierto es 1 ciclo, y su latencia de memoria relativamente baja (10 ciclos). Otros estudios consideran memorias principales off-chip y asumen una tasa de penalización en fallo mayor, como 30 ciclos [ZWY17]. Evidentemente, la mejora en estos sistemas será mucho mayor.

Implementamos *DLock-MS* como un módulo en el entorno de trabajo de Otawa [Tra]. Este entorno nos proporciona el procesado de los binarios y el CFG, así que podemos aplicar directamente nuestra propuesta de transformación de CFG a árbol y generar las restricciones ILP necesarias desde la estructura en árbol. Otawa necesita conocer previamente cierta información de control de flujo, como las cotas de los bucles. Así que, para cada binario, se ha ajustado de manera manual esta información estudiando cuidadosamente el efecto de las optimizaciones del compilador, sobre todo las transformaciones de bucles como desenrollado, fusión, división, recolocación, duplicación, etc. No obstante, se podría usar cualquier herramienta de análisis de cotas de bucles [BMS08; LPR14]. La única transformación que encontramos ha sido la de desenrollado completo del bucle en los niveles de optimización O2 y O3 en los benchmarks *binarysearch*, *crc*, *g723_enc*, *lift*, *matrix1*, *md5*, *ndes*, y *petrinet*.

La tabla 3.2 muestra los benchmarks que usamos en los experimentos de este capítulo, en la sección 2.2 se puede encontrar su caracterización.

En esta tabla se muestra solamente la versión de TACLeBench en los resultados de aquellos benchmarks que son diferentes pero contienen los mismo algoritmos (*binarysearch/bs*, *countnegative/cnt*, *jfdctint/fdct*, *petrinet/nsichneu*). Sin embargo, los experimentos se han llevado a cabo con ambas para confirmar que cada par presentaba el mismo comportamiento.

Lo valores que encontramos bajo «Tamaño de código binario (Bytes)» muestran el número de bytes de las instrucciones analizadas en el fichero binario, dependiendo de su nivel de optimización. Esto es, la suma de todas las instrucciones en el CFG por su tamaño (4 bytes por instrucción). La optimización de tamaño (-Os) se ha descartado debido a que en general necesita funciones de bibliotecas que contienen bucles con números de iteraciones desconocidos.

En la columna CFG podemos encontrar un número aproximado de caminos del CFG (tomando los binarios con optimización O3) que nos permite hacernos una idea de la complejidad del análisis del WCET con respecto al número de caminos de ejecución a analizar. Hay que destacar que contar el número de caminos puede ser

TABLA 3.2: **Benchmarks utilizados en la experimentación del capítulo 3.**

Nombre	Tamaño código binario (Bytes)				Complejidad en O3 (Caminos)		
	O0	O1	O2	O3	CFG	Árbol	A+R
audiobeam	8572	3880	3940	4848	$\sim 2^{597}$	2^{24}	1512
basicmath	10048	4516	4516	5028	$> 2^{2080}$	2^{62}	69
binarysearch	604	264	260	1812	16	2	2
bsort	652	276	200	200	2	2	2
complex_updates	936	428	376	2300	1	1	1
countnegative	780	356	300	320	1	1	1
crc	1120	448	384	580	256	256	256
dijkstra	1544	764	636	808	$> 2^{1860000}$	124	63
fft	1644	960	836	784	$> 2^{2048}$	4	3
filterbank	1652	768	688	880	2	2	2
fir2dim	1456	712	640	1148	1	1	1
fmref	6344	3156	3288	5892	$> 2^{854}$	2^{52}	234995
g723_enc	6772	2724	2960	6104	-	-	-
iir	748	348	340	556	1	1	1
janne_complex	264	100	100	100	1	1	1
jfdctint	2972	1088	1136	1136	1	1	1
lift	3828	2440	2304	3704	$> 2^{1001}$	4462	4462
ludcmp	2540	1084	972	4004	$\sim 2^{95}$	2789600	776
matmult	704	304	324	308	1	1	1
matrix1	592	264	244	380	1	1	1
md5	7968	3268	3128	5432	$> 2^{2827}$	2^{31}	78839
minver	3048	1252	1108	1920	$\sim 2^{30}$	1800	363
ndes	3168	1464	1760	2296	$\sim 2^{998}$	98560	104
petrinet	6644	3700	3604	3644	$> 2^{133}$	2^{67}	2^{67}
pm	8092	3648	3692	3984	$> 2^{2541}$	2^{33}	374641
qsort-exam	1480	620	620	620	$\sim 2^{175}$	40	40
qurt	1092	580	548	540	1000	1000	1000
select	1332	516	520	520	$\sim 2^{64}$	30	16
st	1680	768	780	1044	162	162	162
statemate	9508	7600	6676	6388	$> 2^{1000}$	2^{46}	2^{46}

más difícil que el propio cálculo del WCET, ya que la mayoría de métodos de análisis agrupan caminos precisamente para evitar la explosión combinatoria. Por ejemplo, no hemos sido capaces de determinar el número de caminos en el benchmark *g723_enc*.

En la columna Árbol se presenta el número específico de caminos a analizar con un método de análisis del WCET basado en la estructura. Su valor corresponde con la aplicación de nuestra primera mejora. En la columna A+R se muestran los caminos correspondientes a haber aplicado nuestras regiones en la estructura en árbol, es decir, nuestro método *DLock-MS*.

En este caso, la reducción del número de caminos se debe a que cada región se analiza de manera independiente. Esto significa que el número de caminos de regiones consecutivas no se multiplica sino que se suma, tal como explicamos en la figura 3.3.

Cuando aplicamos *DLock-MS* tenemos que considerar cierto sobrecoste. Asumimos que el bloqueo dinámico se realiza ejecutando funciones, tal como hacen otros estudios [ZWY17]. Esas funciones cargan y bloquean el conjunto de instrucciones necesarias en la cache para la siguiente región del código. En concreto, consideramos una penalización de 47 ciclos por llamada, una por región. Estos ciclos corresponderían a los costes de ejecución, fallos en cache y penalizaciones de la segmentación en una función que, estimando, sería de 12 instrucciones. Además, por cada línea de cache bloqueada, añadimos el coste de latencia de memoria (10 ciclos). Este escenario es bastante conservador ya que para bloquear instrucciones se podrían usar operaciones de carga especializadas, por ejemplo, cargando un gran número de líneas de memoria continuas usando algún tipo de modo de transferencia de memoria en ráfagas, lo que podría reducir de manera notable el tiempo total de transferencia [ZWY17].

El modelo ILP que obtenemos se resuelve por el *solver lp-solve* versión 5.5.2.3.

Debido a las particulares propiedades de anidación de nuestro modelo basado en la estructura, usamos las siguientes opciones en el *solver* -BB, -Bc, -Bd, -Bg, y -Bo para ordenar variables y aplicar una codiciosa ramificación y poda inversa.

3.4.1. Evaluación de tiempos de análisis

En esta sección estudiamos el tiempo necesario para el análisis estático del WCET con el método *DLock-MS*, una extensión de *Lock-MS* con una transformación eficiente de CFG a árbol y una heurística de bloqueo dinámico.

La figura 3.4 muestra nuestros tiempos de análisis comparados con aquellos que necesita el analizador del WCET (*AbsInt + IPET*) de Otawa (owcet v1.2.0) [Tra]. En el eje x están representados los benchmarks y en el eje y la mejora obtenida en el análisis del WCET: el tiempo de ejecución del análisis del WCET de Otawa asumiendo una cache de instrucciones LRU convencional dividido por el tiempo de ejecución de *DLock-MS* en una cache de instrucciones bloqueable del mismo tamaño. En la parte de la derecha se especifica el nivel de optimización. Los benchmarks están ordenados por número de caminos a explorar en la representación basada en árbol para la optimización O3, de manera que refleja su nivel de complejidad. Para cada benchmark, se muestran con boxplots todos los experimentos variando el tamaño de cache (ver tabla 3.1). Esto es, cada columna muestra una caja cuyos límites indican el primer y tercer cuartil, con una marca dentro que determina la mediana. En las líneas verticales fuera de la caja se muestra la variabilidad fuera de estos cuartiles y los puntos más allá de estas líneas indican valores atípicos, es decir, aquellos que no son estadísticamente relevantes. También, la línea horizontal muestra la unidad de mejora, es decir cuando nuestra propuesta no mejora ni empeora frente a Otawa, y por lo tanto el cociente de comparación es uno. Así, cuanto más altos están los boxplots más rápido es nuestro análisis del WCET comparado con el de Otawa.

Con respecto al tiempo de análisis, lo limitamos a un máximo de 10 minutos, así que cualquier experimento que tarde más tiempo se asumirá que ha tardado 10 minutos exactamente. Señalemos entonces que cuando estos casos aparecen en Otawa estamos asumiendo un tiempo de análisis menor que el que debería de ser. Con *DLock-MS*, solo un experimento tarda más de 10 minutos. Sin embargo, es importante destacar que nosotros somos capaces de proporcionar resultados seguros antes de que se complete nuestro análisis/optimización del WCET. Esto es, cuando paramos nuestro análisis a los 10 minutos, ya tenemos un cota segura del WCET y un conjunto de contenidos específicos para cargarlos en cada punto de bloqueo,

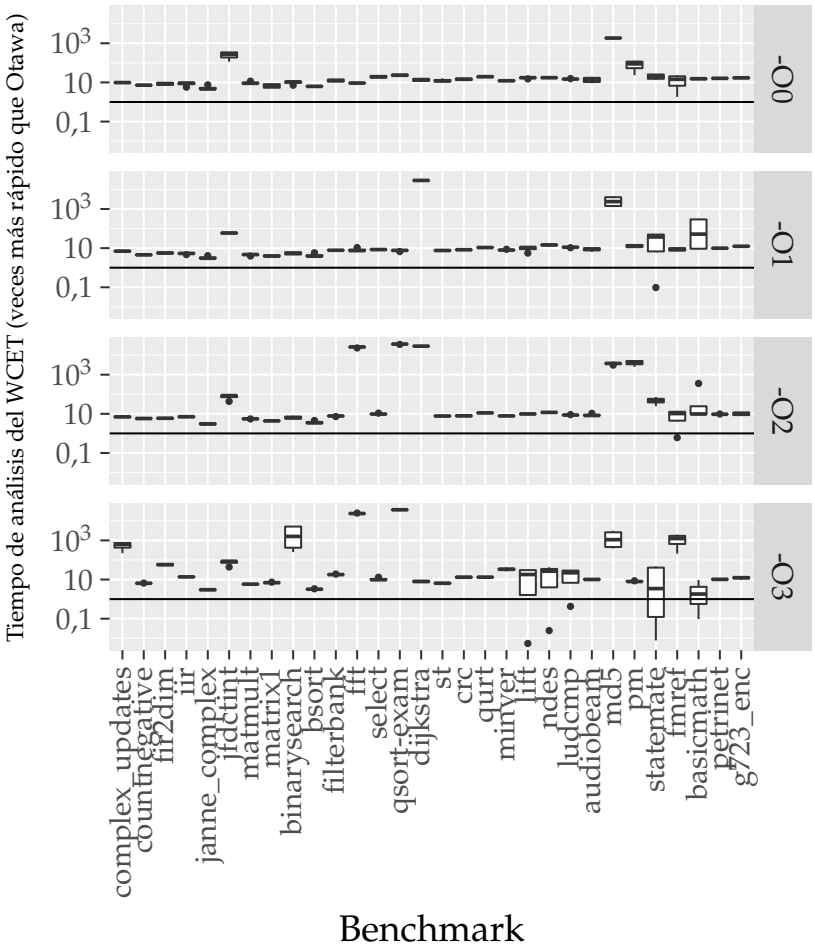


FIGURA 3.4: Comparación de tiempo de análisis estático del WCET para nuestra propuesta (basada en la estructura) y Otawa (*AbsInt + IPET*).

aunque no podamos garantizar que es la configuración óptima, es decir, la que proporciona el WCET menor. Por lo tanto, asumir que los experimentos no tardan más de 10 minutos beneficia a Otawa en la comparación.

En la figura 3.4 vemos que en la mayoría de los casos las cajas son simplemente líneas horizontales. Para estos benchmarks, esto significa que la variación de la mejora es muy pequeña en relación al tamaño de cache. Tanto a través de los benchmarks como de los niveles de optimización, *DLock-MS* es normalmente unas 10 veces más rápido que Otawa, aunque ciertos benchmarks pueden ser especialmente difíciles de analizar para cada método. Por ejemplo, los resultados por encima de 10^4 corresponden a análisis que no se han completado en 10 minutos en Otawa. Por otro lado, en los pocos casos en los que el análisis de Otawa es más rápido parece que se debe a la existencia de varios caminos con un tiempo igual o muy similar de ejecución. Esto evita que el *solver* pueda descartar rápidamente estos caminos como sucede en el caso de *statemate O3*. También se tiene que tener en cuenta que Otawa es más rápido que otras propuestas. Por ejemplo, el tiempo de análisis necesario que encontramos en otros estudios es más de 30 veces mayor que el nuestro [ZWY17].

Aunque no se presenta en la figura 3.4, hemos estudiado también el tiempo que *DLock-MS* invierte en cada parte del análisis. Para los experimentos que se presentan en esta figura, nuestra propuesta tarda una media de 0,06 segundos en obtener el

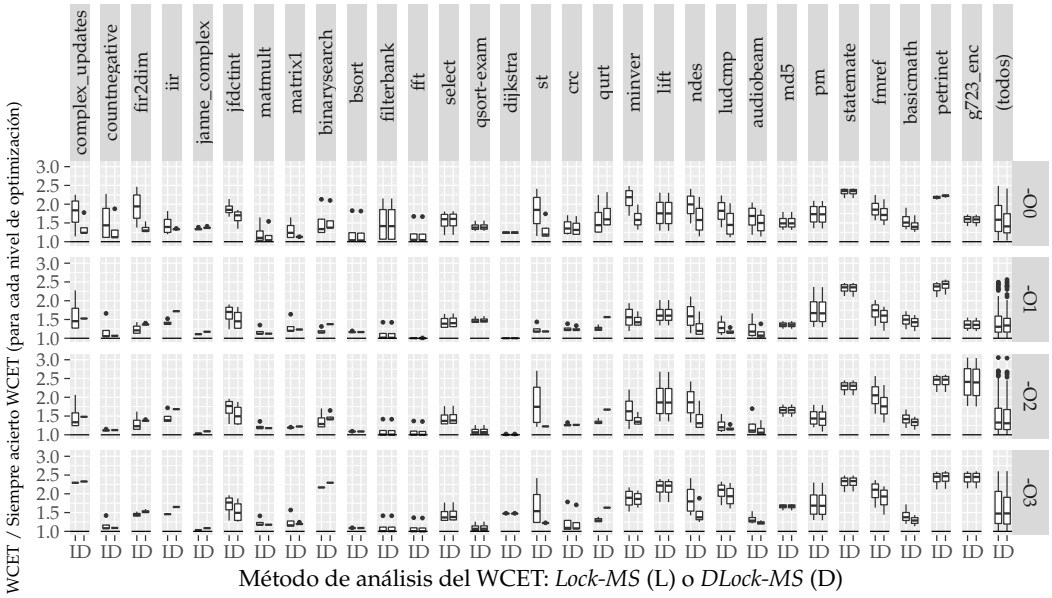


FIGURA 3.5: Comparación de *WCETs* de *Lock-MS* y *DLock-MS*, cuanto menor mejor.

CFG y generar la estructura en árbol y las restricciones de ILP, mientras que el ILP *solver* tarda una media de 2,29 segundos en resolver el problema.

3.4.2. Evaluación de la eficacia

La figura 3.5 muestra la evaluación de la eficacia de *DLock-MS* comparada con el método *Lock-MS* original. Como antes, los resultados están representados por boxplots para cada nivel de optimización. Ya que *DLock-MS* básicamente extiende a *Lock-MS* admitiendo múltiples puntos de carga y bloqueo, sus resultados son, en general, iguales o mejores. En concreto, los *WCETs* se mejoran un 2,2 % de media, incluyendo ya el coste extra de introducir los puntos de bloqueo. Este sobrecoste (una llamada a función en cada punto de bloqueo) supone en media un 2,1 % del WCET.

Aplicando un test de signo de Fisher con un nivel del confianza de 0,99, se obtiene un p-valor de $2{,}2 \cdot 10^{-16}$, lo cual afirma que *DLock-MS* funciona mejor que *Lock-MS*.

Aunque las mejoras en la figura 3.5 puedan parecer pequeñas, hay que tener en cuenta que en varios casos *Lock-MS* ya alcanza el mejor WCET posible (por ejemplo en *dijkstra*), por lo tanto en estos casos no hay margen de mejora. También puede incrementarse el WCET ligeramente al añadir puntos de carga y bloqueo innecesarios, aunque este incremento debería notarse solo en benchmarks sencillos (por ejemplo en *iir*, *janne_complex* y *binarysearch*). En los benchmarks más complejos (los de la parte derecha), las mejoras en el WCET son claras y el único caso problemático es cuando varias regiones están volviendo a cargar los mismos contenidos, como pasa en *qurt*. No obstante, esta situación es bastante trivial de detectar y evitar.

Además de las comparaciones previas, evaluamos también la eficacia de la cache de instrucciones bloqueable cuando se analiza con *DLock-MS*. Debido a que el WCET decrece linealmente con respecto a la tasa de aciertos de la cache de instrucciones, si nuestra tasa de aciertos en el camino del WCET es parecida a aquella en la ejecución real, podemos asegurar que nuestros resultados son lo suficientemente precisos. Además de este modo evitamos que la peculiaridades de los métodos de

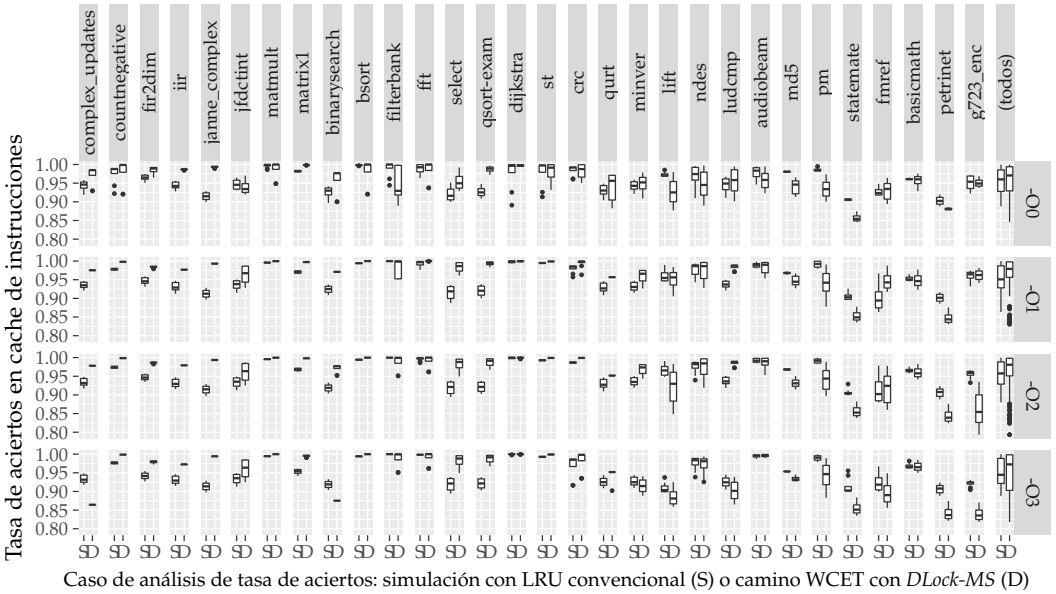


FIGURA 3.6: Comparación de las tasas de aciertos en la cache de instrucciones de nuestra propuesta (tasa de aciertos alcanzada a través del camino WCET analizado) y una simulación de ejecución (tasa de aciertos de una ejecución de simulación con una cache de instrucciones convencional), cuanto mayor mejor.

análisis del WCET enturbien el objetivo real, esto es, ser ligeramente mejor que la cota del WCET alcanzada por otros métodos es mucho menos importante que acercarse al WCET real del programa. Así mismo, comparamos los resultados de nuestra cache de instrucciones bloqueable con una cache de instrucciones LRU para comprobar que *DLock-MS* alcanza un nivel aceptable de rendimiento. Para obtener las tasas de acierto reales usamos el simulador Gem5 v2.0 [Bin+11] configurando una segmentación equivalente con una cache de instrucciones LRU del mismo tamaño. Por otra parte obtenemos la tasa de aciertos de la cache de instrucciones bloqueable en el peor camino de ejecución cuando se aplica *DLock-MS*. Como antes, estos resultados se presentan a través de boxplots para cada benchmark y nivel de optimización.

La figura 3.6 muestra que las tasas de aciertos en el peor camino con una cache bloqueable son comparables con aquellas con una cache LRU en todos los benchmarks y niveles de optimización, y en media es siempre mejor en la cache bloqueada dinámicamente (columna de la derecha). De hecho, hay muchos casos donde la cache bloqueada dinámicamente supera la cache LRU. Esto significa que, para muchos benchmarks, bloquear el código adecuado es probable que funcione tan bien o mejor que la política LRU, cuyo dinamismo puede expulsar contenido que se usará de nuevo pronto. Por otro lado, los benchmarks cuya tasa de aciertos LRU es mayor son mayoritariamente los que están situados a la derecha en la figura 3.6. Esto es coherente, ya que el dinamismo natural de la cache LRU adapta su comportamiento en estos benchmarks que son más grandes y complejos, mientras que la cache bloqueable, incluso con una heurística de bloqueo dinámico, tiene un comportamiento más restringido.

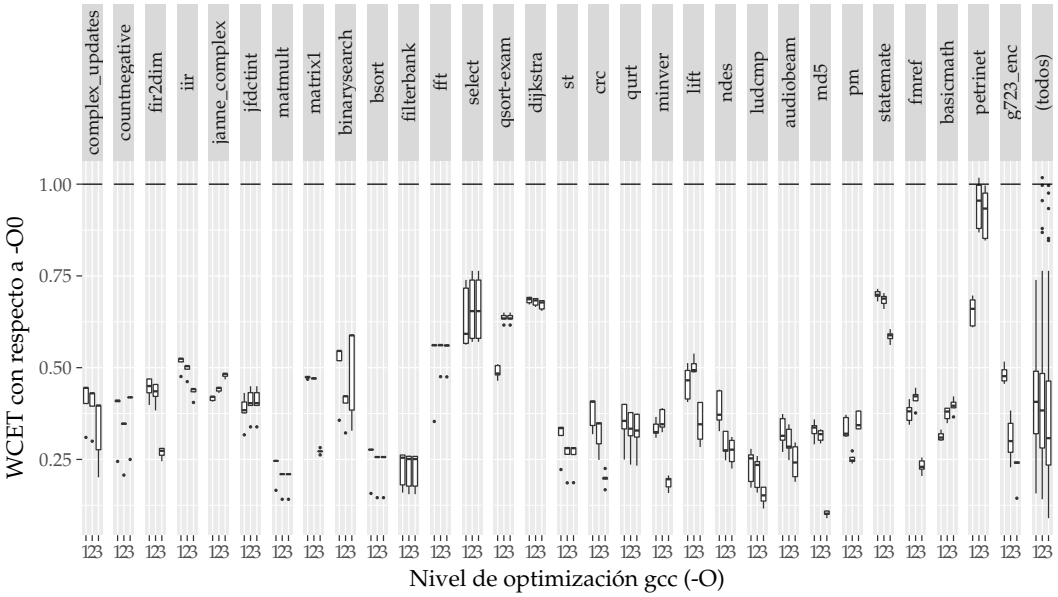


FIGURA 3.7: Efectos del nivel de optimización del compilador (GCC 6.3.1) en el WCET (cuanto menor, mejor).

3.4.3. Impacto del nivel de optimización del compilador

La mayoría de estudios en sistemas de tiempo real deshabilitan las optimizaciones para que la concordancia entre le código binario y el de alto nivel sea más sencilla. Además, hasta donde sabemos, ninguno de ellos ha realizado un análisis exhaustivo sobre cómo afectan las optimizaciones al peor caso de tiempo de ejecución. De manera intuitiva podemos afirmar que las optimizaciones reducen el tiempo medio de ejecución, así que, en general, debería reducirse también el WCET. Sin embargo, cualquier optimización que reduce el tiempo medio de ejecución incrementando el tiempo de ejecución en caminos infrecuentes, resultará en un incremento del WCET si este se encuentra en uno de estos caminos inusuales. En esta sección estudiamos el impacto en el WCET de los niveles de optimización en compilación. Nos centramos en los resultados para GCC 6.3.1, los resultados para GCC 4.8.4 (no se muestran) presentan tendencias casi idénticas.

La figura 3.7 muestra cómo los niveles de optimización afectan al WCET. El eje x representa el WCET relativo a compilar sin ninguna optimización (-O0). Se presenta esta información para cada benchmark más el agregado de todos ellos en la parte de la derecha, y se representan con boxblots como antes. También podemos ver una línea horizontal que indica el caso base (es decir, el WCET del binario compilado sin optimizaciones para cada experimento) en $y = 1$. Por lo tanto, cuanto más bajos estén los bloxpots, menor (mejor) será el WCET correspondiente para esa optimización.

La observación más importante es que, en media, los boxplots se sitúan alrededor del 0,3. Esto significa que, en general, el WCET de un código optimizado será en torno a un tercio de su WCET sin optimizaciones. Por lo tanto, se deberían usar binarios optimizados en los sistemas de tiempo real. En media, los mejores resultados se alcanzan con O3, aunque cada benchmark tiene un comportamiento específico.

Otro detalle interesante es que la mejora del WCET no se debe (exclusivamente) al tamaño del código del binario optimizado. Los binarios de O3 normalmente son mayores que aquellos compilados con O1 y O2 (ver en la tabla 3.2) y sin embargo

presentan un WCET menor.

Por ejemplo, el tamaño de los binarios O3 es alrededor de dos veces el tamaño de O2 en *fir2dim*, *fmref*, *g723_enc*, *iir*, cuatro veces en *ludcmp*, y seis veces en *complex_updates*, y a pesar de ello O3 consigue un WCET menor. Este es un punto muy importante a tener en cuenta ya que todos los experimentos realizados conllevan un gran trabajo para la cache que el tamaño del binario puede incrementar sustancialmente. Sin embargo, estas optimizaciones pueden tener también efectos ligeramente adversos para el WCET, como puede verse en *binarysearch*, dónde O3 es unas 7 veces mayor que el tamaño de O2.

Destaca el efecto singular que encontramos en *petrinet*. Como se puede ver, O2 y O3 muestran un WCET significativamente peor que O1. Esto se debe a las transformaciones que se han llevado a cabo en los bucles por estas optimizaciones. Estas transformaciones han derivado en unos patrones de bucle que de manera inherente se sobrestiman en el proceso de análisis del WCET. A continuación, exploramos más profundamente en este problema de análisis de ciertos patrones de bucle. Dependiendo del código fuente y el nivel de optimización, la distribución de bloques básicos que contienen la cabecera y el cuerpo del bucle pueden ser totalmente distintos. En la figura 3.8 se muestran varios patrones de bucle, donde N es el número máximo de ejecuciones del cuerpo del bucle, que en nuestros experimentos ha sido etiquetado manualmente. En esta figura, las cajas limitadas por líneas discontinuas representan uno o más bloques básicos, mientras aquellas limitadas por líneas continuas representan un solo bloque básico.

En el patrón 1 (a) encontramos el bucle básico *do-while*. Solo hay una salida y está en el mismo bloque básico que vuelve al inicio del bucle. Todos los bloques básicos de este bucle se ejecutan como mucho N veces.

El patrón 2 (b) (un bucle *while* o *for*) tiene solo una salida, que está en el bloque básico de entrada (la cabecera) del bucle. Si el cuerpo del bucle se ejecuta N veces, su cabecera se ejecuta $N + 1$ veces.

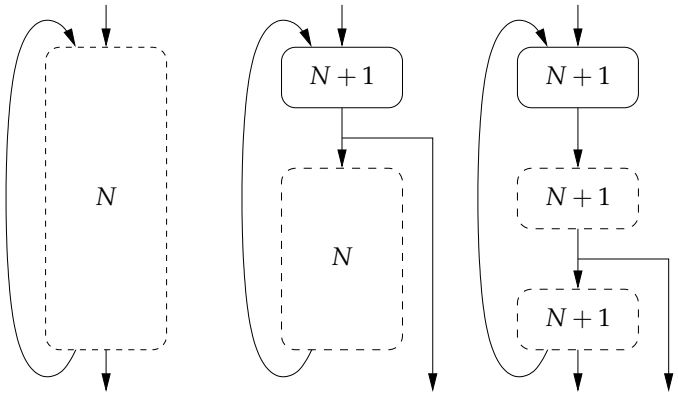
Nos encontramos que en otros patrones de bucle, como el 3 (c), puede ser difícil o incluso imposible saber dónde se encuentra la cabecera o el cuerpo del bucle, los cuales pueden incluso estar intercalados con condiciones de salida de bucle, ambos distribuidos a lo largo de varios bloques básicos. Por lo tanto, una aproximación segura implica asumir que todos los bloques básicos implicados se ejecutan $N + 1$ veces. Esta aproximación, aunque segura, puede producir una sobrestimación. Este es el caso de los binarios O2 y O3 de *petrinet* en la figura 3.7. En O1, este benchmark contiene 151 bloques básicos (de un total de 161) que se ejecutan dos veces cada uno en el peor caso, sin embargo en O3 se tiene que asumir que se pueden ejecutar hasta 3 veces. Esto incrementa la cota del WCET en torno a $3/2$, como se pude ver en la figura 3.7.

3.5. Conclusiones

En este capítulo se ha presentado *DLock-MS*, una extensión del método de anális del WCET *Lock-MS*.

Nuestro objetivo es reducir el tiempo de análisis del WCET en presencia de una cache de instrucciones bloqueable. Nuestra extensión implementa principalmente dos mejoras.

La primera es un algoritmo para transformar el CFG en una estructura en árbol, necesaria para usar *Lock-MS* como un método de análisis del WCET basado en la



(a) Patrón 1 (b) Patrón 2 (c) Patrón 3

FIGURA 3.8: Varios patrones de bucle encontrados en el código binario.

estructura. Este algoritmo genera un árbol cuyo modelo ILP puede resolverse fácilmente a través de una ramificación y poda invertida. Estas transformaciones se realizan en una sola pasada, procesando cada camino alternativo una sola vez. El árbol resultante tiene muchos menos caminos a explorar que el CFG original, lo cual reduce el tiempo de análisis del WCET sin sacrificar precisión para una cache de instrucciones bloqueable.

La segunda mejora es una heurística de bloqueo dinámico basada en los bucles, aplicada en los bucles externos, que permite obtener el contenido óptimo de cache para el WCET de cada región, es decir, la configuración que minimiza el WCET en cada región. La complejidad de esta heurística es muy baja, reduce el WCET explotando de manera efectiva el reuso temporal, y reduce todavía más el tiempo de análisis del WCET aislando el WCET de cada región.

Los resultados muestran que *DLock-MS* es alrededor de 10 veces más rápido que Otawa, una herramienta del estado del arte basada en *AbsInt* e *IPET*. Debido a su rapidez este análisis del WCET pude ser muy significativo en el proceso de diseño de un sistema de tiempo real y una alternativa al análisis paramétrico del WCET. Además, nuestro análisis puede detenerse antes de completarse, proporcionando, incluso en este caso, un WCET seguro y una configuración para la cache bloqueable que lo garantiza. Esto quiere decir que cualquier solución de nuestro modelo es segura, y que es al completar el análisis cuando se garantiza el WCET óptimo (mínimo) para cada región.

También evaluamos la eficacia de *DLock-MS*, confirmando que reduce el WCET respecto al método original *Lock-MS*, y comparamos su tasa de aciertos en una cache bloqueable con una convencional LRU. Nuestros resultados muestran una tasa de aciertos muy similar en todos los benchmarks, siendo la cache bloqueable la que ofrece mejores tasas de aciertos en muchos de ellos. Hay que destacar también que estos resultados se consiguen con un hardware muy sencillo, mucho más que una cache convencional LRU.

Finalmente, estudiamos el impacto de los niveles de optimización en el WCET. Nuestra conclusión es que se deben descartar las compilaciones sin optimización (-O0) ya que generan binarios con un WCET entre 3 y 4 veces peor que con la presencia de optimizaciones. En general, O3 genera los binarios con los *WCETs* más pequeños, pero otros niveles de optimización son también muy eficaces. Sin embargo, optimizaciones agresivas en algunos benchmarks pueden suponer un incremento

significativo del WCET. Estas optimizaciones pueden cambiar los patrones de bucle de manera que el método de análisis del WCET se ve forzado a asumir iteraciones adicionales en los bucles para garantizar la seguridad del análisis.

Este trabajo ha sido publicado en [Ped+20b].

Capítulo 4

Impacto de la vectorización automática en el WCET

En este capítulo continuamos el estudio del análisis del WCET considerando las instrucciones resultantes de la vectorización automática. Desde los años 80 los microprocesadores comerciales han estado añadiendo constantemente extensiones vectoriales a su repertorio de instrucciones y recursos hardware para poder ejecutar eficiente estas instrucciones vectoriales. Sin embargo, su impacto en el WCET no se ha estudiado en profundidad debido a la falta de soporte de las instrucciones vectoriales en las herramientas actuales de análisis del WCET. Este será entonces el objeto de este capítulo. Generamos el código vectorizado de manera automática desde el código fuente a través de las funciones que proporcionan los compiladores actuales. Después, partiendo de las especificaciones temporales de los fabricantes estudiamos las partes vectorizadas permitiéndonos acotar su aportación al WCET de la tarea. Finalmente, integramos las cotas obtenidas en el análisis del WCET de la tarea correspondiente. Como resultado obtenemos que el WCET se reduce si se vectorizan los bucles que concentran la mayor parte del tiempo de ejecución. Además, la eficiencia de la vectorización automática en las herramientas de compilación es bastante limitada, por lo que cabe esperar más beneficios en benchmarks más grandes o si se reescribe el código.

4.1. Introducción

Independientemente del método de análisis del WCET, mejorar una tarea en los sistemas de tiempo real estricto significa reducir la cota calculada del WCET, aunque esto suponga incrementar el tiempo medio de ejecución. Por el contrario, no siempre que se reduce el tiempo medio de ejecución va a verse reducido el WCET. A este respecto, las mejoras microarquitecturales como la memoria cache han demostrado reducir el tiempo medio de ejecución pero no el peor caso de ejecución, tal y como hemos visto. En este capítulo, nos centramos en otra mejora microarquitectural: la vectorización, que explota el paralelismo de datos, analizando su efecto en el WCET.

El procesador Intel Pentium MMX popularizó la vectorización a mediados de los años 90 y actualmente encontramos que existen procesadores de distintos fabricantes que incorporan en su repertorio de instrucciones (*instruction set architecture, ISA*) las llamadas extensiones vectoriales. Estas extensiones utilizan un tipo de paralelismo en las que cada instrucción define una operación que se aplica simultáneamente en los elementos de un vector (*single instruction multiple data, SIMD*). Desde su aparición, la vectorización ha demostrado ampliamente su solidez tanto en aplicaciones

científicas como comerciales [BC11]. De esta manera son las herramientas de compilación las responsables de detectar el paralelismo en datos y transformar automáticamente el código escalar en código vectorial [Mal+11]. Por tanto, la vectorización ya se encuentra disponible en los entornos de tiempo real, pero las herramientas de análisis del WCET deben admitir el análisis de estas instrucciones para poder probar su utilidad. Debido a que el camino crítico de muchas aplicaciones de tiempo real estricto atraviesa bucles de cálculo intensivo, la vectorización del código parece una vía adecuada para reducir su WCET.

Hasta donde sabemos solo hay un estudio que analiza el impacto de la vectorización en el WCET [LPR15]. Éste se centra en rastrear el flujo de información desde el código fuente al binario teniendo en cuenta las transformaciones del compilador asociadas a la vectorización. Sin embargo, no realiza el análisis estático del WCET porque, como ya hemos señalado, las herramientas actuales de análisis del WCET no admiten el análisis de instrucciones vectoriales. En cambio, proporciona los tiempos de ejecución en tareas de un solo camino tomando las mediciones en hardware real. Aunque sus benchmarks no son específicos para tiempo real y se ven restringidos a aquellos con un solo camino, sus resultados son bastante prometedores.

En este capítulo llevamos a cabo la experimentación que se ha propuesto en este estudio previo [LPR15]. Usamos la vectorización automática que se encuentra en los compiladores actuales para generar código vectorizado desde el código de alto nivel. Después, estudiamos en detalle las partes vectorizadas, en función de las especificaciones temporales de los fabricantes, para acotar su tiempo de ejecución. Estas cotas se integran en el análisis del WCET del benchmark correspondiente para obtener la cota del WCET para el benchmark vectorizado. Nuestros experimentos se ejecutan con paquetes de benchmarks de tiempo real (TACLeBench [Fal+16] y Mälardalen [Gus+10]). Nuestros resultados muestran que la vectorización es eficaz reduciendo el WCET cuando los bucles vectorizables concentran gran parte del tiempo de ejecución. Sin embargo, la eficacia de la vectorización automática en las herramientas de compilación es todavía bastante limitada, por lo que cabe esperar más beneficios en benchmarks más grandes o si se reescribe el código.

Este capítulo se organiza de la siguiente manera. En la sección 4.2 se presentan brevemente los beneficios de las instrucciones vectoriales en los programas. En la sección 4.3 describimos el análisis del WCET y la integración del nuevo código vectorial en este análisis. En la sección 4.4 detallamos el entorno experimental. En la sección 4.5 examinamos los resultados obtenidos. Finalmente, las conclusiones del capítulo se exponen en la sección 4.6.

4.2. Paralelismo SIMD: recursos del procesador, ventajas y vectorización automática

Un procesador vectorial es aquel cuyas instrucciones se definen en los elementos de un vector de datos. Desde la aparición del primer procesador vectorial exitoso, el Cray-1S (1975), la vectorización ha estado presente en los procesadores del ámbito científico y desde la aparición del Pentium MMX (1997), también en el comercial. Actualmente, los procesadores de diferentes fabricantes (AMD, ARM, Fujitsu, IBM, Intel, etc.) han incorporado en sus ISAs las denominadas extensiones vectoriales (por ejemplo, NEON en ARM, y AVX en x86). Estas extensiones cuentan con instrucciones que definen operaciones que se aplican simultáneamente a los elementos del vector, denominándose este paralelismo *SIMD*. Con este objetivo la ruta de

datos del procesador se ha ampliado, principalmente las unidades aritmético lógicas (ALU) y el banco de registros. El número de elementos que se pueden procesar simultáneamente en la ALU vectorial queda definido por la anchura del banco de registros y el tamaño de los elementos procesados. Dependiendo de la fuente, este número se denomina grado de vectorización, número de vías vectoriales o anchura de la ALU vectorial.

4.2.1. Ventajas del código vectorial frente al código escalar

Las instrucciones vectoriales de los procesadores están destinadas a procesar los elementos de un vector con la misma operación o secuencia de operaciones. Una fuente evidente de este tipo de paralelismo son los bucles de los programas.

En la figura 4.1, a la izquierda, se muestra un código escalar que suma los vectores *A* y *B* en el vector *C*. A la derecha, se muestra una versión vectorizada del código anterior escrito con funciones *GCC intrínsecas* para NEON ARM, las cuales se ensamblan a través de sus correspondientes instrucciones de la extensión vectorial NEON ARM.

Cada iteración del primer bucle de la versión vectorizada procesa cuatro elementos enteros. Las primeras dos instrucciones del cuerpo del bucle cargan cuatro enteros consecutivos de los vectores *A* y *B* en variables vectoriales *vec_A* y *vec_B*. Después, los cuatro elementos de *vec_A* y *vec_B* se suman en *vec_C* con una sola instrucción, y finalmente, se almacenan en el vector *C*. Los beneficios que encontramos en este código son, primero, que el número de ejecuciones del bucle *for* se divide por cuatro, y segundo, que la suma vectorial (*vaddq_i*32) de cuatro elementos tiene una latencia por entero menor que en la versión escalar de la suma (en este caso, normalmente, un cuarto).

En general, el número de iteraciones no es múltiplo del tamaño del vector que se proporciona por la arquitectura (vías del camino de datos vectorial). Esto quiere decir que habrá algunas iteraciones que no se podrán procesar con instrucciones vectoriales. Estas iteraciones restantes se deben procesar en un bucle escalar como epílogo del bucle vectorial. Por lo tanto, el número de iteraciones de este epílogo siempre será menor que el número de vías vectoriales para el tipo de dato de los vectores procesados.

Este epílogo se corresponde con el segundo bucle de la versión vectorial del código en la figura 4.1, el cual ejecutará como máximo tres iteraciones.

4.2.2. Vectorización automática

Usar funciones intrínsecas para generar código vectorial tiene varias desventajas en términos de programabilidad, legibilidad, y portabilidad [Mal+11]. Afortunadamente, la mayoría compiladores actuales (*GCC, icc, llvm, cc*) permiten generar código vectorizado automáticamente desde el código escalar escrito en lenguaje de alto nivel. Para poder hacer esto el compilador debe realizar algún grado de análisis de inter-dependencia entre iteraciones de los elementos de los vectores de los bucles. En caso de que las iteraciones de los bucles sean independientes, el compilador puede producir de manera segura código vectorial para ese bucle. Además, algunos compiladores llevan a cabo un análisis de la mejora que va a proporcionar la vectorización y, solo si merece la pena, generan el código vectorizado. Para este análisis suelen emplear algún reordenamiento de código o transformación de datos [Kuc+81].

```
int A[N], B[N], C[N];

for (int i = 0; i < N; i++){
  C[i]=A[i]+B[i];
}
```

```
int A[N], B[N], C[N];
int32x4_t vec_A, vec_B, vec_C;

for (int i = 0; i<(N/4); i++){
  vec_A=vld1q_i32(&A[i*4]);
  vec_B=vld1q_i32(&B[i*4]);
  vec_C=vaddq_i32(vec_A,vec_B);
  vst1q(vec_C, &C[i*4]);
}
//epílogo
for (int i=(N/4)*4; i<N; i++){
  C[i]=A[i]+B[i];
}
```

FIGURA 4.1: Código escalar (izquierda) y su código vectorial equivalente con funciones intrínsecas (derecha).

4.3. Análisis del WCET

Como ya presentamos en capítulos previos hay varios métodos para analizar el WCET, como *AbsInt* [CC77; FW99], *IPET* [LMW96], o métodos basados en la estructura [Apa+11]. En cualquiera de ellos, el primer paso a realizar implica procesar el código para obtener el CFG. Aunque algunos métodos obtienen el CFG directamente del código fuente, en este caso no hay otra opción que obtenerlo del código binario ya que las instrucciones vectoriales las produce el compilador. Debido a que estas instrucciones son extensiones del repertorio de instrucciones escalar, los procesadores de binarios de las herramientas de análisis del WCET todavía no son capaces de procesarlas (por ejemplo, Heptane [CP00] y Otawa [Bal+10]), tal como se indica en estudios previos [LPR15]. Como ejemplo de aplicación práctica tomamos el método *Lock-MS* (presentado en la sección 2.1.2) y analizamos el impacto de la vectorización del código en sus resultados.

Recordemos entonces que *Lock-MS* se ha diseñado para analizar el WCET en sistemas con una cache de instrucciones bloqueable. Esto simplifica el análisis de la cache de instrucciones y evita que los posibles efectos de la política de reemplazo LRU afecten al comportamiento de los bucles tanto vectorizados como no vectorizados. Además de obtener el WCET, *Lock-MS* proporciona el conjunto de contenidos óptimo para cargar y bloquear en la cache de instrucciones, es decir aquel que minimiza el WCET.

El método *Lock-MS* ha sido implementado en un módulo en el entorno de trabajo de Otawa [Tra] tal como se expone en el capítulo 3. Aunque el módulo desarrolado implementa el método *DLock-MS*, este constituye una extensión de *Lock-MS*, o lo que es lo mismo, el método *Lock-MS* es un caso particular de *DLock-MS* dónde solo especificamos un punto de carga y bloqueo de instrucciones al inicio de la tarea, prescindiendo de la heurística de selección de puntos de carga.

Estudiamos las instrucciones vectoriales en el código binario para determinar su tiempo de ejecución, según las especificaciones temporales de fabricante. Esta información (su dirección más el tiempo de ejecución para cada instrucción vectorial) se le proporciona al módulo de análisis del WCET de manera similar a como se le proporciona la información referente al flujo (máximo número de iteraciones en bucles,

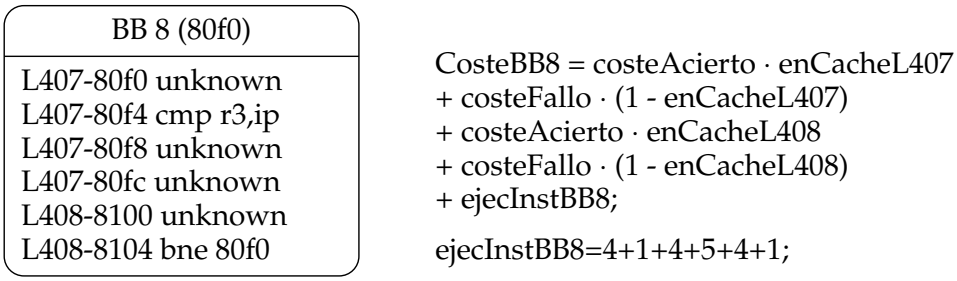


FIGURA 4.2: Ejemplo de un bloque básico y sus restricciones ILP asociadas a su coste.

caminos imposibles, etc.) y se integra en el bloque básico correspondiente. Tras esto, se genera el correspondiente modelo ILP y se resuelve directamente.

En la figura 4.2 presentamos un ejemplo de cómo se integra el tiempo de ejecución de cada instrucción vectorial en las restricciones ILP correspondientes al coste de cada bloque básico que las contiene. La construcción de las restricciones ILP se desarrolla con más profundidad en las secciones 2.1.2 y 3.2, formando parte del módulo que desarrolla el método *DLock-MS*. En el ejemplo puede verse la estructura de un bloque básico con su coste correspondiente (*CosteBB8*) expresado por las restricciones ILP que produce nuestro módulo. En cada instrucción del bloque básico se muestra a qué línea de cache pertenece (*L407* o *L408* en este ejemplo) y su dirección. Las instrucciones vectoriales aparecen denominadas como *unknown* (desconocida), ya que la herramienta no las reconoce. Es en este punto donde le proporcionamos a la herramienta la dirección y coste de ejecución de las instrucciones vectoriales que integra en el sistema ILP. En el ejemplo de la figura 4.2 las instrucciones con direcciones 80f0, 80f8, 80fc y 8100 tienen un coste de 4, 4, 5 y 4 ciclos respectivamente que se integran en la restricción $ejecInstBB8 = 4 + 1 + 4 + 5 + 4 + 1$; que representa el coste total de ejecución de todas las instrucciones del bloque básico. Este coste más el de acierto o fallo en cache de instrucciones de las líneas de cache es el coste total de una ejecución del bloque básico (*CosteBB8*). Por lo tanto, la integración de esta información en nuestro módulo es sencilla y directa.

4.3.1. Cotas en el código vectorial generado automáticamente

Además de considerar el coste de las instrucciones vectoriales debe determinarse el número máximo de iteraciones de los bucles vectorizados. Estos valores se podrían proporcionar automáticamente trazando el flujo de información dentro del propio compilador [LPR15]. En nuestro caso, obtenemos esta información estudiando manualmente los binarios generados. Encontramos tres patrones en los códigos vectorizados. El primero es el que se muestra en la figura 4.1, donde el bucle escalar original se transforma en un bucle vectorial más un bucle escalar a modo de epílogo. En este caso, o el compilador no conoce el número de iteraciones del bucle o este no es divisible por el número de vías vectoriales. El peor caso de este escenario correspondería a aquella situación donde se ejecuta el máximo número de iteraciones posibles en el bucle de epílogo. Este caso continuaría siendo el peor incluso aunque el número de elementos del bucle fuera menor que el número de vías vectoriales para ese tipo de dato. Por lo tanto, encontramos una cota del número máximo de iteraciones en cada bucle en las siguientes expresiones:

$$\begin{aligned} \text{Núm. iteraciones bucle vectorial} &\leq \left\lfloor \frac{\text{Núm. iteraciones bucle escalar}}{\text{\# vías vectoriales para el tipo de dato}} \right\rfloor \\ \text{Núm. iteraciones bucle epílogo} &\leq \text{\# vías vectoriales para el tipo de dato} - 1 \end{aligned}$$

En el segundo patrón, el bucle escalar original ha sido transformado simplemente en un bucle vectorizado, sin ningún bucle escalar a modo de epílogo. En este caso, el compilador conoce que el número de iteraciones de bucle original es múltiplo del número de vías, así que no produce ningún bucle extra. En este escenario, el número de iteraciones del bucle vectorizado es:

$$\text{Núm. Iteraciones bucle vectorizado} = \frac{\text{Núm. Iteraciones bucle escalar}}{\text{\# vías vectoriales para el tipo de dato}}$$

Finalmente, en el último patrón encontrado en nuestros experimentos el bucle escalar ha sido transformado en un bucle vectorizado, otro escalar a modo de epílogo y un tercer bucle. Este último y su camino correspondiente coincide con el bucle original sin modificar. En este caso el compilador ha mantenido en el código el bucle original además de la versión vectorizada como dos caminos alternativos. Por lo tanto, el peor caso en este escenario será que el camino atraviese el bucle original, así que la vectorización no aporta ninguna mejora en términos del WCET.

4.4. Entorno experimental

En esta sección describimos el conjunto de herramientas utilizadas para compilar los binarios de la arquitectura ARMv6, junto con las opciones de compilación. También presentamos la selección de benchmarks y de las latencias de instrucciones utilizadas en nuestros experimentos.

4.4.1. Compilación

Para poder evaluar la mejora que la vectorización supone al WCET de un programa, generamos binarios ARMv6 con una herramienta de compilación cruzada x86-ARM. En concreto, utilizamos *arm-none-eabi* v. 4.8.4 de GNU Arm Embedded Toolchain [Fre12]. Para cada benchmark generamos dos binarios distintos. Uno en el que le indicamos al compilador con opciones de compilación (*-ftree-vectorize* y *-mfpu=neon*) que buscara automáticamente los bucles potencialmente vectorizables, y otra donde evitara esta vectorización (*-ftree-no-vectorize*). También determinamos otras opciones de compilación en ambas versiones para tener una versión base escalar que fuera competitiva (*-O2 -march=armv6 -g -static -lm*). Finalmente, utilizamos una opción de compilación que permite conocer la eficiencia del compilador en los bucles auto-vectorizados (*-fopt-info-vec-optimized*).

4.4.2. Modelo del procesador y latencias de las instrucciones vectoriales

Nuestro modelo de procesador experimental consiste en un camino de datos multiciclo simple. Cada instrucción recorre el camino de datos en orden secuencial y no se permite empezar su recorrido a ninguna instrucción hasta que haya finalizado la anterior. Cada instrucción, tanto escalar como vectorial, supone un ciclo de acierto en búsqueda. Si el peor caso para un instrucción, tal como se presenta en el análisis

TABLA 4.1: Latencia de las instrucciones escalares y vectoriales más comunes, en ciclos.

Escalar	Latencia (c)	Vectorial	Latencia (c)
multiplicación	4	multiplicación	5
aritmético-lógica	1	aritmético-lógica	1
lectura, escritura	2	lectura, escritura	4, 5
lec. o escr. múltiple	$min(2, \lceil \#registers/2 \rceil)$	conversión	1

TABLA 4.2: Benchmarks utilizados en la experimentación del capítulo 4.

Nombre	Tamaño código binario (Bytes)	Vect	No-Vect.	Vías
audiobeam	3808	2	4	4
basicmath	4520	1	3	16
bsort	200	1	3	4
cnt	256	1	3	4
complex_updates	376	1	3	4
countnegative	300	1	3	4
fft	840	3	8	4
filterbank	688	4	9	4
fir2dim	640	3	14	4
fmref	3280	1	10	4
g723_enc	2780	2	7	4
jfdctint	868	4	0	4,16
lift	2292	1	9	4
matmult_opti	204	2	3	4
matrix1	240	4	3	4
md5	3124	4	2	4,16
pm	3704	5	23	4,16

del WCET de la sección 4.3, es un fallo en la cache de instrucciones, se añaden 10 ciclos al coste de búsqueda.

Nuestro modelo toma NEON como extensión vectorial del repertorio de instrucciones que está definida por ARM con registros vectoriales de 128 bit [Pye16, Capítulo 10]. Por lo tanto, un registro vectorial puede almacenar cuatro, ocho o dieciséis enteros de 32-bit, 16-bit o 8-bit respectivamente, así como cuatro datos de coma flotante de 32-bit. La tabla 4.1 muestra las latencias que asumimos de las instrucciones tanto escalares como vectoriales más comunes que hemos encontrado, tomándolas del procesador Cortex A8 [Lima; Limb]. En cuanto a las instrucciones de lectura y escritura de datos consideramos un coste fijo de acceso a memoria principal, que se añade a su latencia de ejecución.

En nuestros experimentos, este coste fijo ha sido de 1 ciclo que equivale a siempre acierto en cache, por ejemplo porque los datos están precargados en ella, o 10 ciclos que equivale a que no tener cache de datos y acceder siempre a memoria principal.

4.4.3. Benchmarks

La tabla 4.2 muestra los benchmarks que usamos en nuestros experimentos de este capítulo. En la sección 2.2 se puede encontrar su caracterización.

En la tabla encontramos bajo «Tamaño de código binario (Bytes)» el número de bytes de las instrucciones analizadas en el fichero binario. Estos es, la suma de todas las instrucciones en el CFG por su tamaño (4 bytes por instrucción). El código fuente no se ha modificado por lo tanto el compilador detecta por sí mismo las posibilidades de vectorización.

De todo nuestro repertorio de benchmarks encontramos que el compilador no ha producido ninguna vectorización del código en 18 de ellos[1]. Este hecho nos señala que es necesario seguir trabajando para identificar si esta falta de paralelismo vectorial es real o si por el contrario puede subsanarse con transformaciones sencillas del código o de los datos. Debido a que nuestro propósito es comparar benchmarks en sus versiones escalar y vectorizada, aquellos en los que la vectorización no ha surtido efecto se han descartado. Además en este capítulo utilizamos el benchmark *matmult_opti* en vez de *matmult* para facilitar su vectorización.

Cabe destacar que aunque hay varios benchmarks de la suite TACLeBench cuyo algoritmo coincide con otro benchmark de Mälardalen (*binarysearch/bs*, *countnegative/cnt*, *jfdctint/fdct*, *petrinet/nsichneu*), su comportamiento al ser vectorizados por el compilador no coincide ya que este resultado depende en gran medida del código en alto nivel. De hecho, de estas duplas, la única que de la que se obtiene una vectorización de la suite de Mälardalen es el benchmark *cnt*. Este hecho refuerza la idea de la importancia de una programación que facilite la vectorización.

El número de iteraciones de los bucles escalares que se dan en el peor caso está anotado en el código fuente de los programas. Estas anotaciones o bien se encontraban ya en el código original del benchmark o las hemos añadido nosotros manualmente. En el caso del número de iteraciones en los bucles vectoriales se han calculado tal como se expone en la sección 4.3.1.

4.5. Resultados experimentales

En esta sección, evaluamos el impacto que tiene en el WCET la vectorización de código en los benchmarks de tiempo real. Calculamos el WCET de nuestros benchmarks con nuestro entorno de análisis estático de binarios (sección 4.3) basado en OTAWA [Bal+10] y Lock-MS [Apa+11]. Para cada uno de los benchmarks producimos dos binarios distintos. Uno se ha generado activando la vectorización automática de bucles del compilador GCC 4.8.4 mientras que el otro binario se ha generado evitando esta vectorización. El resto de opciones de compilación están detalladas en la sección 4.4.1.

En las tres columnas más a la derecha de la tabla 4.2 encontramos información adicional que nos permite mostrar la eficacia del compilador GCC 4.8.4 vectorizando los bucles del código fuente de cada benchmark.

En concreto, se muestra el número de bucles que han sido vectorizados por el compilador (Columna *Vect.*), el número de bucles que no se han vectorizado (Column *No-Vect.*), y el número de elementos que se procesan simultaneamente en cada iteración de los bucles vectorizados (Columna *Vías*).

En el caso de *jfdctint*, *md5* y *pm*, los tipos de dato de los bucles vectorizados son enteros de 32-bit y caracteres así que el número de elementos que se procesan simultáneamente son 4 y 16, respectivamente. El hecho de que la vectorización automática de GCC 4.8.4 haya funcionado en solo 17 de 35 benchmarks, parece indicar que la autovectorización de este compilador es bastante limitada. Seguramente reescribiendo algunas partes del código original podríamos facilitar esta vectorización automática, pero esta tarea va más allá del propósito de este trabajo.

La figura 4.3 muestra la mejora relativa del código vectorizado sobre el código no vectorizado para cada benchmark. Asumimos una cache de instrucciones bloqueable de mapeo directo de 128 bytes, con líneas de cache de 32 bytes. También

[1] *bynarysearch, bs, crc, dijkstra, fdct, iir, janne_complex, ludcmp, matmult, ndes, nsichneu, petrinet, qsort-exam, qurt, select, st, statemate, minver*.

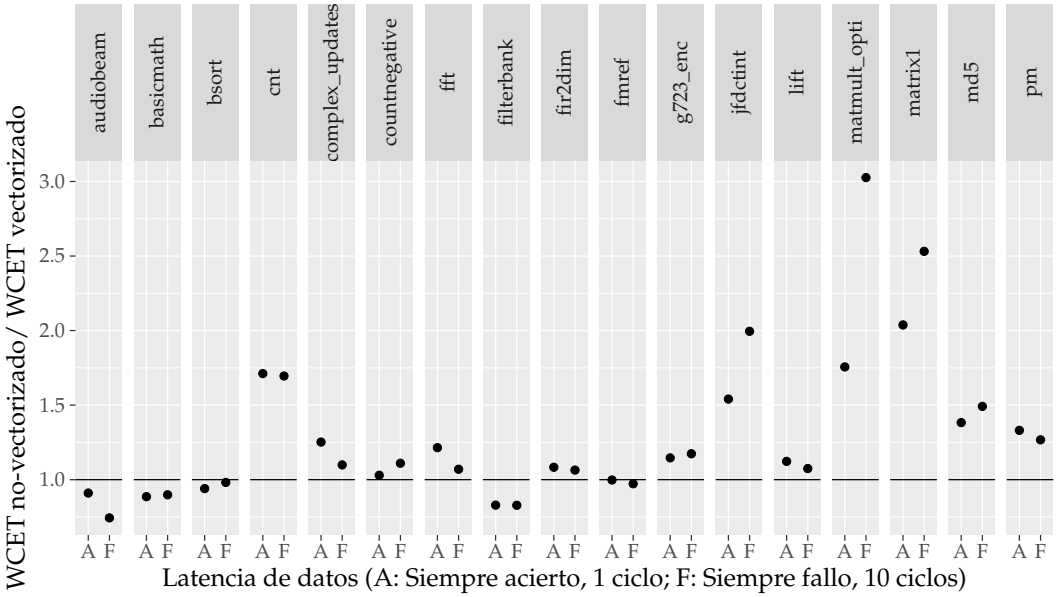


FIGURA 4.3: Comparación del impacto de la vectorización en el WCET.

asumimos la presencia de un *line-buffer*: una cache convencional (dinámica) de instrucciones de solo una línea de cache [Apa+11; PD02]. Por tanto, el método de análisis del WCET contabiliza los aciertos (1 ciclo) y fallos (11 ciclos) en la búsqueda de instrucciones.

Consideramos dos escenarios diferentes que dan una perspectiva de la importancia de la cache de datos (ya sean escalares o vectoriales). El primer escenario (*A*) corresponde a una jerarquía de cache de datos ideal donde cada acceso a datos resulta siempre en acierto. En el segundo escenario (*F*), los accesos a datos siempre resultan en fallo y se penalizan con 10 ciclos debido a la latencia de memoria. Probablemente este último escenario se acerca más a la realidad ya que normalmente las instrucciones vectoriales en los procesadores comerciales actuales omiten a la cache de datos L1 y solicitan los datos que necesitan directamente a L2.

Destacan los resultados de la vectorización en el caso de *matmult_opti* y *matrix1* ya que son hasta 3 veces mejor que la versión no vectorizada. Este valor es muy cercano al máximo esperado cuando 4 elementos se procesan simultáneamente, que resultaría una mejora de 4 (ver tabla 4.2). Al inspeccionar manualmente el binario vectorizado descubrimos que el bucle que contiene la mayor carga computacional se ha vectorizado especialmente bien y que no se ha necesitado un bucle de epílogo. Estos casos muestran que las instrucciones vectoriales reducen de forma efectiva el WCET. Para otros benchmarks como *fft*, *g723-enc* y *lift*, a pesar de que algunos de sus bucles se han vectorizado, no consiguen una mejora mayor del 1.25. En esta ocasión, los bucles vectorizados no forman parte de donde se concentra la mayor parte de la ejecución, así que la mejora es menor.

Desafortunadamente, en algunos benchmarks, (*audiobeam*, *basicmath*, *bsort* and *filterbank*) el rendimiento del binario vectorizado es peor que el no vectorizado. En estos casos el número de iteraciones es especialmente pequeño comparado con otros benchmarks. Por ejemplo, el bucle vectorial en *basicmath* se ejecuta una sola vez. Este no puede compensar el sobrecoste generado en el código al vectorizar el bucle. Esto es, el código en el que se elige entre el bucle escalar o vectorial, las instrucciones adicionales que los bucles vectoriales necesitan para preparar el recorrido del bucle

(por ejemplo, la propagación de constantes en las vías de un registro vectorial) y los fallos adicionales en la cache de instrucciones que el código vectorial pueda causar.

De todos modos, nuestra recomendación es comparar los WCET de las versiones escalares y vectoriales y elegir el mejor de ellos, es decir, el menor.

4.6. Conclusiones

En este capítulo analizamos la eficacia del código vectorizado en el WCET estudiando las prestaciones de la vectorización automática de GCC en paquetes de benchmark de tiempo real. Tras analizar los código vectoriales, se integran sus especificaciones temporales correspondientes en la herramienta de análisis del WCET.

Nuestros resultados muestran que el WCET se reduce de manera efectiva cuando los bucles vectorizados forman parte del código dónde se lleva a cabo la mayor parte de la ejecución. Por ejemplo, en la versión vectorial del benchmark *matmult_opti* el WCET se reduce por un factor de 3 respecto a la versión escalar. Sin embargo, sus beneficios no resultan tan evidentes como se esperaba, debido, en parte, a la eficacia limitada que ofrecen las herramientas de compilación en la vectorización automática. Se podría solventar reescribiendo el código fuente de manera que se facilitara la vectorización automática o incluso escribiendo explícitamente el código vectorial, pero la eficacia de esta vectorización dependerá de la habilidad de quien programa.

El número de iteraciones de los bucles en algunos de los benchmarks de tiempo real que hemos analizado es bastante pequeño, probablemente porque estos benchmarks son simples *kernels*, es decir, operaciones/algoritmos que forman parte de los núcleos de programas de tiempo real mucho mayores. Por lo tanto, el número de iteraciones no es lo suficientemente grande para compensar el sobrecoste de utilizar código vectorial. En estos casos, se debe evitar la vectorización. Como conclusión, pensamos que nuestros resultados fomentan el uso de los recursos vectoriales en sistemas de tiempo real estricto, posiblemente escribiendo códigos con el paralelismo de datos en mente, tal como puede ser el benchmark *matmult_opti* que se ha presentado en la sección 4.4.3. Actualmente, como los recursos del chip continúan creciendo, el procesamiento vectorial esta convirtiéndose en una utilidad básica en los mayoría de los procesadores del mercado. Este trabajo podría continuarse en el futuro examinando con más detenimiento los benchmarks que no han podido vectorizarse, para encontrar a qué se debe. Para ello se tendría que evaluar la cantidad de paralelismo en datos y comprobar el estilo de programación buscando construcciones que dificulten la vectorización tal como punteros, reuso escalar a través de distintas iteraciones, vectores no alineados, etc.

Capítulo 5

Tasa ideal y predecible de aciertos para la transposición de matrices en caches de datos

Tras concluir en los capítulos anteriores el gran impacto de la latencia en la jerarquía de memoria en el WCET y explorar distintas vías tanto para agilizar su análisis como para reducir el propio WCET, en este capítulo, analizamos un algoritmo fundamental en los sistemas de tiempo real: la transposición de matrices. Analizamos el número de accesos y tasa de aciertos en cache de datos, cuyo efecto es de gran relevancia en el cálculo de su WCET.

Una transposición de matrices es una operación básica. Sin embargo, su tasa de aciertos en cache de datos para grandes matrices es muy baja y no se puede predecir fácilmente. En el contexto de tiempo real es imprescindible que la tasa de aciertos (su peor caso) se pueda predecir de manera segura. En este capítulo obtenemos la relación entre los parámetros de cache que garantizan la tasa (predecible) ideal de aciertos en datos tomando una cache de datos LRU.

Tras ello y teniendo en cuenta nuestras valoraciones analíticas comparamos los algoritmos de transposición de matrices *tiling* y *cache-oblivious*. El resultado de esta comparación demuestra que, con el tamaño adecuado de tile, la versión *tiling* del algoritmo obtiene una tasa de aciertos en datos igual o mejor. Adicionalmente analizamos el consumo de energía y el tiempo de ejecución de la transposición en hardware real con caches PLRU.

Gracias a que nuestro análisis de aciertos y fallos proporciona cotas para el peor caso podemos permitir el uso de caches de datos LRU (poco predecibles en general) para la transposición de matrices en sistemas de tiempo real. Además, nuestras valoraciones analíticas permiten restringir sin ningún impacto negativo los recursos dedicados a la transposición de matrices. Se consigue así reducir tanto la contaminación a otros procesos como el consumo de energía, lo que es realmente útil en general y específicamente en la computación de alto rendimiento.

5.1. Introducción

La transposición de matrices es una operación fundamental en áreas tales como el álgebra lineal o las transformadas de Fourier. Además tiene muchas aplicaciones en otras como el análisis numérico, el procesado de imágenes y gráficos. Hay muchos ejemplos de aplicaciones desarrolladas en los centros de supercomputación que usan, de un modo u otro, la transposición de matrices como parte esencial para solucionar sus problemas [RJ16; Bal+16].

Aunque la transposición en sí misma es un problema bastante simple, cuando tomamos matrices de gran tamaño presenta una tasa de aciertos en cache de datos muy pequeña [CS00]. Esto se debe a que aunque los elementos por filas son accedidos consecutivamente se intercalan con el recorrido por columnas que conlleva la transposición. El que los elementos se accedieran consecutivamente por filas haría posible que encajasen en la misma línea de cache y por lo tanto hubiera reuso temporal. Sin embargo, la interferencia del acceso por columnas conlleva que los aciertos potenciales acaban generando fallos por capacidad.

Existen transformaciones de código conocidas y usadas de manera habitual para solventar este problema. La transformación *tiling* o *blocking* está presente en las bibliotecas de alto rendimiento (p.e. Intel MKL, NVIDIA cuBLAS) y consiste en dividir todo el problema en pequeños *tiles* («teselas») que quepan en la cache [LRW91] y con los que se puede trabajar de manera independiente. Al transponer los tiles completamente uno detrás de otro (o por parejas) permite que los tiles implicados permanezcan en cache mientras se están procesando, evitando así los fallos por capacidad.

Aunque aplicar eficientemente una transformación de *tiling* incrementa la tasa de aciertos en la cache de datos todavía podemos encontrar dos importantes desventajas. Primero, añadir bucles implica añadir más instrucciones para poder llevar a cabo la transposición de matrices, con su correspondiente tiempo de ejecución. La segunda desventaja que tenemos que considerar es que la tasa de aciertos del código transformado está influenciada por el tamaño de la matriz a transponer y por la configuración específica de la cache del sistema correspondiente (es decir, número de conjuntos y vías, tamaño de línea de cache, política de reemplazo, prebúsqueda, cache víctima, etc.). Estos inconvenientes se han abordado desde la perspectiva de optimización de compiladores tratando de minimizar el número de fallos en cache que se generan desde el código binario [Bao+18].

Cuando trabajamos con sistemas de tiempo real estas desventajas se vuelven críticas ya que se debe concretar el peor tiempo de ejecución (WCET) en el momento del diseño [Rei+07].

Por esta razón si los aciertos y fallos no se pueden predecir con antelación el sistema de tiempo real se diseñará asumiendo una sobre-estimación en su comportamiento, lo que nos llevará a desaprovechar los recursos hardware e incrementar el consumo de energía. Además, los sistemas de tiempo real necesitan predicciones seguras así que no se pueden aplicar técnicas que no garanticen el resultado del peor caso. También estos inconvenientes son importantes en la computación en la nube ya que las configuraciones de las caches de las máquinas virtuales utilizadas y conocidas pueden no coincidir con las máquinas físicas y además ser desconocidas.

Por otro lado podemos usar un algoritmo *cache-oblivious* de transposición de matrices. Este algoritmo es, esencialmente, una versión recursiva del algoritmo de *tiling* (hasta alcanzar tiles de 2×2 elementos) por lo tanto el rendimiento óptimo de la cache se alcanza por su propia naturaleza recursiva y su configuración no depende de los parámetros de la cache.

Como concluyen Tsifakis et al [TRS04]: «No es trivial predecir a priori como va a ser el rendimiento del algoritmo *cache-oblivious*», por lo tanto persisten las desventajas anteriormente presentadas. No obstante este algoritmo evita tener que añadir un parámetro extra que tener en cuenta: el tamaño de *tile*.

Independientemente de como implementemos la transposición de matrices, hasta donde alcanza nuestro conocimiento, no hay ningún trabajo previo que proporcione una valoración analítica de aciertos/fallos de este problema. Sin este análisis se puede estudiar la tendencia del rendimiento para unos parámetros específicos

de cache pero no va a ser posible ajustar de forma precisa estos parámetros y de la misma manera tampoco se va a poder dar una predicción concisa del rendimiento.

Para abordar este problema estudiamos la transposición de matrices desde una perspectiva teórica y cómo afectan a la tasa de aciertos tanto los parámetros de cache (número de conjuntos, vías y tamaño de línea) como el *padding* («relleno») aplicado a la matriz para el algoritmo *tiling* y una cache con política de reemplazo LRU.

Validamos las expresiones analíticas obtenidas en nuestro estudio a través de simulaciones en las que consideramos un amplio rango de parámetros. Y también comparamos nuestros resultados con los de una implementación *cache-oblivious* mejorada con una nueva técnica de *padding* que hemos denominado *phantom padding* («relleno fantasma»). Para terminar analizamos el rendimiento de una transposición de matrices en hardware real (con una configuración específica de cache) que nos permitirá verificar si se cumplen nuestras predicciones teóricas.

El resto del capítulo se estructura de la siguiente manera. En la sección 5.2 presentamos el trabajo relacionado con el estudio del comportamiento en cache de la transposición de matrices. En la sección 5.3 presentamos el orden de acceso de las diferentes implementaciones de la transposición de matrices. A continuación en la sección 5.4 calculamos las tasas ideales de acierto para la transposición de matrices independientemente de la cache y del tipo de algoritmo. En la sección 5.5 analizamos qué configuraciones de cache LRU son necesarias para alcanzar estas tasas. En la siguiente sección 5.6 proponemos un *padding* extra que permite alcanzar la tasa ideal de aciertos con menos recursos de cache. Todas las expresiones analíticas obtenidas en estas secciones son validadas por medio de simulaciones extensivas en la sección 5.7. En esta sección, además, los resultados obtenidos se comparan con aquellos proporcionados por el algoritmo *cache-oblivious*. Finalmente, las conclusiones de este capítulo se exponen en la sección 5.8.

5.2. Estudios previos

En esta sección examinamos algunos trabajos que analizan el comportamiento de aciertos y fallos de la cache de varios algoritmos de transposición de matrices.

Debido a que el algoritmo de transposición de matrices es bastante simple y hace un uso intensivo de memoria numerosos trabajos se han centrado en tratar de analizar y mejorar el uso de la jerarquía de memoria.

Se han investigado dos alternativas para gestionar el patrón de accesos en una cache para este algoritmo tal y como se han introducido en este capítulo: *tiling* y *cache-oblivious*. Ambos enfoques buscan exponer a la jerarquía de memoria a un patrón de acceso de datos que explote el reuso (espacial y temporal) en la cache subyacente [Fri+99; LRW91; CS00; TRS04; Yot+07; Lei03; Fri+12]. A continuación, presentamos las diferencias entre los trabajos relacionados y el nuestro.

Cache-Efficient Matrix Transposition Reference [CS00] describe varios algoritmos de transposición de matrices y compara su rendimiento usando tanto simulación como ejecuciones reales en un sistema basado en *Sun UltraSPARC II*. A través de las simulaciones se muestra que el algoritmo *cache-oblivious* es el que tiene menos número de fallos para matrices de pequeño tamaño. Sin embargo, ocurre al contrario para matrices grandes ya que es el que presenta mayor número de fallos. Además, los tiempos de ejecución presentados muestran que, en la mayoría de los casos, el algoritmo *cache-oblivious* es significativamente más lento que los otros algoritmos analizados. En el estudio se sugiere que la razón por la que el rendimiento no es el esperado es porque no hay suficiente asociatividad. Si lo comparamos con nuestro

enfoque, su análisis se limita a una evaluación experimental sin proporcionar ninguna valoración analítica de aciertos y fallos de los diferentes algoritmos.

Cache Oblivious Matrix Transposition: Simulation and Experiment Reference [TRS04] analiza en mayor profundidad el algoritmo de transposición de matrices *cache-oblivious* con la intención de racionalizar los resultados de Chatterjee y Sen [CS00]. Estudian el rendimiento del algoritmo, con respecto a los fallos en cache, tanto con simulación como con contadores hardware en dos sistemas *Sun UltraSPARC* completamente distintos entre sí. Como en nuestro trabajo, comparan los algoritmos *tiling* y *cache-oblivious* pero se centran en el comportamiento de este último. Sin embargo, analizan solo una configuración específica de cache y un solo tamaño de *tile*, variando únicamente el tamaño de la matriz. Sus resultados muestran que los fallos en cache se caracterizan por un patrón bastante estructurado que depende de la configuración de la cache y del tamaño de la matriz. No obstante, no determinan cuándo el algoritmo *cache-oblivious* va a tener un buen o un mal rendimiento, sino solamente que incrementando el número de vías de la cache se mejora la tasa de aciertos. En nuestro trabajo, evaluamos analíticamente esta cuestión.

The Cache Performance and Optimizations of Blocked Algorithms Reference [LRW91] se centra en optimizar el rendimiento de la cache a través del transformación *blocking (tiling)* de algoritmos. Su enfoque consiste en, primero, mostrar el comportamiento de caches bajo *tiling*, para después mejorar su rendimiento a través de técnicas software y/o hardware. Analizan el algoritmo de multiplicación de matrices en su versión *tiling* y concluyen que el rendimiento de la cache es extremadamente dependiente del tamaño del problema y del *tile*. También afirman que las tasas de fallo tienen una gran sensibilidad con respecto al tamaño de la matriz. Sin embargo, como veremos en nuestra propuesta, un ajuste preciso del algoritmo *tiling* de transposición proporciona a quien programa muchos más grados de libertad.

An Experimental Comparison of Cache-oblivious and Cache-conscious Programs Reference [Yot+07] compara de manera experimental un programa *cache-oblivious* con uno *cache-conscious* (cache consciente, tiene aplicado la transformación de *tiling*) tanto para la multiplicación como para la transposición de matrices. Afirman que hay un sobrecoste que pagan los programas *cache-oblivious* por su habilidad para adaptarse automáticamente a la jerarquía de memoria. También determinan que incluso aquellos programas *cache-oblivious* que están altamente optimizados tienen un rendimiento significativamente peor que su respectivo programa *cache-conscious*.

Sin embargo, a diferencia de nosotros, no analizan un extenso rango de configuraciones de cache ni de tamaños de *tile* debido a limitaciones experimentales.

En general, consideramos al comparar estos trabajos que para llevar a cabo sus experimentaciones e implementaciones muchos de ellos usan un tamaño de matriz múltiplo del tamaño de línea de la cache y del tamaño de *tile* lo cual facilita el análisis [CS00; LRW91; TRS04] y otros consideran el *padding* como una técnica para evitar esta restricción pero no evalúan analíticamente sus efectos [Yot+07].

De todos modos, ninguno ha considerado las colisiones en los conjuntos de la cache en el caso de filas consecutivas de la misma columna, que es el caso principal de acceso a las columnas en la transposición. Nosotros abordamos esta cuestión añadiendo un *padding* adicional, el *row-shift padding* [Hon+16; Pan+99; Bac+94]. Existe otro problema más para el algoritmo *cache-oblivious* de transposición de matrices, que hará que se beneficie de la aplicación de un *phantom padding* (hasta dónde sabemos una propuesta original de nuestro trabajo) que también analizaremos en este capítulo.

5.3. Implementaciones de la transposición de matrices

En esta sección examinamos los recorridos de las diferentes implementaciones de la transposición de matrices ya introducidas en este capítulo. Estas propuestas pretendenden incrementar la tasa de aciertos en cache de datos (objeto de nuestro estudio) variándo el orden de acceso a los elementos de la matriz y, por tanto, a las líneas de cache que los contienen determinando su presencia en la cache (con el consecuente fallo o acierto en el acceso) según la política de reemplazo.

En la figura 5.1 se muestran el orden de acceso a los elementos de una matriz 8×8 para cada uno de los algoritmos:

- *Simple*: Recorre la matriz por filas intercambiando sus elementos en la posición (i, j) por aquellos en la posición (j, i). Esta implementación es simple pero presenta una tasa de aciertos muy pequeña en grandes matrices ya que no se beneficia del reuso temporal. En la figura 5.1(a) se muestra este comportamiento.

- *Tiling*: divide todo el problema en pequeños *tiles* («teselas») que quepan en la cache [LRW91] y con los que se puede trabajar de manera independiente. Esto permite un aumento de la tasa de aciertos gracias al reuso temporal. Básicamente esta transformación consiste en añadir bucles externos al código original de modo que la secuencia de accesos globales no se extiende a toda la memoria sino que se localiza en el proceso de cada tile. En la figura 5.1(c) encontramos un ejemplo con tamaño de tile 4×4 elementos. La matriz se divide en 4 tiles, aquellos que pertenecen a la diagonal se transponen sobre si mismos y los otros se transponen entre sí por parejas (siguiendo uno un recorrido por filas, y el otro por columnas). En cada tile se aplica internamente el recorrido del algoritmo «Simple». Análogamente, en la figura 5.1(d) se muestra el algoritmo *tiling* si determinamos un tamaño de tile 2×2 elementos.

- *Cache-oblivious:* Este algoritmo es, esencialmente, una versión recursiva del algoritmo de *tiling* por lo tanto el rendimiento óptimo de la cache se alcanza por su propia naturaleza recursiva. Es decir, un algoritmo *cache-oblivious* no precisa conocer explícitamente ningún parámetro de la configuración de la cache.

  Tal como se muestra en la figura 5.1(b), el algoritmo divide primero la matriz en tiles de 4×4 elementos para dividir cada una en tiles de 2×2. La principal diferencia en el recorrido con el algoritmo *tiling* con tiles de 2×2 elementos es el orden en la que recorre los tiles, como se puede apreciar en la figura.

5.4. Tasas ideales en cache de datos para la transposición de matrices

La tasa de aciertos en cache de datos es el porcentaje de accesos de datos que producen aciertos en cache. Llamamos tasa ideal de aciertos a aquella que se produciría en una cache de capacidad ilimitada. Concretamente consideramos una cache con un número ilimitado de líneas, inicialmente vacía, y con un tamaño de línea capaz de almacenar L elementos de la matriz a trasponer ($1 \leq L \in \mathbb{N}$). Además, la tasa ideal de aciertos solo tiene en cuenta los fallos obligatorios y no los fallos por capacidad, por conflicto o la secuencia de accesos del algoritmo de transposición. No obstante, el algoritmo 2, que implementa una transposición de matriz directa, es decir, sin transformaciones ni optimizaciones, se puede usar como referencia.

0	1	3	5	7	9	11	13
2	0	15	17	19	21	23	25
4	16	0	27	29	31	33	35
6	18	28	0	37	39	41	43
8	20	30	38	0	45	47	49
10	22	32	40	46	0	51	53
12	24	34	42	48	52	0	55
14	26	36	44	50	54	56	0

(a) Simple.

0	2	6	10	26	30	34	38
1	0	8	12	28	32	36	40
5	7	0	4	42	46	50	54
9	11	3	0	44	48	52	56
25	27	41	43	0	14	18	22
29	31	45	47	13	0	20	24
33	35	49	51	17	19	0	16
37	39	53	55	21	23	15	0

(b) Oblivious.

0	1	3	5	13	15	17	19
2	0	7	9	21	23	25	27
4	8	0	11	29	31	33	35
6	10	12	0	37	39	41	43
14	22	30	38	0	45	47	49
16	24	32	40	46	0	51	53
18	26	34	42	48	52	0	55
20	28	36	44	50	54	56	0

(c) Tiling 4×4.

0	1	4	8	14	18	32	36
2	0	6	10	16	20	34	38
3	5	0	11	22	26	40	44
7	9	12	0	24	28	42	46
13	15	21	23	0	29	48	54
17	19	25	27	30	0	50	52
31	33	39	41	47	49	0	55
35	37	43	45	51	53	56	0

(d) Tiling 2×2.

FIGURA 5.1: Orden de acceso a los elementos de una matriz 8×8 para cada uno de los algoritmos estudiados de transposición de matrices. El color solo aporta una guía visual de los recorridos y divisiones por tiles.

Algoritmo 2 TransposicionDeMatriz($Matrix$, N): Transpone una matriz $N \times N$.

```
1: for indice1 ← 1 to N do                                  # N iteraciones
2:     for indice2 ← indice1 + 1 to N do                    # N iteraciones
3:         temp ← Matrix_{indice1,indice2}                  # Lectura de memoria
4:         Matrix_{indice1,indice2} ← Matrix_{indice2,indice1}   # Lectura y escritura de memoria
5:         Matrix_{indice2,indice1} ← temp                  # Escritura en memoria
6:     end for
7: end for
```

Crear modelos de casos ideales resulta útil para determinar las cotas del mejor escenario de tasa de aciertos dada una determinada configuración de cache, mapeo de elemento a línea o algoritmo. Comenzamos considerando que solo los elementos que se van a transponer en la matriz van a ser tenidos en cuenta para los fallos en cache, por lo tanto se excluyen los elementos de la diagonal y como resultado obtenemos una cota superior para cualquier tasa ideal de aciertos. A continuación mostramos también las tasas ideales de aciertos incluyendo los efectos del *padding*. Consideramos una matriz $N \times N$ para transponer, con sus elementos almacenados por filas, alineada con el tamaño de línea de cache. Todos los resultados que se presentan en este capítulo son válidos también para elementos almacenados por columnas pero, por simplicidad, trataremos solo uno de los casos. También asumimos que la matriz final, ya transpuesta, se sobrescribe en la matriz inicial y no se usan otras estructuras de memoria más allá de registros.

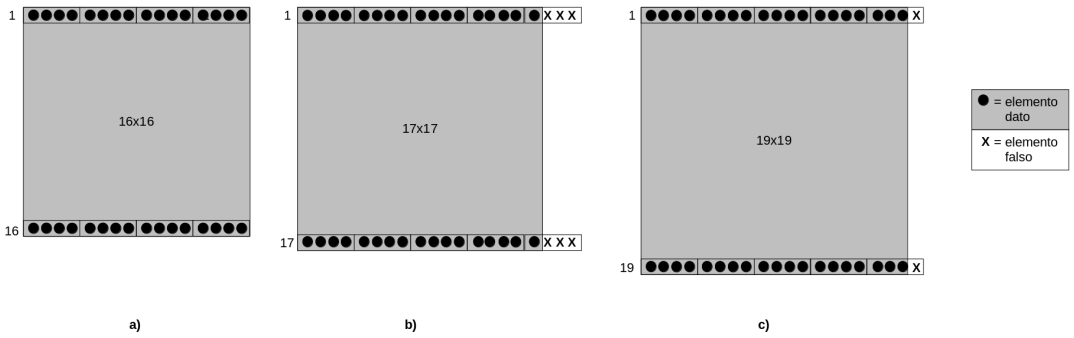


FIGURA 5.2: Diferentes mapeos de memoria cuando se aplica *padding* a una matriz $N \times N$ con líneas de cache que contienen L elementos: (**a**) $r = (N \bmod L) = 0$, (**b**) $r = 1$, (**c**) $r = 3$.

5.4.1. Cota de la tasa ideal de aciertos con una línea de cache que contiene L elementos

Teniendo en cuenta las consideraciones anteriores se puede calcular fácilmente una cota superior para la tasa ideal de aciertos. Exceptuando los elementos diagonales, todos los elementos $(N^2 - N)$ se acceden tanto para ser leídos como para ser escritos en memoria, lo que nos da $2(N^2 - N)$ accesos en total. Si asumimos que los $N^2 - N$ elementos caben perfectamente en las líneas de cache, tendremos $(N^2 - N)/L$ fallos obligatorios. Dividir por L implica implícitamente que los elementos de la diagonal no comparten su línea de cache con otros elementos. Aunque asumir esto no es realista (complicaría de hecho cualquier acceso indexado), proporciona la siguiente cota superior para la tasa de aciertos ideal:

$$1 - \frac{(N^2 - N)/L}{2(N^2 - N)} = 1 - \frac{1}{2L} \tag{5.1}$$

Sin embargo, para calcular otras tasas ideales de aciertos tendremos que analizar la relación entre el tamaño de la matriz y el tamaño de línea de cache.

5.4.2. Tasa ideal de aciertos en una matriz con su tamaño de fila múltiplo del tamaño de línea de cache ($r = 0$)

Suponemos que la matriz $N \times N$ a trasponer se almacena en memoria por filas y está alineada con el tamaño de línea de cache, incluyendo los elementos diagonales. También asumimos una línea de cache de datos que contiene L elementos consecutivos de una misma fila, con N múltiplo de L. Usando la notación de *módulo* $r = (N \bmod L) = 0$. Esto corresponde a la figura 5.2(a).

El número total de accesos es el mismo que en la ecuación 5.1. El número de fallos obligatorios en este caso es $(N/L)N$, es decir, el número de líneas de cache necesarias para recorrer una fila por el número de filas. Por lo tanto, la tasa ideal de aciertos es:

$$1 - \frac{N^2/L}{2(N^2 - N)} = 1 - \frac{1}{2L} - \frac{1}{2L(N-1)}. \tag{5.2}$$

5.4.3. Tasa ideal de aciertos en una matriz con *padding* ($r = 1$)

Cuando el número de elementos en una fila de una matriz (N) no es múltiplo del número que caben en una línea de cache (L) decrece el rendimiento. Una conocida transformación de datos es el *padding*, dónde cada fila en la matriz se extiende con $L - r$ elementos «falsos» para terminar de llenar una línea de cache. Además podemos garantizar que el primer elemento de cada fila siempre comienza en una nueva línea de cache gracias a que la matriz está almacenada de manera que sus direcciones están alineadas con las líneas de cache. Por lo tanto, para $r > 0$, cada fila produce un número obligatorio de fallos igual a $(N \text{ div } L) + 1 = N/L - r/L + 1$. En este apartado tomamos $r = (N \text{ mód } L) = 1$ así que se añaden $L - 1$ elementos de *padding* a cada fila de la matriz. Es importante señalar que para el caso $r = 1$ la última línea de cache solo almacena el último elemento diagonal, y por no tanto no será accedida. Podemos encontrar este caso en la figura 5.2(b). La tasa ideal de aciertos para $r = 1$ es:

$$1 - \frac{(N/L - 1/L + 1)N - 1}{2(N^2 - N)} = 1 - \frac{1}{2L} - \frac{1}{2N}. \tag{5.3}$$

5.4.4. Tasa ideal de aciertos en una matriz con *padding* ($r > 1$)

Para el resto de casos de *padding*, $1 < r = (N \text{ mód } L) < L$ $(r \in \mathbb{N})$, que se muestra en figura 5.2(c), la tasa ideal de aciertos es:

$$1 - \frac{(N/L - r/L + 1)N}{2(N^2 - N)} = 1 - \frac{1}{2L} - \frac{1 + L - r}{2L(N - 1)} \tag{5.4}$$

La figura 5.3 muestra la tasa ideal de aciertos para $L = 4$ elementos cuando se incrementa el tamaño de la matriz. Cada línea señala el r correspondiente para cada N, es decir, $r = 0$ corresponde con Ns múltiplos de L (ecuación 5.2), $r = 1$ representa la ecuación 5.3, y $r = 2$ y $r = 3$ representan la ecuación 5.4. Todas estas líneas tienden a la asíntota de tasa de aciertos en datos $1 - \frac{1}{2L}$ (ecuación 5.1).

5.5. Configuración de la cache LRU necesaria para obtener tasas ideales de aciertos

La sección anterior estudia las tasas ideales de aciertos en datos para las transposición de matrices. En esta sección analizamos como alcanzarlas con una cache LRU asociativa por conjuntos. Asumimos que la matriz se transpone dividiéndola por *tiles* («teselas»). Recordemos que los tiles son submatrices de $T \times T$ elementos que caben en la cache [LRW91] y con los que se puede trabajar de manera independiente (Sección 5.3). En el caso de la transposición de matrices, los tiles diagonales se transponen sobre sí mismos, y el resto se transponen por parejas según su posición correspondiente en la transposición. Esto corresponde al algoritmo *tiling* de transposición de matrices (algoritmo 3).

De esta manera, los tiles se van transponiendo completamente uno detrás de otro, permitiendo que cada tile permanezca en cache mientras se está procesando, lo que evita fallos por capacidad. Básicamente, esta transformación implica añadir bucles externos al código original de manera que la secuencia global de accesos no se expande sino que se localiza en el tile que está siendo procesado. A continuación, nos centramos en la relación entre el tamaño de la matriz ($N \times N$ elementos, almacenada

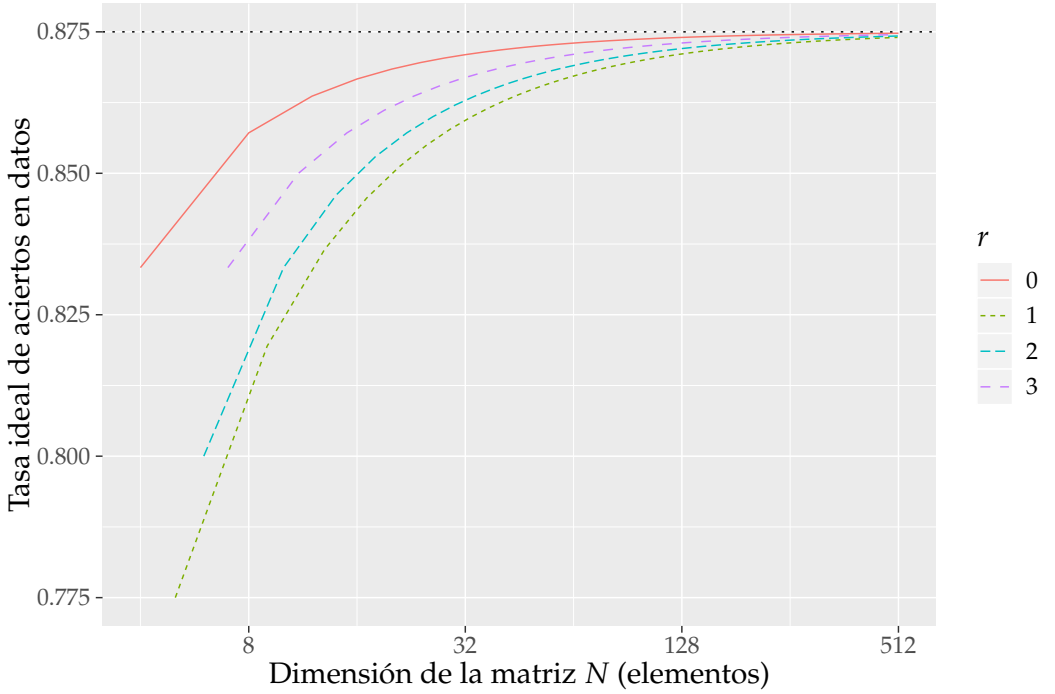


FIGURA 5.3: Tasa ideal de aciertos para $L = 4$ elementos con valores diferentes de r. La línea negra punteada muestra la cota superior de tasa de aciertos de la ecuación 5.1.

por filas), el tamaño del tile ($T \times T$ elementos) y la configuración de la cache LRU (S conjuntos, W vías por conjuntos, y el tamaño de línea que almacena L elementos).

Excepto los tiles con elementos de la diagonal, el resto se procesa por parejas de tiles. Uno de ellos se recorre por filas y el otro por columnas. En cada tile, todas las filas de T elementos horizontales se intercambian con los T verticales del otro tile. La tasa ideal de aciertos se alcanza cuando los recursos disponibles de cache son suficientes y se usan de manera eficaz. Es decir, el tamaño de línea de cache L y el número de conjuntos S deben ser suficientes para almacenar eficientemente en cache la transposición de cada par de tiles. Si no, la asociatividad de la cache (W) debe ser suficientemente grande para evitar los posibles fallos por conflicto. Para el tile que se recorre por filas basta con una sola línea de cache ya que cuando todos los elementos de una línea han sido transpuestos no se vuelve a acceder a ellos. Para el tile que se recorre por columnas, todas las líneas de cache que tienen los elementos de cada columna deben permanecer en cache hasta que todos los elementos de esas líneas hayan sido transpuestos, lo cual precisa o bien de un mínimo número de conjuntos o bien de asociatividad. Centrémonos en este último tile, el que es recorrido por columnas. Lo ideal sería que el número de conjuntos fuera igual al número de filas de cada tile, esto es, T conjuntos. Sin embargo, si varios elementos de una columna son mapeados en el mismo conjunto se necesitarán más conjuntos. Esto depende de la relación *máximo común divisor* entre número de líneas de memoria en una fila ($\lceil N/L \rceil$) y el número de conjuntos S. Si el número de conjuntos es suficiente, cualquier cache con mapeo directo ($W = 1$) bastará para almacenar las líneas de cache necesarias. De lo contrario, las líneas adicionales de cache tendrán que ser proporcionadas por un incremento en el número de vías W (donde cada vía consta de S líneas de cache)

De ahí que el número de vías necesarias para el tile recorrido por columnas es:

$$W_{PorColumnas} = \left\lceil \frac{T}{\left\lceil \frac{S}{mcd(\lceil N/L \rceil, S)} \right\rceil} \right\rceil \tag{5.5}$$

Algoritmo 3 Tiling($MatrizConPadding$, N, T): Transpone una $N \times (N + padding)$ matriz con *padding*, almacenada por filas, mediante $T \times T$ tiles [LRW91].

```
1:  for i ← 1 to N step T do                    # Recorrido triangular de la matriz, por tiles
2:    for j ← 1 to i step T do
3:      for indice1 ← i to mín(i + T − 1, N) step 1 do        # transpone tiles no diagonales
4:        for indice2 ← j to mín(j + T − 1, N) step 1 do
5:          temp ← MatrizConPadding_{indice1,indice2}
6:          MatrizConPadding_{indice1,indice2} ← MatrizConPadding_{indice2,indice1}
7:          MatrizConPadding_{indice2,indice1} ← temp
8:        end for
9:      end for
10:   end for
11:   for indice1 ← i to mín(i + T − 2, N) step 1 do      # transpone tiles diagonales
12:     for indice2 ← indice1 + 1 to mín(i + T − 1, N) step 1 do
13:       temp ← MatrizConPadding_{indice1,indice2}
14:       MatrizConPadding_{indice1,indice2} ← MatrizConPadding_{indice2,indice1}
15:       MatrizConPadding_{indice2,indice1} ← temp
16:     end for
17:   end for
18: end for
```

Recordemos que se necesita una línea extra de cache para el tile que se recorre por filas. Además, hay que considerar que el orden específico de la política de reemplazo LRU cuando se procesan *todas* las filas/columnas en un tile puede necesitar otra línea de cache adicional. Por ejemplo, una cache totalmente asociativa ($S = 1$) necesita 4 líneas para completar la transposición de un tile con $T = L = 2$. Así que, como mucho, serán necesarias dos vías adicionales para alcanzar la tasa ideal de aciertos:

$$W = \left\lceil \frac{T}{\left\lceil \frac{S}{mcd(\lceil N/L \rceil, S)} \right\rceil} \right\rceil + 2. \tag{5.6}$$

Dado que la ecuación anterior incluye muchas situaciones diferentes, es preciso un análisis más detallado para cada caso, el cual se podrá ajustar añadiendo más *padding*.

5.6. Reducción de los requisitos de caches LRU asociativas por conjuntos añadiendo padding

Los requisitos anteriores asumen una matriz donde cada fila ha sido completada (a través de la técnica *padding*) hasta llenar la línea de cache (Sección 5.4.3). En esta sección consideramos un *padding* extra para alcanzar la tasa ideal de aciertos con menos recursos de cache. Si es necesario, añadimos a cada fila un *row-shift padding*

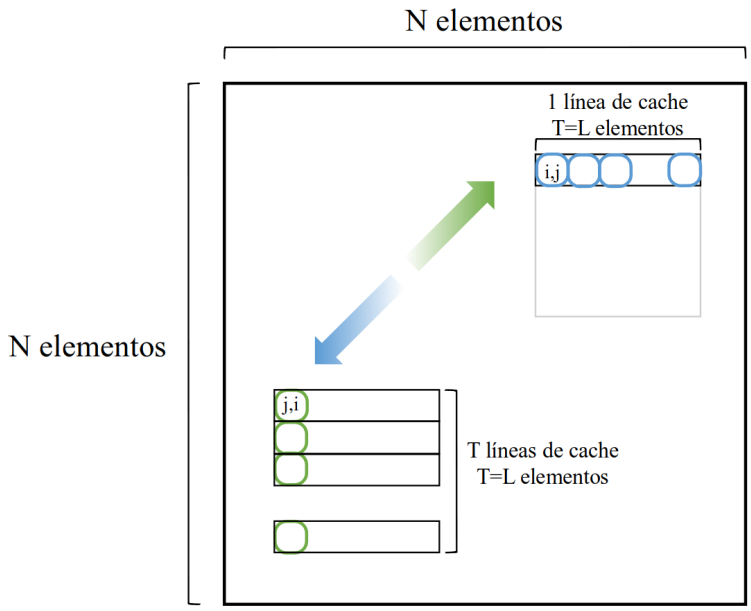


FIGURA 5.4: Transposición de una matriz $N \times N$ almacenada por filas asumiendo $T = L$.

(desplazador de fila) equivalente al tamaño de una línea de cache L para asegurar que los primeros elementos en dos filas consecutivas están mapeados en conjuntos distintos. Esta es, también, una ampliamente conocida transformación *padding* destinada a minimizar los fallos por conflicto [Hon+16; Pan+99; Bac+94]. De esta manera no hay conflictos entre las líneas de cache que tienen elementos de la misma columna, siempre y cuando $S \geq T$ ($mcd = 1$ en la ecuación 5.6).

Encontramos tres casuísticas diferentes dependiendo de la relación entre el tamaño del tile y el tamaño de la línea de cache, estos son: $T = L$, $T > L$, y $T < L$, con $T \geq 1$, $T \in \mathbb{N}$.

5.6.1. Cotas en W para alcanzar la tasa ideal de aciertos con $T = L$

El algoritmo 3 transpone una matriz tile por tile. En cada tile T elementos horizontales son intercambiados por T verticales. La figura 5.4 nos muestra un esquema de como, dado un tile, los elementos son intercambiados y almacenados en las líneas de cache. Con $T = L$, el *padding* (sección 5.4.3) garantiza que todos los elementos horizontales en el tile caben en una única línea de cache. También, con el *row-shift padding* [Hon+16; Pan+99; Bac+94] aseguramos que las líneas de cache que contienen elementos de la misma columna del tile son mapeadas en diferentes conjuntos de la cache. En el análisis que presentamos a continuación distinguimos varios casos según el número de conjuntos: $S \geq L$, $1 < S < L$ y $S = 1$.

Caso S ≥ L

Como $S \geq T$, el *row-shift padding* garantiza que el único conflicto posible implica a la línea de cache que contiene los elementos horizontales y una de las líneas de cache que contiene un elemento vertical. Este conflicto se evita con cualquier cache LRU con al menos 2 vías. La figura 5.5 muestra la localización de estas líneas de cache en un cache asociativa por conjuntos con $S \geq L$. Por lo tanto, la tasa ideal de aciertos (sección 5.4) se puede lograr con una cache LRU con $T = L$, $S \geq L$, $W \geq 2$,

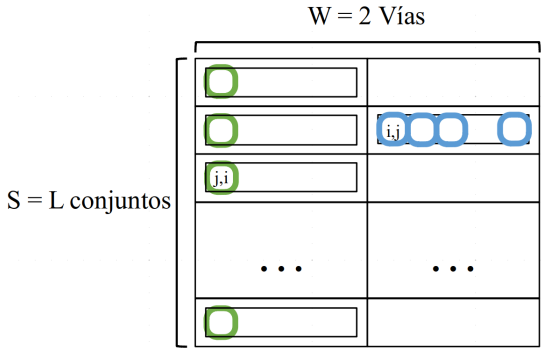


FIGURA 5.5: Cache asociativa por conjuntos almacenando un tile completo ($T = L$), con $S \geq L$ y $W = 2$.

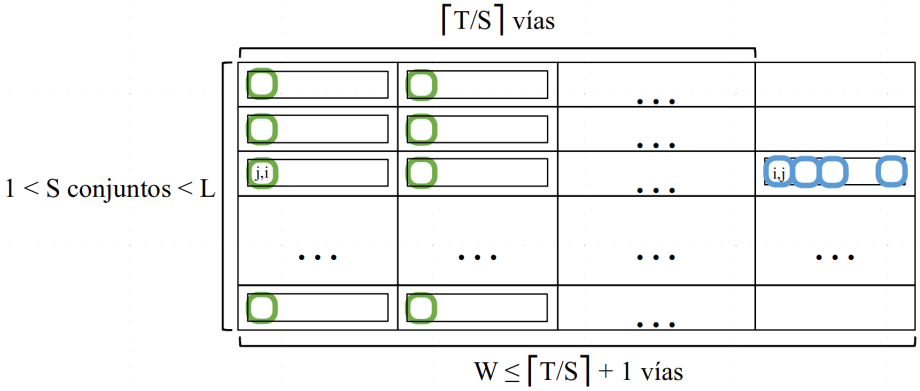


FIGURA 5.6: Cache asociativa por conjuntos almacenando un tile completo ($T = L$), con $1 < S < L$.

independientemente del tamaño de la matriz. En otras palabras, la asociatividad de la cache necesaria con $T = L$ y $S \geq L$ para alcanzar la tasa ideal de aciertos está acotada por 2:

$$W = 2, \text{ con } T = L,\ S \geq L. \tag{5.7}$$

Caso $1 < S < L$

En general, las caches tienen un número bastante elevado de conjuntos, así que la ecuación 5.7 modela los casos más comunes. Sin embargo, con la intención de llevar a cabo un análisis exhaustivo, consideramos el caso $S < L$. Con $S < L$ aparecerán nuevos conflictos entre las líneas que contienen elementos verticales a transponer. Por lo tanto, para poder alcanzar la tasa ideal de aciertos van a ser necesarias más vías que absorban esos conflictos.

Exactamente se necesitan $\lceil T/S \rceil$ vías para evitar los conflictos entre los elementos verticales más la vía adicional que evita los conflictos entre los las líneas que almacenan los elementos horizontales a transponer. La figura 5.6 muestra una representación de una cache para este caso. Con estas condiciones, la asociatividad necesaria para alcanzar la tasa ideal de aciertos está acotada por:

$$W = \left\lceil \frac{T}{S} \right\rceil + 1, \text{ con } T = L,\ 1 < S < L. \tag{5.8}$$

Caso S = 1

Finalmente, las caches totalmente asociativas ($S = 1$) constituyen un caso particular. Como se estudia en los casos anteriores la asociatividad debe ser lo suficientemente grande para proporcionar todas las líneas de cache necesarias para procesar cada fila/columna en un tile, esto es, $T + 1$ líneas de cache. Sin embargo, debido a la ordenación específica de la política de reemplazo LRU cuando se procesan *todas* las filas/columnas de un tile puede ser necesaria una línea extra de cache. Por lo tanto, la asociatividad de cache necesaria con $T = L$ y $S = 1$ para alcanzar siempre la tasa ideal de aciertos está acotada por:

$$W = T + 2, \text{ con } T = L, \ S = 1. \tag{5.9}$$

Por ejemplo, una cache totalmente asociativa necesita 6 líneas para llevar a cabo la transposición de un tile con $T = L = 4$. Usamos este ejemplo para mostrar la validez de la ecuación 5.9. La figura 5.7 muestra el mapeo en memoria de dos tiles (4×4 elementos) que se transponen en una matriz de $N \times N$ elementos. Las ocho líneas de cache involucradas en la transposición van de B_a a B_h. En cada línea un par de índices distintos identifican cada elemento a transponer. Para llevar a cabo la transposición de estos dos tiles la tabla 5.1 muestra la secuencia de instrucciones que acceden a memoria durante la transposición del tile: lecturas (ld) y escrituras (st) de los elementos $E_{fila,columna}$. Para cada paso la tabla 5.1 muestra una pila LRU de tamaño 6 ($T + 2$, en consonancia con la ecuación 5.9). En la parte de arriba, la posición MRU corresponde con la línea usada más recientemente (*most recently used*), es decir la línea que contiene el último elemento accedido para lectura/escritura. En la parte de abajo, la posición LRU corresponde a la línea de cache menos recientemente usada (*least recently used*). En una cache totalmente asociativa con una política de reemplazo LRU, la línea que está en la posición LRU es la candidata para ser reemplazada en caso de fallo.

En los pasos del 1 al 4 se representa la transposición de los elementos $E_{i,j}$ y $E_{j,i}$. Estos pasos se corresponden con las líneas 5 a 7 del algoritmo 3. En los pasos 1 y 2 se leen dos elemento en la matriz original de su correspondiente línea de cache, B_a y B_e, respectivamente. En los pasos 3 y 4 se almacenan estos elementos en su posición transpuesta de la matriz. Por ejemplo, la línea más recientemente usada en el paso 4 es B_e ya que la última instrucción ejecutada ("st $E_{j,i}$") almacena en $E_{j,i}$ (que se encuentra en B_e) el contenido previamente leído ("ld $E_{i,j}$" en el paso uno) de índice (i, j) (inicialmente en B_a). En el paso 16, todos los elementos en la línea de cache B_a se han transpuesto con sus correspondientes elementos de las líneas de cache $B_{[e,f,g,h]}$. En este punto, la pila LRU contiene 5 líneas de cache, esto es $T + 1$. Después (pasos del 17 al 32), los elementos en la línea B_b deben transponerse con sus correspondientes elementos de las líneas $B_{[e,f,g,h]}$. Para garantizar que ninguna de las líneas B_e, B_f, B_g, B_h resulta expulsada cuando la línea B_b llega, necesitamos una entrada extra, esto es $T + 2$. En el paso 32, la línea de cache menos recientemente usada es B_a que no se ha referenciado durante la transposición de la línea B_b en los pasos 16 a 32. Así que B_a puede ser expulsada sin pérdida en el paso 33, cuando se empieza la transposición la siguiente línea B_c.

5.6.2. Cota en W para alcanzar la tasa de aciertos ideal con $T > L$

Cuanto más grandes son los tiles menos instrucciones en los bucles exteriores del algoritmo 3 se necesitan. Esta relación resulta muy interesante para poder establecer las condiciones para alcanzar la tasa ideal de aciertos con tamaños de tile mayores

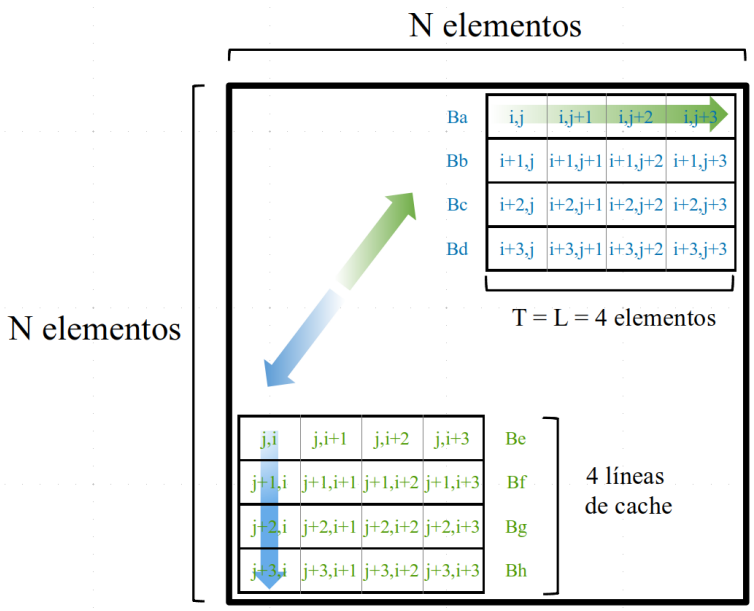


FIGURA 5.7: Mapeo en memoria de los bloques $B_{i\in[a-h]}$ que contienen cada uno, $T = L = 4$ elementos a transponer.

($T > L$). En estos casos, son necesarias $\lceil (T/L)/S \rceil$ líneas adicionales de cache que contengan elementos horizontales a transponer. Dependiendo de la relación entre los parámetros de la cache LRU y el tamaño de la matriz, la tasa ideal de aciertos se puede lograr con un número de vías ligeramente distinto. Por ejemplo, las matrices cuya dimensión (N) no es múltiplo de 2 necesitan una vía menos. De todos modos, la asociatividad necesaria para alcanzar la tasa ideal de aciertos para cualquier S está acotada por:

$$W = \left\lceil \frac{T}{S} \right\rceil + \left\lceil \frac{T/L}{S} \right\rceil + 1, \text{ con } T > L. \tag{5.10}$$

5.6.3. Cota en W para alcanzar la tasa ideal de aciertos con $T < L$

Es bastante poco práctico suponer que usamos una dimensión de tile cuyo número de elementos T es menor que la capacidad de una línea de cache L ya que esto implica que al procesar un tile de $T \times T$ elementos traemos a cache más contenido que el estrictamente necesario. Por lo tanto, para alcanzar la tasa ideal de aciertos este contenido que no se usa no puede ser expulsado hasta que se haya realmente usado. En otras palabras, se necesita una cache lo suficientemente grande para evitar fallos por capacidad. Concretamente, el número de líneas de cache ($S \times W$) tiene que ser mayor que el doble del número de columnas de la matriz ($2N$). Con este tamaño de tile tan inapropiado, el número de vías necesarias para alcanzar la tasa ideal de aciertos depende del tamaño de la matriz, acotándose de la siguiente manera:

$$W = \left\lceil \frac{2N}{S} \right\rceil + 1, \text{ con } T < L. \tag{5.11}$$

5.7. Experimentos

En esta sección validamos las expresiones analíticas que hemos obtenido previamente por medio de simulaciones extensivas. Calculamos la tasa de aciertos en

TABLA 5.1: Evolución del contenido de una pila LRU durante la transposición de un tile en una cache con $L = T = 4$ y $S = 1$. La captura de cada pila LRU se toma después de la ejecución de las instrucciones de lectura (ld) o escritura (st) que acceden a los $E_{fila,columna}$ elementos, y nos muestra la secuencia ordenada de todas las líneas de cache B_i accedidas previamente.

	Secuencia de accesos a memoria de las instrucciones de lectura (ld) y escritura (st)								
	1 ld $E_{i,j}$	2 ld $E_{j,i}$	3 st $E_{i,j}$	4 st $E_{j,i}$	5 ld $E_{i,j+1}$	6 ld $E_{j+1,i}$	7 st $E_{i,j+1}$	8 st $E_{j+1,i}$	9 ld $E_{i,j+2}$
MRU	B_a	B_e	B_a	B_e	B_a	B_f	B_a	B_f	B_a
1		B_a	B_e	B_a	B_e	B_a	B_f	B_a	B_f
2						B_e	B_e	B_e	B_e
3									
4									
LRU									
	10 ld $E_{j+2,i}$	11 st $E_{i,j+2}$	12 st $E_{j+2,i}$	13 ld $E_{i,j+3}$	14 ld $E_{j+3,i}$	15 st $E_{i,j+3}$	16 st $E_{j+3,i}$	17 ld $E_{i+1,j}$	18 ld $E_{j,i+1}$
MRU	B_g	B_a	B_g	B_a	B_h	B_a	B_h	B_b	B_e
1	B_a	B_g	B_a	B_g	B_a	B_h	B_a	B_h	B_b
2	B_f	B_f	B_f	B_f	B_g	B_g	B_g	B_a	B_h
3	B_e	B_e	B_e	B_e	B_f	B_f	B_f	B_g	B_a
4					B_e	B_e	B_e	B_f	B_g
LRU								B_e	B_f
	19 st $E_{i+1,j}$	20 st $E_{j,i+1}$	21 ld $E_{i+1,j+1}$	22 ld $E_{j+1,i+1}$	23 st $E_{i+1,j+1}$	24 st $E_{j+1,i+1}$	25 ld $E_{i+1,j+2}$	26 ld $E_{j+2,i+1}$	27 st $E_{i+1,j+2}$
MRU	B_b	B_e	B_b	B_f	B_b	B_f	B_b	B_g	B_b
1	B_e	B_b	B_e	B_b	B_f	B_b	B_f	B_b	B_g
2	B_h	B_h	B_h	B_e	B_e	B_e	B_e	B_f	B_f
3	B_a	B_a	B_a	B_h	B_h	B_h	B_h	B_e	B_e
4	B_g	B_g	B_g	B_a	B_a	B_a	B_a	B_h	B_h
LRU	B_f	B_f	B_f	B_g	B_g	B_g	B_g	B_a	B_a
	28 st $E_{j+2,i+1}$	29 ld $E_{i+1,j+3}$	30 ld $E_{j+3,i+1}$	31 st $E_{i+1,j+3}$	32 st $E_{j+3,i+1}$	33 ld $E_{i+2,j}$			
MRU	B_g	B_b	B_h	B_b	B_h	B_c			
1	B_b	B_g	B_b	B_h	B_b	B_h			
2	B_f	B_f	B_g	B_g	B_g	B_b			
3	B_e	B_e	B_f	B_f	B_f	B_g			
4	B_h	B_h	B_e	B_e	B_e	B_f			
LRU	B_a	B_a	B_a	B_a	B_a	B_e			
evicted						B_a			

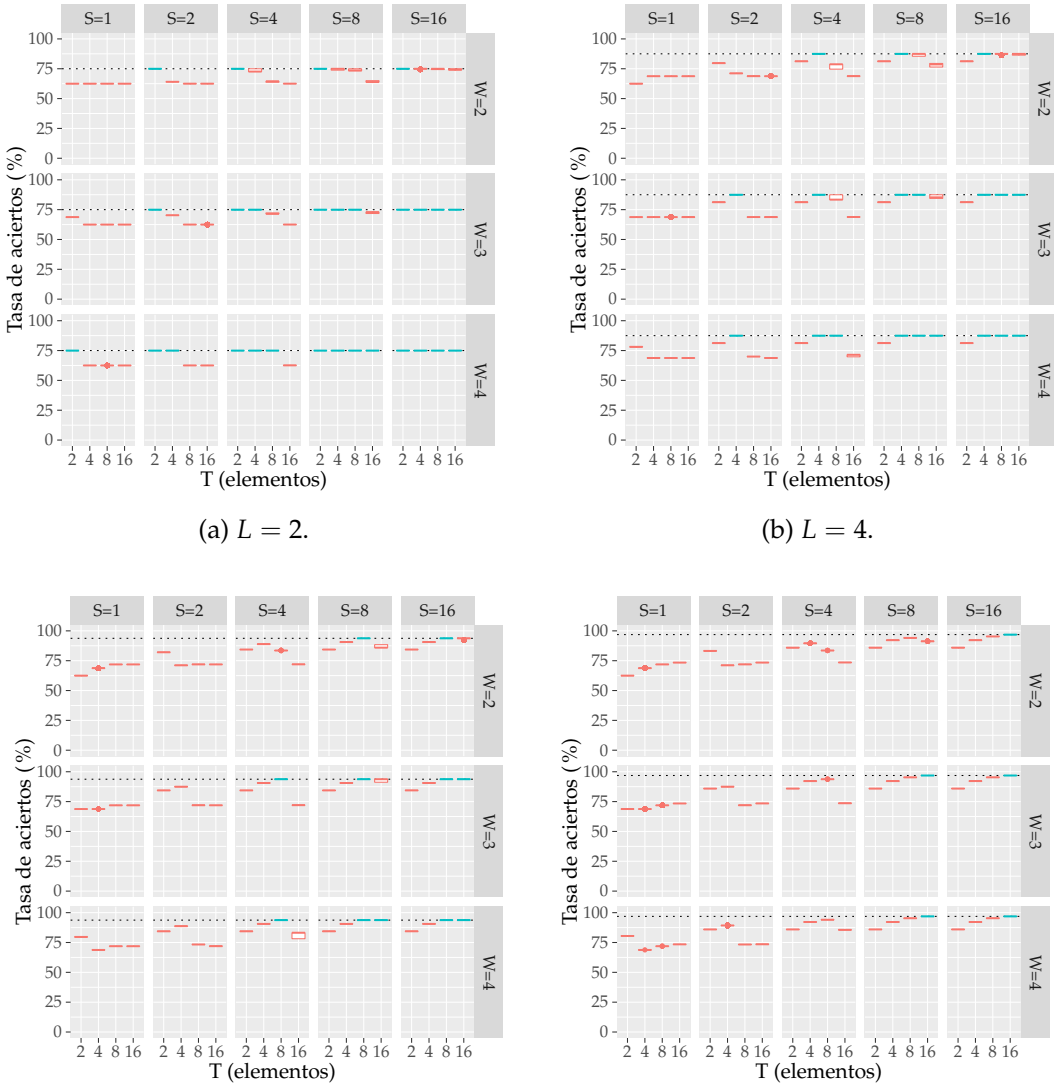
datos en configuraciones de cache diferentes para un amplio tamaño de matrices. Además, los resultados obtenidos se comparan con aquellos proporcionados por el algoritmo *cache-oblivious* demostrando que si el tamaño del tile $T \times T$ se establece como igual al tamaño de la línea de cache ($T = L$), el algoritmo *tiling* siempre va a tener un rendimiento (en términos de tasa de aciertos en datos) mejor o igual que el de *cache-oblivious*. También, llevamos a cabo múltiples experimentos en hardware real para estudiar los tiempos de ejecución (y no solo las tasas de aciertos en cache de datos). Para finalizar esta sección, utilizamos los resultados obtenidos para conseguir un modelo del consumo de energía de la jerarquía de cache para cada uno de los sistemas estudiados. Esto nos permite evaluar el gasto energético debido al comportamiento de la actividad de reemplazo en la jerarquía de memoria.

5.7.1. Tasa de aciertos en cache de datos en el algoritmo *tiling* de transposición de matrices

En esta sección verificamos si la tasa ideal de aciertos (sección 5.4) se puede alcanzar para el algoritmo *tiling* de transposición de matrices. Para ello, suponemos una cache de datos LRU y que a la matriz a trasponer se le ha aplicado la técnica de *padding* tal y como se describe en la sección 5.6. La figura 5.8 muestra varias gráficas que corresponden a diferentes tamaños de línea de cache L (en número de elementos). Todas ellas muestran un boxplot por cada combinación de número de conjuntos (S), vías (W) y tamaños de tile (T). Cada boxplot se prolonga verticalmente y representa la tasa de aciertos en datos para matrices de tamaño desde 1024×1024 hasta 2048×2048 elementos, es decir, 1025 experimentos. En la mayoría de los casos solo aparece la mediana de los datos, como una línea horizontal, debido a que todas las tasas de aciertos de todas las matrices representadas son iguales o sus diferencias son inapreciables. La línea horizontal y punteada representa la asíntota de la tasa de aciertos $1 - \frac{1}{2L}$ (ecuación 5.1). El color del boxplot indica si se ha alcanzado la tasa ideal de aciertos (sección 5.4) (azul), o si no (rojo). Como podemos ver, la tasa ideal de aciertos se logra cuando el eje x (T) coincide con el tamaño de línea en las columnas ($T = L$), dependiendo del número de conjuntos S (ecuaciones 5.7–5.9). Cuando se alcanzan resultados ideales con un tamaño de tile subóptimo ($T > L$, ecuación 5.10), con los mismos parámetros, un tamaño óptimo de tile ($T = L$) también lo logra. Además las propiedades de la política LRU garantizan que cuando la tasa ideal de aciertos se alcanza con cierta configuración de cache, se sigue alcanzando al aumentar el número de conjuntos y/o vías de esta configuración. Encontramos que la tasa ideal de aciertos puede no alcanzarse solo si la cache es demasiado pequeña o si sus parámetros no están correctamente ajustados. Afortunadamente, estos casos no son realistas, y la caches de datos actuales son mucho más grandes que las representadas en la figura 5.8. De hecho, la cache más grande que hemos probado ($W = 4$, $S = 16$) con tamaño de línea de 64 bytes tiene un tamaño de solo 4 KiB.

5.7.2. Tasa de aciertos en datos: *Tiling vs. Oblivious*

Los resultados anteriores nos muestran que, con el tamaño apropiado de tile, la tasa ideal de aciertos se logra con caches muy pequeñas. En vez de un algoritmo *tiling*, se podría usar un algoritmo *cache-oblivious* [Fri+99; Fri+12]. En esta sección evaluamos los requisitos de la cache de datos para lograr la tasa ideal de aciertos con un algoritmo *cache-oblivious*, tras lo cual lo compararemos con nuestros resultados con *tiling*. Un algoritmo *cache-oblivious* se puede ver como un algoritmo *tiling* en el cual la matriz a procesar se divide recursivamente hasta llegar, en la transposición de matrices, a un tamaño de tile de 2×2 elementos. Por lo tanto, es probable que el procesado de cada tile pueda caber en cache. El orden en el que se lleva a cabo la transposición viene dado por la recursión. Este orden es una variación de *Z-order* el cual conserva la localidad [Mor66] (ver figura 5.1(b)). Así, no es necesario conocer la configuración de cache para aplicar este algoritmo. Sin embargo, su rendimiento depende de los parámetros de cache y predecir cuando será bueno o malo no es trivial [TRS04]. Debido a que el algoritmo *oblivious* divide recursivamente la matriz a transponer surgen problemas cuando la dimensión de esta matriz (N) no es potencia de 2. Podemos ver este efecto en la figura 5.9 donde solo alcanzan la tasa ideal de aciertos aquellas matrices cuya dimensión es potencia de dos. Proponemos aplicar un *phantom padding* («Relleno fantasma») que permite resolver este inconveniente.



(a) $L = 2$.

(b) $L = 4$.

(c) $L = 8$.

(d) $L = 16$.

FIGURA 5.8: Las gráficas (**a**–**d**) muestran las tasas de aciertos en datos variando el número de elementos por línea ($L = 2, 4, 8, 16$ elementos) para diferentes configuraciones de cache con S conjuntos, W vías, y tamaño de tile $T \times T$ elementos. Cada boxplot reúne los resultados de todas las matrices de 1024×1024 a 2048×2048 elementos. El color de cada boxplot muestra si se alcanza siempre la tasa ideal de aciertos (azul) o no (rojo).

Su aplicación no modifica el mapeo en memoria de la matriz a transponer pero fuerza al algoritmo *oblivious* a trabajar como si la dimensión de la matriz fuera potencia de 2. A continuación, solo se intercambian los elementos que son reales, en otras palabras, el intercambio de elementos no se da en los elementos «fantasma» que se añaden debido a la aplicación de este *padding*. Esta transformación tampoco afecta al número de accesos a datos, simplemente se asegura que la secuencia de accesos en el algoritmo *oblivious* sea regular. Hasta donde sabemos esta técnica de padding es una propuesta original de este trabajo.

Los algoritmos 4 y 5 muestran los algoritmos de transposición de matrices *cache-oblivious* basados en la propuesta original [Fri+99; Fri+12]. Básicamente, estos algoritmos asumen que el parámetro N es potencia de 2 y la matriz que realmente se

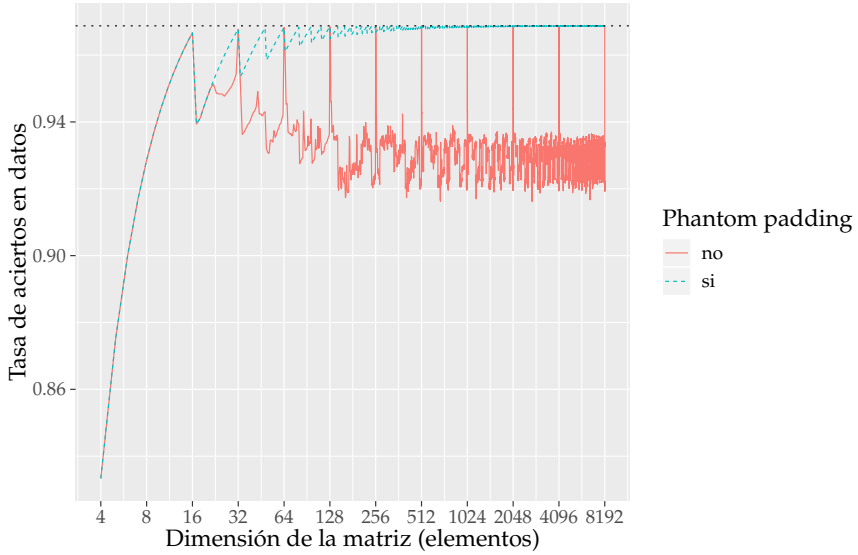


FIGURA 5.9: Tasas de aciertos en cache de datos con y sin *phantom padding* (algoritmos 4 y 5) para la transposición de matrices *cache-oblivious*. La configuración de la cache de datos es: $L = 16, S = 16$ y $W = 2$. Los tamaños de matriz varían de 4×4 a 8192×8192 elementos. La tasa ideal de aciertos en datos (ecuaciones 5.2–5.4) coincide con la versión *phantom padding*. La línea punteada negra muestra la cota superior de tasa de aciertos de la ecuación 5.1.

debe transponer se especifica a través de los parámetros de índice. De esta manera, solo se lleva a cabo el intercambio de elementos cuando el índice correspondiente es menor que N. Podemos encontrar esta condición que hemos introducido en la línea 6 en el algoritmo 4 y en la línea 8 en el algoritmo 5.

Algoritmo 4 Transpuesta($MatrizConPadding, N, indice1 = 1, indice2 = N$): transpone recursivamente una matriz con *phantom padding*, almacenada por filas.

```
1:  if indice2 − indice1 ≤ 2 then
2:      MatrizConPadding_{indice1,indice1+1} ↔ MatrizConPadding_{indice1+1,indice1}
3:  else
4:      indicemitad ← (indice1 + indice2)/2;
5:      Transpuesta(MatrizConPadding, N, indice1, indicemitad);
6:      if indicemitad < N then
7:          Transpuesta(MatrizConPadding, N, indicemitad, indice2);
8:          TranspuestaIntercambio(MatrizConPadding, N, indicemitad, indice1, indice2, indicemitad);
9:      end if
10: end if
```

Algoritmo 5 TranspuestaIntercambio($MatrizConPadding$, N, rs, cs, re, ce).

```
1: if ((re − rs) ≤ 2 and (ce − cs) ≤ 2) then
2:   for indice1 ← rs to re − 1 do
3:     for indice2 ← cs to ce − 1 do
4:       MatrizConPadding_{indice1,indice2} ↔ MatrizConPadding_{indice2,indice1}
5:     end for
6:   end for
7: else
8:   if rs < N then
9:     rmitad ← (rs + re)/2;
10:    cmitad ← (cs + ce)/2;
11:    TranspuestaIntercambio(MatrizConPadding, N, rs, cs, rmitad, cmitad);
12:    TranspuestaIntercambio(MatrizConPadding, N, rmitad, cs, re, cmitad);
13:    TranspuestaIntercambio(MatrizConPadding, N, rs, cmitad, rmitad, ce);
14:    TranspuestaIntercambio(MatrizConPadding, N, rmitad, cmitad, re, ce);
15:  end if
16: end if
```

A continuación, repetimos los experimentos de la sección 5.7.1 para este algoritmo *oblivious* mejorado. La figura 5.10 recapitula nuestros resultados para *tiling* (figura 5.8), y los correspondientes a *oblivious*. Esta gráfica muestra el mínimo número de vías de cache W necesarias para lograr la tasa ideal de aciertos para cada combinación de tamaño de línea de cache L y número de conjuntos de cache S, es decir, cuanto más pequeña mejor. Encontramos los resultados de *tiling* a la izquierda y los de *oblivious* a la derecha. Solo se muestran tamaños adecuados de tile. Las configuraciones inadecuadas de *tiling* ($T \neq L$) tienen peores resultados y no aparecen en esta gráfica. La barras verdes señalan el mínimo número de conjuntos necesarios para alcanzar la tasa ideal de aciertos. Se puede ver que, con un número suficiente de conjuntos ($S \geq L$), *oblivious* necesita el mismo número de vías en cache que *tiling*. Sin embargo, en caches con un menor número de conjuntos (incluidas las totalmente asociativas) *oblivious* necesita más vías que *tiling* para lograr la tasa ideal de aciertos. Esto se debe a que *oblivious* no se detiene cuando la recursión alcanza el tamaño de tile más apropiado sino que continúa reduciendolo hasta llegar a tiles de tamaño 2×2 elementos. Así, cuanto mayor es el tamaño de línea, más contenido innecesario se trae de memoria. Este contenido (que se necesitará para los tiles posteriores) debe mantenerse en cache para poder garantizar la tasa ideal de aciertos en datos. De modo que si fijamos el número de conjuntos, será necesario incrementar el número de vías.

Las caches de datos tienen habitualmente un número relativamente grande de conjuntos($S \geq L$), así que en general en la transposición de matrices, tanto *tiling* como *cache-oblivious*, se alcanza la tasa ideal de aciertos. Aun así hay que tener en cuenta que las caches en los procesadores actuales pueden ser compartidas por diferentes hilos hardware al mismo tiempo, de modo que sigue siendo importante acceder a ellas de la manera más eficiente posible. Los resultados anteriores muestran que *tiling* alcanza la tasa ideal de aciertos con menos recursos que *oblivious*. Además es importante señalar que los códigos recursivos, como el del algoritmo *oblivious*, suelen ser más lentos que los iterativos debido al sobrecoste de las llamadas a funciones y del procesamiento de la pila. También hay que tener en cuenta que para sistemas de tiempo real, con un algoritmo recursivo, tendríamos que proporcionar como información en el análisis el máximo nivel de recursión y comprobar que no

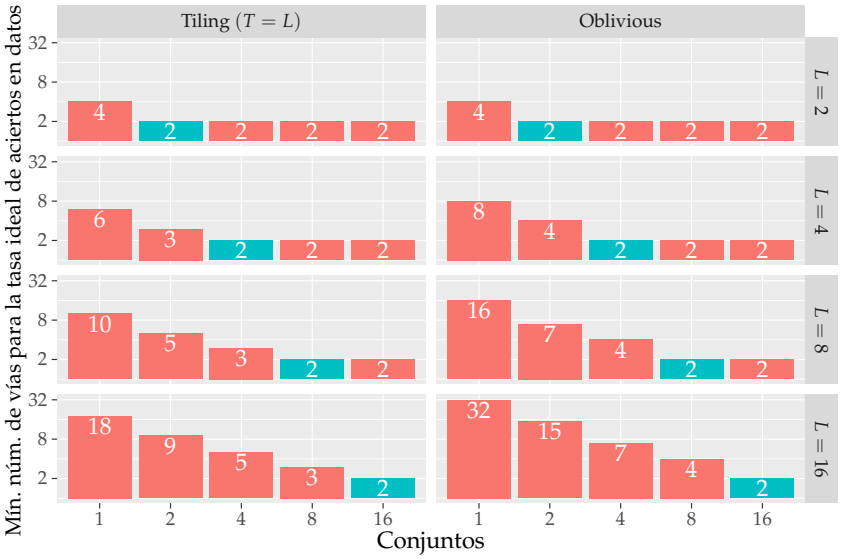


FIGURA 5.10: Mínimo número de vías (W) necesarias para alcanzar la tasa ideal de aciertos, según el número de conjuntos S y tamaño de línea L de la cache, para la transposición de matrices con el algoritmo *tiling* con tamaño de tiles $T \times T$ ($T = L$) y para el algoritmo *oblivious* con *phantom padding* (cuánto más pequeño mejor).

aparecen problemas en la pila debido a ello.

5.7.3. PLRU y tiempo de ejecución en plataformas reales

Los resultados anteriores cuentan con una cache asociativa por conjuntos LRU. Debido a que cuesta bastante construir una política LRU que sea eficiente para una gran cache, la mayoría de caches en los procesadores comerciales usan una política PLRU la cual ofrece un comportamiento parecido pero es más simple de implementar [AR13]. En esta sección primero comparamos la tasa de aciertos en datos LRU con la de PLRU por medio de simulaciones. La figura 5.11 muestra las tasas de aciertos en datos para la transposición de matrices de 4096×4096 elementos de 8 B con una configuración común de cache de datos L1 (64 conjuntos, 8 vías, y 64 bytes por línea, siendo su tamaño total de 32 KiB). Con este tamaño de línea, cada una contiene 8 elementos de la matriz. Los experimentos varían la dimensión del tile de $T = 2$ a $T = 512$. El área sombreada muestra los tamaños de tile que alcanzan la tasa ideal de aciertos en datos con LRU para tiles cuya dimensión (T) es múltiplo del tamaño de línea ($L = 8$). Esto se puede ver claramente entre 8 y 16 de tamaño de tile donde decrece la tasa de aciertos. El tile 8×8 corresponde a $T = L$ (tasa ideal de aciertos con los mínimos recursos de cache). Al incrementar el tamaño del tile a partir de 8×8 hasta 256×256 elementos se alcanza la tasa ideal de aciertos pero utilizando más conjuntos/vías (ecuación 5.10). Se puede ver que tanto LRU como PLRU tienen prácticamente las mismas tasas de acierto. Tal como se espera la tasa de aciertos disminuye fuera del área sombreada tanto en LRU como en PLRU. Las únicas diferencias que se pueden apreciar realmente, aunque mínimas, se encuentran en torno a $T = 256$, es decir, la cota de la ecuación 5.10.

Gracias a las conclusiones obtenidas en las secciones previas de este capítulo podemos garantizar la predictibilidad en LRU, tan importante en ciertas áreas como en la que se desarrolla nuestro trabajo, los sistemas de tiempo real. Sin embargo, no puede garantizarse para PLRU que se deberá evitar en estos sistemas, como ya han

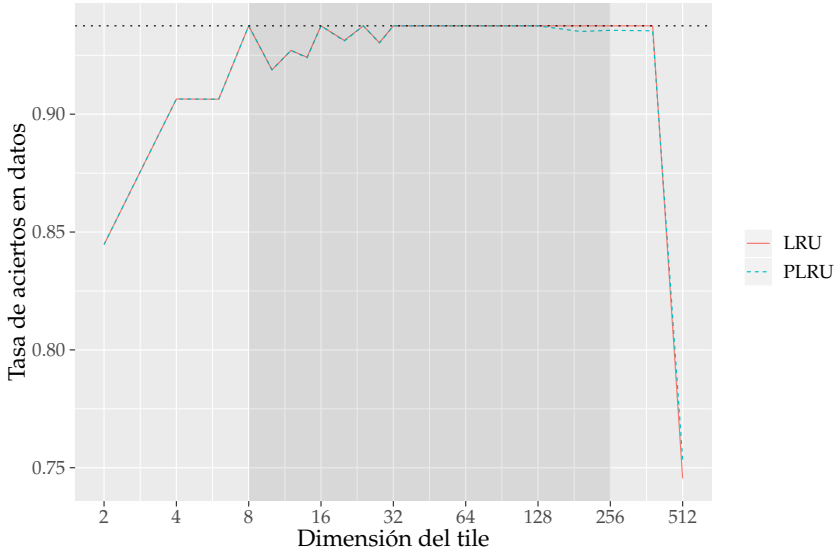


FIGURA 5.11: Tasa de aciertos en datos para la transposición de matrices de 4096 × 4096 elementos (cuanto mayor, mejor) para política de reemplazo LRU y PLRU en una configuración de cache de datos L1 común (64 conjuntos, 8 vías, y 64 bytes por línea, con un tamaño total de 32 KiB). La línea negra punteada muestra la cota superior de tasa de aciertos de la ecuación 5.1.

indicado otros estudios [Ber06; Rei+07; AR14]. Aunque el factor más determinante para el rendimiento en estas simulaciones sea probablemente la tasa de aciertos en datos, hay también otros elementos importantes que pueden afectar a este rendimiento como el número de instrucciones ejecutadas (dependiente del tamaño del tile). Para poder tener en cuenta estos y otros factores (p. ej. prebuscadores hardware de datos) medimos los tiempos de ejecución. Estos experimentos se han llevado a cabo en varias máquinas: Intel-Xeon-L5410 2.33 GHz, Intel-i7-4810MQ 2.80 GHz, Intel-Core-2-Quad-Q9550 2.83 GHz,y Intel-i7-2640M 2.80 GHz. Todas ellas tienen una cache L1 de instrucciones y otra de datos, ambas con reemplazo PLRU, 64 conjuntos, 8 vías, y 64 bytes por línea, con un tamaño total de 32 KiB, tal como la que simulamos en la figura 5.11.

Se ha codificado el algoritmo 3 en C (código fuente disponible online[1]) y, para cada máquina, se ha generado un binario distinto usando el compilador disponible (GCC-4.7.0 for i7-4810M and Core-2-Quad-Q9550, Intel C++ Composer XE 2013 for Xeon-L5410, y GCC-4.9.2 for i7-2640), pero siempre con optimización de nivel 3.

La figura 5.12 muestra el menor tiempo de ejecución para la transposición de matrices de 4096 × 4096 elementos, variando el número de elementos por tile. Cada uno de los tiempos representados es el menor de 400 repeticiones, representando la ejecución más rápida, es decir la que estaría más cerca de una transposición de matrices aislada de interferencias externas. El área sombreada indica las configuraciones de tile que alcanzan la tasa ideal de aciertos con una cache LRU para tiles múltiplos del tamaño de línea, como antes. El tiempo debería incrementarse para LRU fuera del área sombreada y de hecho se puede ver que este es el comportamiento que presenta PLRU en la gráfica en la plataforma 2QuadQ9550. Sin embargo, en las otras tres plataformas el tiempo de ejecución crece para grandes tiles aunque muy ligeramente. Esto se debe al prebuscador en datos, el cual reconoce los patrones de acceso a

[1] webdiis.unizar.es/gaz/repositories/tiling-matrix-transposition

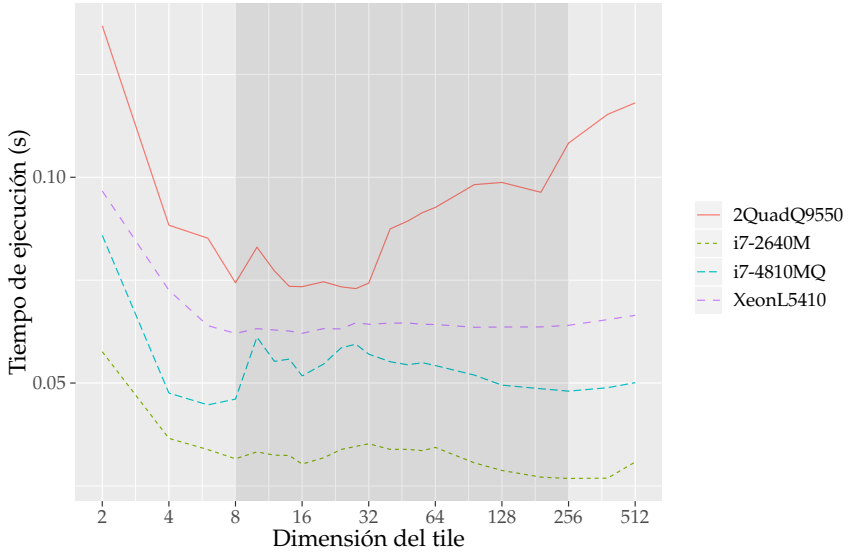
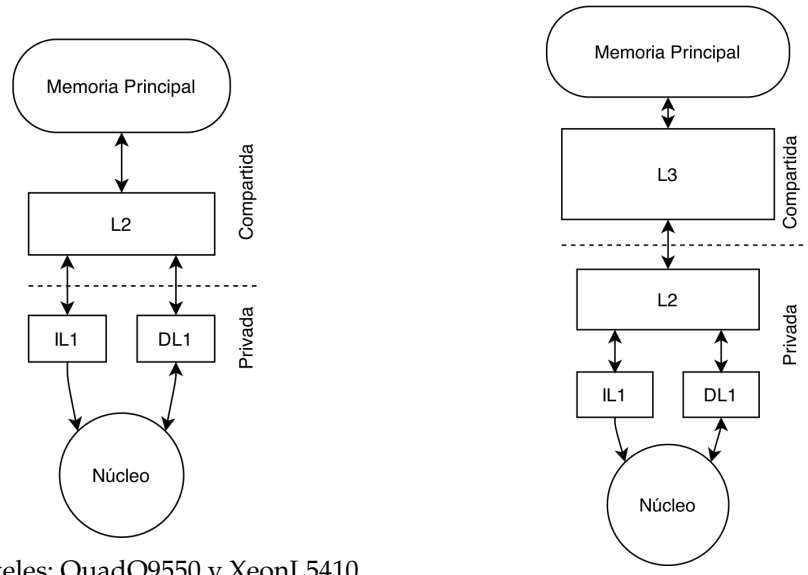


FIGURA 5.12: Tiempos de ejecución para la transposición de matrices de 4096×4096 elementos. Además de los tamaños de tile en el eje-x, se puede ver los tiempos de ejecución para otros valores de T: 6, 10, 12, 14, 20, 24, 28, 40, 48, 56, 96, 192, y 384.

datos y evita las penalizaciones en tiempo. En el área de la izquierda los tiles son demasiado pequeños para que el prebuscador reconozca ningún patrón de acceso, por ello el tiempo de ejecución para estos tiles es bastante mayor. Los tamaños que son múltiplos de 8 trabajan con líneas completas de cache y por lo tanto, en general, obtiene un rendimiento mejor que aquellos a su alrededor. Se puede ver claramente que entre 8 y 16 todas las gráficas muestran un incremento del tiempo de ejecución. Por último, cuanto mayor es el tamaño del tile, menos instrucciones se ejecutan. Para cada tile, se tienen que calcular las cotas de sus índices y también asegurar que en los tiles de los extremos de la matriz no se intercambian datos de fuera de ella. Esto es, cuanto menores son los tamaños de tile dentro del área sombreada más eficiente es el uso que hacen de la cache, pero tienen más instrucciones. Por tanto, se produce una compensación entre los recursos de cache y las instrucciones ejecutadas y la tendencia en cuanto al tiempo de ejecución diferirá dependiendo de la máquina en la que se ejecute la transposición.

5.7.4. Modelado y reducción del consumo de energía en el subsistema de memoria para la transposición de matriz versión *tiling*

En esta sección analizamos el consumo de energía en los subsistemas de memoria de las cuatro plataformas de la sección 5.7.3. Para poder hacerlo usamos la tasa analítica de aciertos introducida en la sección 5.6 y la combinamos con las gráficas de energía obtenidas por la herramienta de modelización CACTI 7.0 [Bal+17]. Esta herramienta permite calcular la energía por acceso y la energía estática dada un configuración de cache o de memoria principal. Primero, describimos los subsistemas de memoria de las cuatro plataformas, presentamos las gráficas de potencia y energía de cada componente y explicamos cómo calcular la energía global. Después, nos centramos en un tamaño específico de matriz tomando varios tamaños de tile con lo que obtenemos el consumo de energía global y tras lo cual sugerimos mejoras en el software/hardware.



(a) 2 niveles: QuadQ9550 y XeonL5410.

(b) 3 niveles: i7-2640 y i7-4810.

FIGURA 5.13: Organización de las jerarquías de memoria.

Modelado analítico de consumo de energía

La figura 5.13 muestra las dos jerarquías que encontramos en las cuatro plataformas de nuestra experimentación. Xeon-L5410 y 2QuadQ9550 tienen dos niveles de cache, y i7-2640M y i7-4810MQ tienen tres. El último nivel de cache, el más cercano a la memoria principal, está compartido entre los núcleos (solo se muestra uno en la figura 5.13), mientras que los niveles más bajos son privados para cada uno de ellos. Todas las caches de datos son caches *copy-back*, lo que quiere decir que el efecto de una sola escritura no se ve en niveles superiores hasta que no se expulsa por completo la línea modificada. Las operaciones de lectura y escritura que se ejecutan en el núcleo acceden primero a la cache L1. Cuando se da un fallo, se busca en el siguiente nivel de cache hasta que se produce un acierto o se llega a memoria principal. Un fallo en el último nivel trae la línea de memoria principal al menos a cache L1 y, dependiendo de la política de control de contenido entre niveles, también a las caches L2 y L3. Por simplicidad, ya que las cuatro plataformas tienen diferentes políticas, asumimos que todas tienen una política inclusiva en la cual se ordena que se copien las líneas que vengan de memoria principal a todos los niveles de cache, es decir, L1 $\subset$ L2 $\subset$ L3.

La tabla 5.2 muestra los parámetros de cache y tecnológicos para todas las plataformas. Las gráficas de energía y potencia han sido calculadas por CACTI, considerando la tecnología del nodo, el nivel de memoria, y los diferentes tamaños y asociatividades. Solo se muestra la energía para leer/escribir un elemento de 8 bytes en las caches L1, cuando las instrucciones de lectura y escritura operan a esta granularidad, mientras que la energía para leer/escribir objetos de 64 bytes permiten considerar las transferencias de líneas entre todos los niveles de memoria. Elegimos una memoria principal de 1 GiB porque es lo suficientemente grande para almacenar 128 MiB, el tamaño de la matriz de 4096×4096 elementos de 8 bytes que se usa en la figura 5.12 y en la siguiente sección. Ahora podemos usar el modelo de la tasa de aciertos de la sección 5.6 para contar todos los eventos de interés en cualquier nivel de la jerarquía inclusiva expuesta arriba: escrituras, lecturas, aciertos y reemplazos. Entonces podemos calcular la energía dinámica total multiplicando el número de

TABLA 5.2: Parámetros del subsistema de memoria para las cuatro plataformas.

	2QuadQ9550	**i7-2640M**	**i7-4810MQ**	**XeonL5410**
Número de niveles de cache	2	3	3	2
Tamaño de línea	64 B	64 B	64 B	64 B
Tecnología del nodo	45 nm	32 nm	22 nm	45 nm
Tamaño L1	32 KiB	32 KiB	32 KiB	32 KiB
Asociatividad L1	8 vías	8 vías	8 vías	8 vías
Energía de lectura L1 (pJ) 8 B	19.3	13.3	6.9	19.3
Energía de lectura L1 (pJ) 64 B	665.3	464.8	327.5	66.5
Energía de escritura L1 (pJ) 8 B	24.2	16.3	8.2	24.2
Energía de escritura L1 (pJ) 64 B	670.3	468.2	333.0	67.0
Potencia de fuga L1 (mW)	65.2	45.2	20.4	65.2
Tamaño L2	6 MiB	256 KiB	256 KiB	12 MiB
Asociatividad L2	24 vías	8 vías	8 vías	24 vías
Energía de lectura L2 (nJ) 64 B	3.7	0.8	0.4	23.0
Energía de escritura L2 (nJ) 64 B	4.0	0.8	0.5	25.1
Potencia de fuga L2 (W)	10.6	0.3	0.1	16.2
Política de inclusión L2 simulada (real)	Inclusiva	Inclusiva (no-inclusiva)	Inclusiva (no-inclusiva)	Inclusiva
Tamaño L3	N/A	4 MiB	6 MiB	N/A
Asociatividad L3	N/A	16 vías	32 vías	N/A
Energía de lectura L3 (nJ) 64 B	N/A	2.3	1.5	N/A
Energías de escritura L3 (nJ) 64 B	N/A	2.4	1.6	N/A
Potencia de fuga L3 (W)	N/A	2.4	1.7	N/A
Política de inclusión L3	N/A	Inclusiva	Inclusiva	N/A
Tamaño memoria principal (DDR3)	1 GiB	1 GiB	1 GiB	1 GiB
Energía de lectura MP (nJ) 64 B	17.7	17.7	17.7	17.7
Energía de escritura MP (nJ) 64 B	17.7	17.7	17.7	17.7
Potencia de fuga MP (W)	2.2	2.2	2.2	2.2

eventos (lecturas, escrituras, reemplazos, etc.) por la energía correspondiente a cada evento indicada en la tabla 5.2. Para calcular la energía estática debemos integrar toda potencia de fuga durante el tiempo de ejecución que hemos medido en cada plataforma para cada tamaño de tile (figura 5.12).

Estimación de consumo de energía y posibles mejoras

La figura 5.14 muestra las estimaciones de consumo de energía para las cuatro plataformas variando el tamaño de tile en múltiplos del tamaño de línea para poder alcanzar siempre la tasa ideal de aciertos. Lo primero que tenemos que tener en cuenta es que las tecnologías más antiguas (45 nm) consumen más energía debido principalmente al gran consumo de energía estática. Por este motivo, en 45 nm, sobre un 80 % del consumo de energía es estático mientras que en 32 y 22 nm este porcentaje se reduce a en torno el 50 %. Para todas las ejecuciones llevadas a cabo, el consumo de energía está fuertemente correlacionado con la ejecución del algoritmo de transposición de matrices. Por tanto, mientras el tamaño de tile sea óptimo, el consumo permanece constante para todas las plataformas excepto para Q9550, donde el incremento del tiempo de ejecución desencadena el consumo de energía estática cuando se incrementa el tamaño del tile.

La transposición de matrices es una operación de álgebra lineal que se usa normalmente como parte de una secuencia que resuelve una problema mayor, como se

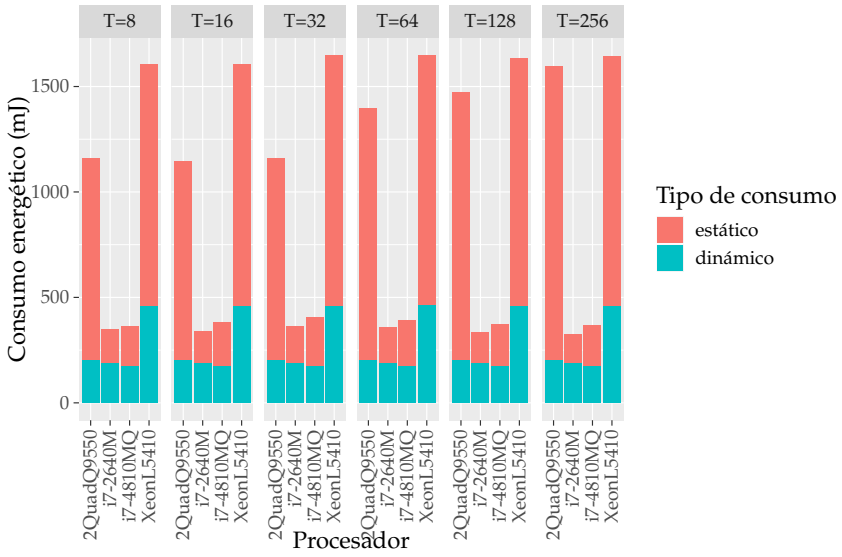


FIGURA 5.14: Energía consumida por la jerarquía de memoria de los sistemas experimentales para distintos tamaños de tile.

presenta en diversos estudios [RJ16; Bal+16]. En el marco de la supercomputación, el tamaño de los datos de entrada es lo suficientemente grande para que no quepa en ninguno de los niveles de la jerarquía de cache. En nuestro caso, una matriz sencilla de 4096×4096 elementos (8 bytes cada uno) ocupa 128 MiB. Además, el algoritmo de transposición *tiling* asegura que cada elemento que se visita nunca vuelve a visitarse. Esto también ocurre en otros algoritmos *tiling* de algebra lineal como la multiplicación de vector por matriz. Aunque los elementos en la transposición de matrices *tiling* solo se usan de manera efectiva en L1, las políticas de inclusión y reemplazo guardan copias de los elementos que ya han sido usados en los diferentes niveles de cache. Gracias a nuestro modelo analítico de energía podemos anticipar cuanta energía se puede ahorrar si somos capaces de eludir los niveles intermedios de la jerarquía de cache cuando sabemos que su trabajo no va a ser necesario. Así que proponemos dos posibles enfoques para ahorrar energía. En el primero, garantizamos que los fallos en cache L1 solo se traen a L1, y que los reemplazos en L1 son expulsados directamente a memoria principal. En este enfoque se evita la actividad asociada con los otros niveles de cache pero no el consumo de su energía estática, ya que queremos que continúen dando servicio a otras aplicaciones que puedan estar ejecutándose en el sistema. Como segundo enfoque, además de la estrategia evitando los niveles intermedios, proponemos poner todos los niveles de cache excepto L1 en un estado de «hibernación». En este estado, se minimiza el voltaje de alimentación lo cual reduce de manera significativa el consumo estático, pero sin perder el contenido almacenado como sucedería si desconectáramos totalmente la alimentación. La desventaja de este estado es que no se puede reanudar el funcionamiento normal de la cache antes de reestablecer el voltaje nominal. Por ejemplo, en 32 nm podemos reducir al 56 % el consumo de energía estática [FLS13; Fla+02]. La tabla 5.3 muestra los porcentajes de ahorro de energía en la jerarquía de memoria para cada uno de los enfoques presentados previamente en relación con los resultados de la figura 5.14. Estos porcentajes varían bastante entre plataformas y van desde un 3 % a un 46 %. Como se esperaba, la segunda propuesta gana en todas las plataformas, especialmente en Q9550 y Xeon L5410 ya que es donde el consumo estático se impone. Por otro lado, las plataformas i7-2640 y i7-4810 manifiestan una reducción de consumo

TABLA 5.3: Porcentaje de ahorro de consumo energético en la jerarquía de memoria para los dos enfoques presentados.

	2QuadQ9550						i7-2640M					
	8	16	32	64	128	256	8	16	32	64	128	256
Enfoque 1	4.2	4.3	4.2	3.5	3.3	3.1	10.7	10.9	10.2	10.3	11.1	11.5
Enfoque 2	34.2	34.1	34.2	34.5	34.6	34.8	21.7	21.6	21.8	21.8	21.6	21.5
	i7-4810MQ						XeonL5410					
	8	16	32	64	128	256	8	16	32	64	128	256
Enfoque 1	6.5	6.1	5.8	6.0	6.3	6.4	19.1	19.1	18.6	18.6	18.8	18.7
Enfoque 2	16.8	17.0	17.1	17.1	16.9	16.9	46.6	46.6	46.5	46.5	46.5	46.5

más contenida. Esto se debe a que sus jerarquías de cache no son tan grandes como en Xeon L5410 (12 MiB), y también a la mejora tecnológica. En el marco de la supercomputación, donde el procesador ejecuta el algoritmo de transposición de matrices intensivamente/exclusivamente, la disminución del consumo de energía se multiplicará por los miles de nodos que ejecuten este algoritmo simultáneamente. Debido al tamaño de los datos que maneja el algoritmo de transposición de matrices y el hecho de que todo el reuso de este algoritmo se explota en L1, los niveles intermedios de cache no ayudan a mejorar el rendimiento. Y además esto causa un notable sobrecoste de consumo de energía. La industria parece haber notado que hay algoritmos que no presentan un reuso convencional y causan sobrecostes en el subsistema de memoria. Por ello, desde la extensión de instrucciones SSE 4.1, Intel ofrece instrucciones de anotación (*Non-Temporal Aligned Hint*) que permiten acceder a memoria sin contaminar la jerarquía de cache. Lamentablemente, estas instrucciones ni siquiera cargan el dato accedido en L1, con lo cual en el caso de los algoritmos *tiling* parece que no serán útiles para este propósito. Otro ejemplo es la instrucción CLDEMOTE que estará próximamente disponible en la futuras microarquitecturas «Tremont» y sus posteriores desarrollos. La instrucción CLDEMOTE es una anotación en el hardware que quizá pueda ayudar a mover una línea de cache de los niveles, o nivel, más cercanos al núcleo a otro que esté más lejos. Sin embargo, lo que necesitaríamos sería una variación de CLDEMOTE que envíe directamente a memoria principal la línea expulsada de L1.

5.8. Conclusiones

En este capítulo estudiamos la tasa de aciertos para el problema de la transposición de matrices dado el impacto de esta en el WCET y la importancia de este algoritmo en los sistemas de tiempo real. Primero obtenemos una cota superior de la tasa ideal de aciertos independiente del algoritmo de transposición y de la configuración de cache calculando el número mínimo de fallos obligatorios. Después, analizamos desde una perspectiva teórica la relación entre los parámetros de una cache LRU y el tamaño del tile para alcanzar la tasa ideal de aciertos considerando varias opciones de *padding*. Nuestros resultados teóricos muestran que, con una correcta configuración de tile (la dimensión del tile igual al tamaño de línea de cache), se logra la tasa ideal de aciertos con muy pocos conjuntos y no más de dos vías en una cache asociativa por conjuntos. Estos resultados se confirman por medio de simulaciones extensivas que también validan los requisitos adicionales de cache cuando no se configura

el tamaño de tile correctamente. También comparamos nuestros resultados con un algoritmo *cache-oblivious* el cual ofrece muy buen rendimiento independientemente de la configuración de cache. Nuestros resultados muestran que tanto el algoritmo *tiling* (si se configura correctamente) como el *cache-oblivious* necesitan dos vías para alcanzar la tasa ideal de aciertos para una cache con los conjuntos suficientes. Si el número de conjuntos en la cache de datos es demasiado pequeño, el algoritmo *tiling* supera a *cache-oblivious* que necesitará una cache con más vías para alcanzar la tasa ideal de aciertos. Por lo tanto, *tiling* proporciona una tasa de aciertos mejor o igual y al mismo tiempo evita la naturaleza recursiva (que es menos eficiente en general) de los algoritmos *cache-oblivious*.

Por último, para probar el rendimiento general y no solo la tasa de aciertos, estudiamos el tiempo de ejecución de la transposición de matrices en diferentes máquinas. Aunque muchas de las CPUs actuales usan una política PLRU en vez de LRU, nuestros resultados muestran que las tasas de acierto son prácticamente idénticas. Así que las variaciones en su tiempo de ejecución reflejan aspectos tales como la prebúsqueda de datos o el número de instrucciones ejecutadas. También modelamos las jerarquías de memoria de estas máquinas para medir el consumo de energía durante la ejecución de la transposición de matrices. Nuestro análisis muestra que las jerarquías convencionales de memoria no ofrecen instrucciones específicas para una transposición bien configurada y proponemos dos enfoques para eludir los niveles intermedios de cache reduciendo así el consumo de energía sin ningún impacto negativo. De hecho, eludir estos niveles seguramente mejore el rendimiento de otras tareas que se estén ejecutando en el sistema ya que evita la contaminación innecesaria que produciría la transposición de matrices. Las expresiones analíticas para la configuración óptima de la cache de datos para la transposición de matrices que obtenemos en nuestra investigación puede aplicarse de manera directa en nuestra área de estudio, los sistemas de tiempo real, y en otras como la supercomputación.

En el caso de los sistemas de tiempo real, un cálculo preciso del número de aciertos y fallos dependiendo de los parámetros de cache y del tamaño del tile son básicos para calcular el WCET. En el de la supercomputación, los parámetros apropiados en el algoritmo garantizarán un uso óptimo del sistema de supercomputación en términos de rendimiento y eficiencia de los recursos utilizados.

Este trabajo ha sido publicado en [Ped+20a].

Capítulo 6

Conclusiones

En este capítulo resumimos las conclusiones que se han recogido a lo largo de esta tesis.

Los sistemas de tiempo real son cada vez más importantes tanto en la industria como en nuestra vida cotidiana, encontrándose en múltiples sectores. Esta presencia va a continuar incrementándose tal como se muestra en recientes estudios de mercado. Éstos prevén que debido a la llegada de sensores inteligentes, el avance de la interfaz hombre-máquina, y la presencia del Internet de las cosas en los sistemas empotrados, las ganancias para 2026 en el segmento de mercado de los sistemas de tiempo real se verán incrementadas en un 9 % [Ins20].

Un análisis preciso y seguro del WCET es imprescindible para una buena planificación en los sistemas de tiempo real estricto. Además podemos mejorar estos sistemas reduciendo tanto el tiempo de análisis como el WCET en sí mismo. En general, la jerarquía de memoria es el componente con mayor impacto en el WCET, tanto por su continuo funcionamiento como por su latencia variable. Por ello este trabajo se ha centrado en mejorar el aprovechamiento de la jerarquía de memoria en los sistemas de tiempo real en relación al WCET aunque también se han estudiado otros aspectos como el consumo energético.

En el capítulo 3 reducimos el tiempo de análisis del WCET en presencia de una cache de instrucciones bloqueable a través del desarrollo de una extensión, *DLock-MS*, del método de análisis del WCET *Lock-MS*. Esta extensión incorpora dos mejoras. La primera es un algoritmo que transforma el CFG en una estructura en árbol, la cual permite usar *Lock-MS* como un método basado en la estructura y, a partir de esta estructura en árbol, genera un modelo ILP que puede resolverse fácilmente a través de una ramificación y poda invertida. Ya que estas transformaciones se realizan en una sola pasada y se reduce el número de caminos a explorar en la transformación a árbol, conseguimos reducir el tiempo de análisis del WCET sin perder precisión para una cache de instrucciones bloqueable.

La segunda aportación es una heurística de bloqueo dinámico basada en los bucles externos que permite minimizar el WCET en cada región obteniendo su contenido óptimo. Pese a su aparente sencillez esta heurística reduce el WCET explotando de manera efectiva el reuso temporal. Además reduce el tiempo de análisis del WCET (10 veces más rápido que Otawa) aislando el WCET de cada región. Este análisis por regiones continúa permitiendo (como en *Lock-MS*) que en caso de que no se complete el análisis hasta llegar al óptimo (mínimo) de cada región se pueda garantizar que todas las soluciones a su modelo son seguras.

La evaluación de eficiencia de nuestra extensión nos confirma que se reduce el WCET (un 2,2 % de media) respecto al método original *Lock-MS* y que, con una cache bloqueable, se obtiene una tasa de aciertos muy similar a una cache LRU convencional pero con un hardware mucho más sencillo.

Completamos este estudio analizando el impacto de los niveles de optimización en el WCET y concluyendo que es conveniente descartar las compilaciones sin optimización (-O0) ya que el WCET de los binarios generados resulta entre 3 y 4 veces peor que con la presencia de optimizaciones. Sin embargo, hay que evitar optimizaciones demasiado agresivas que puedan cambiar los patrones de bucle dificultando, y empeorando, el análisis del WCET.

El trabajo desarrollado en este capítulo se ha presentado en *A. Pedro-Zapater, J. Segarra, C. Rodríguez, R.G. Tejero y V. Viñals-Yúfera (2016) «Obtención del WCET óptimo en caches de instrucciones bloqueables (Lock-MS) en Otawa» V Simposio de sistemas de tiempo real, 13-16 Septiembre 2016, Salamanca.* y se ha publicado en *A. Pedro-Zapater, J. Segarra, C. Rodríguez, R.G. Tejero y V. Viñals-Yúfera (2020) «Reducing the WCET and analysis time of systems with simple lockable instruction caches» PLOS ONE 15(3): e022998*.

En el capítulo 4 ampliamos el estudio a aquellos programas con código vectorizado analizando su impacto en el WCET. Para ello se estudian las vectorizaciones automáticas generadas por GCC en paquetes de benchmarks de tiempo real, integrando sus especificaciones temporales en la herramienta de análisis del WCET.

Si los bucles vectorizados forman parte del código donde se concentra la mayor parte del tiempo de ejecución, nuestros resultados confirman que el WCET se reduce de manera significativa (obteniendo hasta una reducción por un factor de 3 del WCET en alguno de los benchmarks). Sin embargo, en los benchmarks en los que no se da esta situación no hay tanto margen de mejora. Esto se debe en parte a las propias limitaciones de la vectorización automática, y también de la propia programación de los benchmarks. Por tanto, para obtener mejores resultados es necesario o bien reescribir el código facilitando el trabajo de vectorización del compilador o bien escribir código vectorial explícito. En cualquiera de los dos casos la eficiencia de la vectorización queda en manos de quien programa.

Otra conclusión que recogemos en este capítulo es que debe evitarse la vectorización si el número de iteraciones de los bucles vectorizados no es lo suficientemente grande para compensar el sobrecoste de utilizar código vectorial. Este sobrecoste se debe al código generado para seleccionar el modo escalar o vectorial, a las instrucciones adicionales que preparan el bucle o a los fallos adicionales en cache de instrucciones por las instrucciones vectoriales.

En resumen, estos resultados junto con el hecho de que el procesamiento vectorial está convirtiéndose en una utilidad básica en la mayoría de los procesadores del mercado alientan a usar los recursos vectoriales en sistemas de tiempo real estricto teniendo en cuenta las cuestiones planteadas como un punto de partida para un estudio más extenso.

En el capítulo 5 abordamos el análisis del WCET en presencia de cache de datos a través del análisis de patrones de accesos de la operación de transposición de matrices, una operación básica en sistemas de tiempo real. Determinar de forma precisa y, si es posible, reducir el número de accesos y aumentar la tasa de aciertos conlleva un impacto directo en el cálculo del WCET.

En este capítulo obtenemos las expresiones analíticas que permiten determinar el comportamiento ideal de la cache de datos (fallos obligatorios) para la transposición de matrices y cuál será la configuración óptima para una cache de datos LRU en la versión *tiling* del algoritmo. Tras ello, validamos las expresiones anteriores y las comparamos con la versión *cache-oblivious* del algoritmo por medio de simulaciones. En esta comparación observamos que, con el tamaño adecuado de tile, la versión tiling del algoritmo obtiene una tasa de aciertos en datos igual o mejor. Finalmente, ofrecemos resultados experimentales en hardware real y una comparación con PLRU.

Nuestra principal aportación, la cual era objeto de este estudio, es proporcionar resultados que permiten determinar de forma sencilla los aciertos y fallos de la versión *tiling* de la transposición de matrices (imprescindible para aplicaciones de tiempo real) en una cache LRU de datos y, bajo determinadas configuraciones de cache, en una cache PLRU.

Adicionalmente a estos resultados podemos concluir que no van a ser necesarias más de dos vías en la cache de datos y unos pocos conjuntos para lograr la tasa de aciertos ideal y por tanto las vías que no se usen en una cache de datos asociativa por conjuntos pueden desconectarse sin ningún impacto negativo proporcionando un ahorro de energía (desde un 3 % a un 46 % en nuestros experimentos, dependiendo de la plataforma). Además, pueden reservarse en exclusiva dos vías en las caches de los últimos niveles para la transposición de matrices, evitando la contaminación de otros procesos.

El trabajo desarrollado en este capítulo se ha presentado en *A. Pedro-Zapater, C. Rodríguez, J. Segarra, R.G. Tejero y V. Viñals-Yúfera (2019), «Tasa de aciertos ideal y predecible para la transposición de matrices en caches de datos», XXX Jornadas de Paralelismo (JP2019), 18-20 Septiembre 2019, Cáceres.* y se ha publicado en *A. Pedro-Zapater, C. Rodríguez, J. Segarra, R.G. Tejero y V. Viñals-Yúfera (2020), «Ideal and Predictable Hit Ratio for Matrix Transposition in Data Caches», Mathematics., February, 2020. Vol. 8(2), pp. 184. MDPI AG.*

Bibliografía

[ALE02] Todd M. Austin, Eric Larson y Dan Ernst. «SimpleScalar: An Infrastructure for Computer System Modeling». En: *IEEE Computer* 35.2 (2002), págs. 59-67. DOI: 10.1109/2.982917.

[AP] Alexis Arnaud e Isabelle Puaut. «Dynamic instruction cache locking in hard real-time systems». En: *In RTNS*.

[Apa+08] Luis C. Aparicio, Juan Segarra, Clemente Rodríguez, J. L. Villarroel y Víctor Viñals. «Avoiding the WCET Overestimation on LRU Instruction Cache». En: *The Fourteenth IEEE Internationl Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2008, Kaohisung, Taiwan, 25-27 August 2008, Proceedings*. IEEE Computer Society, 2008, págs. 393-398. DOI: 10.1109/RTCSA.2008.10.

[Apa+10] Luis C. Aparicio, Juan Segarra, Clemente Rodríguez y Víctor Viñals. «Combining Prefetch with Instruction Cache Locking in Multitasking Real-Time Systems». En: *16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2010, Macau, SAR, China, 23-25 August 2010*. IEEE Computer Society, 2010, págs. 319-328. DOI: 10.1109/RTCSA.2010.8.

[Apa+11] Luis C. Aparicio, Juan Segarra, Clemente Rodríguez y Víctor Viñals. «Improving the WCET computation in the presence of a lockable instruction cache in multitasking real-time systems». En: *J. Syst. Archit.* 57.7 (2011), págs. 695-706. DOI: 10.1016/j.sysarc.2010.08.008.

[AR13] Andreas Abel y Jan Reineke. «Measurement-based modeling of the cache replacement policy». En: *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013*. IEEE Computer Society, 2013, págs. 65-74. DOI: 10.1109/RTAS.2013.6531080.

[AR14] Andreas Abel y Jan Reineke. «Reverse engineering of cache replacement policies in Intel microprocessors and their evaluation». En: *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*. IEEE Computer Society, 2014, págs. 141-142. DOI: 10.1109/ISPASS.2014.6844475.

[ARM] GNU ARM. *GNU ARM Embedded Toolchain Version 6-2017-q2-update*. URL: developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads (visitado 04-05-2020).

[ARS13] Luis Carlos Aparicio Cardiel, Clemente Rodríguez Lafuente y Juan Segarra Flor. «» En: (2013). Presentado: 21 02 2013.

[AU14] Pavel G. Zaykov Arthur Pyka Mathias Rohde y Sascha Uhrig. «Case Study: On-Demand Coherent Cache for Avionic Applications». En: *2nd Workshop on High-performance and Real-time Embedded Systems*. 2014.

[Bac+94] David F. Bacon, Jyh-Herng Chow, Dz-ching Ju, Kalyan Muthukumar y Vivek Sarkar. «A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness». En: *Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research, October 31 - November 3, 1994, Toronto, Ontario, Canada*. Ed. por John E. Botsford, Ann Gawman, W. Morven Gentleman, Evelyn Kidd, Kelly A. Lyons, Jacob Slonim y J. Howard Johnson. IBM, 1994, pág. 3. URL: dl.acm.org/citation.cfm?id=782188.

[Bal+10] Clément Ballabriga, Hugues Cassé, Christine Rochange y Pascal Sainrat. «OTAWA: An Open Toolbox for Adaptive WCET Analysis». En: *Software Technologies for Embedded and Ubiquitous Systems - 8th IFIP WG 10.2 International Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings*. Ed. por Sang Lyul Min, Robert G. Pettit IV, Peter P. Puschner y Theo Ungerer. Vol. 6399. Lecture Notes in Computer Science. Springer, 2010, págs. 35-46. DOI: 10.1007/978-3-642-16256-5_6.

[Bal+16] Sanford Ballard, James Hipp, Brian Kraus, Andre Encarnacao y Christopher Young. «GeoTess: A Generalized Earth Model Software Utility». En: *Seismological Research Letters* 87 (mayo de 2016), págs. 719-725. DOI: 10.1785/0220150222.

[Bal+17] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee y Vaishnav Srinivas. «CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories». En: *TACO* 14.2 (2017), 14:1-14:25. DOI: 10.1145/3085572.

[Bao+18] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet y P. Sadayappan. «Analytical modeling of cache behavior for affine programs». En: *PACMPL* 2.POPL (2018), 32:1-32:26. DOI: 10.1145/3158120.

[BC11] Shekhar Borkar y Andrew A. Chien. «The future of microprocessors». En: *Commun. ACM* 54.5 (2011), págs. 67-77. DOI: 10.1145/1941487.1941507.

[Ber06] Christoph Berg. «PLRU Cache Domino Effects». En: *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, July 4, 2006, Dresden, Germany*. Ed. por Frank Mueller. Vol. 4. OASICS. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006. URL: drops.dagstuhl.de/opus/volltexte/2006/672.

[Bin+11] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark D. Hill y David A. Wood. «The gem5 simulator». En: *SIGARCH Computer Architecture News* 39.2 (2011), págs. 1-7. DOI: 10.1145/2024716.2024718.

[BMS08] Armelle Bonenfant, Marianne de Michiel y Pascal Sainrat. «oRange: A tool for static loop bound analysis». En: *Proceedings of the Workshop on Resource Analysis*. 2008.

[Cam+03] A. M. Campoy, A. Perles, F. Rodriguez y J. V. Busquets-Mataix. «Static use of locking caches vs. dynamic use of locking caches for real-time

systems». En: *CCECE 2003 - Canadian Conference on Electrical and Computer Engineering. Toward a Caring and Humane Technology (Cat. No.03CH37436)*. Vol. 2. 2003, 1283-1286 vol.2.

[Cam+05] Antonio Martí Campoy, Eugenio Tamura, S. Sáez, Francisco Rodríguez y José V. Busquets-Mataix. «On Using Locking Caches in Embedded Real-Time Systems». En: *Embedded Software and Systems, Second International Conference, ICESS 2005, Xi'an, China, December 16-18, 2005, Proceedings*. Ed. por Laurence Tianruo Yang, Xingshe Zhou, Wei Zhao, Zhaohui Wu, Yian Zhu y Man Lin. Vol. 3820. Lecture Notes in Computer Science. Springer, 2005, págs. 150-159. DOI: 10.1007/11599555_17.

[Caz+13] Francisco J. Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery D. Berger, Jaume Abella, Franck Wartel, Michael Houston, Luca Santinelli, Leonidas Kosmidis, Code Lo y Dorin Maxim. «PROARTIS: Probabilistically Analyzable Real-Time Systems». En: *ACM Trans. Embedded Comput. Syst.* 12.2s (2013), 94:1-94:26. DOI: 10.1145/2465787.2465796.

[CB02] Antoine Colin y Guillem Bernat. «Scope-Tree: A Program Representation for Symbolic Worst-Case Execution Time Analysis». En: *14th Euromicro Conference on Real-Time Systems (ECRTS 2002), 19-21 June 2002, Vienna, Austria, Proceedings*. IEEE Computer Society, 2002, pág. 50. DOI: 10.1109/EMRTS.2002.1019185.

[CC77] Patrick Cousot y Radhia Cousot. «Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints». En: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. Ed. por Robert M. Graham, Michael A. Harrison y Ravi Sethi. ACM, 1977, págs. 238-252. DOI: 10.1145/512950.512973.

[Che14] Maryline Chetto. *Real-time systems scheduling*. ISTE Ltd, 2014.

[CIM01] Marti Campoy, A. Perles Ivars y J. V. Busquets Mataix. «Static Use of Locking Caches in Multitask Preemptive Real-Time Systems». En: *In Proceedings of IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium*. 2001.

[CK07] Jian-Jia Chen y Chin-Fu Kuo. «Energy-Efficient Scheduling for Real-Time Systems on Dynamic Voltage Scaling (DVS) Platforms». En: *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007), 21-24 August 2007, Daegu, Korea*. IEEE Computer Society, 2007, págs. 28-38. DOI: 10.1109/RTCSA.2007.37.

[CP00] Antoine Colin e Isabelle Puaut. «Worst Case Execution Time Analysis for a Processor with Branch Prediction». En: *Real-Time Systems* 18.2/3 (2000), págs. 249-274. DOI: 10.1023/A:1008149332687.

[CS00] Siddhartha Chatterjee y Sandeep Sen. «Cache-Efficient Matrix Transposition». En: *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture, Toulouse, France, January 8-12, 2000*. IEEE Computer Society, 2000, págs. 195-205. DOI: 10.1109/HPCA.2000.824350.

[DLM13] Huping Ding, Yun Liang y Tulika Mitra. «Integrated instruction cache analysis and locking in multitasking real-time systems». En: *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*. ACM, 2013, 147:1-147:10. DOI: 10.1145/2463209.2488916.

[Fal+16] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sorensen, Peter Wägemann y Simon Wegener. «TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research». En: *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*. Ed. por Martin Schoeberl. Vol. 55. OASICS. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 2:1-2:10. DOI: 10.4230/OASIcs.WCET.2016.2.

[FH04] Christian Ferdinand y Reinhold Heckmann. «aiT: worst case execution time prediction by static program analysis». En: *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22-27 August 2004, Toulouse, France*. Ed. por René Jacquart. Vol. 156. IFIP. Kluwer/Springer, 2004, págs. 377-383. DOI: 10.1007/978-1-4020-8157-6_29.

[Fla+02] Krisztián Flautner, Nam Sung Kim, Steven M. Martin, David T. Blaauw y Trevor N. Mudge. «Drowsy Caches: Simple Techniques for Reducing Leakage Power». En: *29th International Symposium on Computer Architecture (ISCA 2002), 25-29 May 2002, Anchorage, AK, USA*. Ed. por Yale N. Patt, Dirk Grunwald y Kevin Skadron. IEEE Computer Society, 2002, págs. 148-157. DOI: 10.1109/ISCA.2002.1003572.

[FLS13] B. Fitzgerald, S. Lopez y Julio Sahuquillo. «Drowsy cache partitioning for reduced static and dynamic energy in the cache hierarchy». En: *International Green Computing Conference, IGCC 2013, Arlington, VA, USA, June 27-29, 2013, Proceedings*. IEEE Computer Society, 2013, págs. 1-6. DOI: 10.1109/IGCC.2013.6604475.

[Fre12] Inc. Free Software Foundation. *GCC, the GNU Compiler Collection*. 2012. URL: gcc.gnu.org/ (visitado 20-05-2020).

[Fri+12] Matteo Frigo, Charles E. Leiserson, Harald Prokop y Sridhar Ramachandran. «Cache-Oblivious Algorithms». En: *ACM Trans. Algorithms* 8.1 (2012), 4:1-4:22. DOI: 10.1145/2071379.2071383.

[Fri+99] Matteo Frigo, Charles E. Leiserson, Harald Prokop y Sridhar Ramachandran. «Cache-Oblivious Algorithms». En: *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*. IEEE Computer Society, 1999, págs. 285-298. DOI: 10.1109/SFFCS.1999.814600.

[FW99] Christian Ferdinand y Reinhard Wilhelm. «Efficient and Precise Cache Behavior Prediction for Real-Time Systems». En: *Real-Time Systems* 17.2-3 (1999), págs. 131-181. DOI: 10.1023/A:1008186323068.

[Ger+11] Mike Gerdes, Julian Wolf, Irakli Guliashvili, Theo Ungerer, Michael Houston, Guillem Bernat, Stefan Schnitzler y Hans Regler. «Large drilling machine control code - Parallelisation and WCET speedup». En: *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on, SIES 2011. Vasteras, Sweden, June 15-17, 2011*. IEEE, 2011, págs. 91-94. DOI: 10.1109/SIES.2011.5953688.

[Gra+13] Ruben Gran, Juan Segarra, Clemente Rodríguez, Luis C. Aparicio y Víctor Viñals. «Optimizing a combined WCET-WCEC problem in instruction fetching for real-time systems». En: *J. Syst. Archit.* 59.9 (2013), págs. 667-678. DOI: 10.1016/j.sysarc.2013.07.012.

[Gra+15] Ruben Gran, Juan Segarra, A. Pedro-Zapater, Luis C. Aparicio, Víctor Viñals y Clemente Rodríguez. «A predictable hardware to exploit temporal reuse in real-time and embedded systems». En: *J. Syst. Archit.* 61.5-6 (2015), págs. 227-238. DOI: 10.1016/j.sysarc.2015.05.001.

[Gus+03] Jan Gustafsson, Björn Lisper, Christer Sandberg y Nerina Bermudo. «A Tool for Automatic Flow Analysis of C-programs for WCET Calculation». En: *8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003), 15-17 January 2003, Guadalajara, Mexico*. IEEE Computer Society, 2003, págs. 106-112. DOI: 10.1109/WORDS.2003.1218072.

[Gus+10] Jan Gustafsson, Adam Betts, Andreas Ermedahl y Björn Lisper. «The Mälardalen WCET Benchmarks – Past, Present and Future». En: *WCET2010*. Ed. por Björn Lisper. Brussels, Belgium, jul. de 2010, págs. 137-147. DOI: 10.4230/OASIcs.WCET.2010.136.

[Gus00] Jan Gustafsson. «Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation». Tesis doct. Department of Computer Engineering, Mälardalen University, Box 883, S-721 23 Västerås, Sweden, y Department of Computer Systems, Information Technology, Uppsala University, Box 325, S-751 05 Uppsala, Sweden, mayo de 2000. URL: http://www.es.mdh.se/publications/231-.

[Hea+99] Christopher A. Healy, Robert D. Arnold, Frank Mueller, David B. Whalley y Marion G. Harmon. «Bounding Pipeline and Instruction Cache Performance». En: *IEEE Trans. Computers* 48.1 (1999), págs. 53-70. DOI: 10.1109/12.743411.

[Hon+] Honeywell, BSC, Université Toulouse III - Paul Sabatier y Rapita Systems. *MERASA Project*. URL: cordis.europa.eu/project/id/216415/es (visitado 15-06-2020).

[Hon+16] Changwan Hong, Wenlei Bao, Albert Cohen, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam y P. Sadayappan. «Effective padding of multidimensional arrays to avoid cache conflict misses». En: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. Ed. por Chandra Krintz y Emery Berger. ACM, 2016, págs. 129-144. DOI: 10.1145/2908080.2908123.

[HRP17] Damien Hardy, Benjamin Rouxel e Isabelle Puaut. «The Heptane Static Worst-Case Execution Time Estimation Tool». En: *17th International Workshop on Worst-Case Execution Time Analysis, WCET 2017, June 27, 2017, Dubrovnik, Croatia*. Ed. por Jan Reineke. Vol. 57. OASICS. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 8:1-8:12. DOI: 10.4230/OASIcs.WCET.2017.8. URL: https://doi.org/10.4230/OASIcs.WCET.2017.8.

[II] INRIA e I3S. *MasCotTE Project*. URL: www-sop.inria.fr/teams/mascotte/ (visitado 15-06-2020).

[Ins20] Global Market Insights. *Embedded Systems Market Size By Component (Hardware, [ASIC & ASSP, Microcontroller, Microprocessor, Power Management Integrated Circuit (PMIC), Field Programmable Gate Array (FPGA), Digital Signal Processor (DSP), Memory], Software (OS, Middleware)], By Function (Standalone System, Real-Time System, Network System, Mobile System), By Application (Automotive, Consumer Electronics, Manufacturing, Retail, Media & Entertainment, Military & Defense, Telecom), Industry Analysis Report, Regional Outlook, Application Potential, Competitive Market Share & Forecast, 2020-2026*. 2020. URL: gminsights.com/industry-analysis/embedded-system-market (visitado 14-09-2020).

[IUU] INRIA Siège, Université Toulouse III - Paul Sabatier y Université París VI- Pierre et Marie Curie. *MORE Project*. URL: anr.fr/Project-ANR-06-ARFU-0002 (visitado 15-06-2020).

[JML06] Ramkumar Jayaseelan, Tulika Mitra y Xianfeng Li. «Estimating the Worst-Case Energy Consumption of Embedded Software». En: *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006), 4-7 April 2006, San Jose, California, USA*. IEEE Computer Society, 2006, págs. 81-90. DOI: 10.1109/RTAS.2006.17.

[Kos+14] Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones y Francisco J. Cazorla. «Efficient Cache Designs for Probabilistically Analysable Real-Time Systems». En: *IEEE Trans. Computers* 63.12 (2014), págs. 2998-3011. DOI: 10.1109/TC.2013.182.

[Kuc+81] David J. Kuck, Robert H. Kuhn, David A. Padua, Bruce Leasure y Michael Wolfe. «Dependence Graphs and Compiler Optimizations». En: *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages, Williamsburg, Virginia, USA, January 1981*. Ed. por John White, Richard J. Lipton y Patricia C. Goldberg. ACM Press, 1981, págs. 207-218. DOI: 10.1145/567532.567555.

[Lei03] Charles E. Leiserson. «Cache-Oblivious Algorithms». En: *Algorithms and Complexity, 5th Italian Conference, CIAC 2003, Rome, Italy, May 28-30, 2003, Proceedings*. Ed. por Rossella Petreschi, Giuseppe Persiano y Riccardo Silvestri. Vol. 2653. Lecture Notes in Computer Science. Springer, 2003, pág. 5. DOI: 10.1007/3-540-44849-7_5.

[Li+07] Xianfeng Li, Liang Yun, Tulika Mitra y Abhik Roychoudhury. «Chronos: A timing analyzer for embedded software». En: *Sci. Comput. Program.* 69.1-3 (2007), págs. 56-67. DOI: 10.1016/j.scico.2007.01.014.

[Lima] ARM Limited. *Armv6 reference manual*. URL: infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0489e/CJAJIIGG.html (visitado 11-05-2020).

[Limb] ARM Limited. *Cortex-A8 reference manual*. URL: infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/DDI0344K_cortex_a8_r3p2_trm.pdf (visitado 11-05-2020).

[Lim+95] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon y Chong-Sang Kim. «An Accurate Worst Case Timing Analysis for RISC Processors». En: *IEEE Trans. Software Eng.* 21.7 (1995), págs. 593-604. DOI: 10.1109/32.392980.

[Lis14] Björn Lisper. «SWEET - A Tool for WCET Flow Analysis (Extended Abstract)». En: *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II*. Ed. por Tiziana Margaria y Bernhard Steffen. Vol. 8803. Lecture Notes in Computer Science. Springer, 2014, págs. 482-485. DOI: 10.1007/978-3-662-45231-8_38.

[LM95] Yau-Tsun Steven Li y Sharad Malik. «Performance Analysis of Embedded Software Using Implicit Path Enumeration». En: *Proceedings of the 32st Conference on Design Automation, San Francisco, California, USA, Moscone Center, June 12-16, 1995*. Ed. por Bryan Preas. ACM Press, 1995, págs. 456-461. DOI: 10.1145/217474.217570.

[LMW96] Yau-Tsun Steven Li, Sharad Malik y Andrew Wolfe. «Cache modeling for real-time software: beyond direct mapped instruction caches». En: *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96), December 4-6, 1996, Washington, DC, USA*. IEEE Computer Society, 1996, págs. 254-263. DOI: 10.1109/REAL.1996.563722.

[LPR14] Hanbing Li, Isabelle Puaut y Erven Rohou. «Traceability of Flow Information: Reconciling Compiler Optimizations and WCET Estimation». En: *22nd International Conference on Real-Time Networks and Systems, RTNS '14, Versaille, France, October 8-10, 2014*. Ed. por Mathieu Jan, Belgacem Ben Hedia, Joël Goossens y Claire Maiza. ACM, 2014, pág. 97. DOI: 10.1145/2659787.2659805.

[LPR15] Hanbing Li, Isabelle Puaut y Erven Rohou. «Tracing Flow Information for Tighter WCET Estimation: Application to Vectorization». En: *21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2015, Hong Kong, China, August 19-21, 2015*. IEEE Computer Society, 2015, págs. 217-226. DOI: 10.1109/RTCSA.2015.18.

[LRW91] Monica S. Lam, Edward E. Rothberg y Michael E. Wolf. «The Cache Performance and Optimizations of Blocked Algorithms». En: *ASPLOS-IV Proceedings - Forth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, USA, April 8-11, 1991*. Ed. por David A. Patterson y Bob Rau. ACM Press, 1991, págs. 63-74. DOI: 10.1145/106972.106981.

[Lun02] Thomas Lundqvist. «A WCET Analysis Method for Pipelined Microprocessors with Cache Memories». Tesis doct. Chalmers University of Technology, Gothenburg, Sweden, 2002. URL: http://publications.lib.chalmers.se/publication/606-a-wcet-analysis-method-for-pipelined-microprocessors-with-cache-memories.

[Mal+11] Saeed Maleki, Yaoqing Gao, Maríaa Jesús Garzarán, Tommy Wong y David A. Padua. «An Evaluation of Vectorizing Compilers». En: *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011, Galveston, TX, USA, October 10-14, 2011*. Ed. por Lawrence Rauchwerger y Vivek Sarkar. IEEE Computer Society, 2011, págs. 372-382. DOI: 10.1109/PACT.2011.68.

[Mal08] Rajib Mall. *Real-time systems: theory and practice*. Published by Dorling Kindersley (India), licensees of Pearson Education in South Asia, 2008.

[Mit16] Sparsh Mittal. «A Survey of Techniques for Cache Locking». En: *ACM Trans. Design Autom. Electr. Syst.* 21.3 (2016), 49:1-49:24. DOI: 10.1145/2858792.

[Mor66] G. H. Morton. «A computer oriented geodetic data base and a new technique in file sequencing». En: 1966.

[Pan+99] Preeti Ranjan Panda, Hiroshi Nakamura, Nikil D. Dutt y Alexandru Nicolau. «Augmenting Loop Tiling with Data Alignment for Improved Cache Performance». En: *IEEE Trans. Computers* 48.2 (1999), págs. 142-149. DOI: 10.1109/12.752655.

[Par] ParMERASA. *parMERASA use cases*. URL: http://www.parmerasa.eu/files/parMERASA%5C_Brochure.pdf (visitado 01-06-2020).

[PD02] Isabelle Puaut y David Decotigny. «Low-Complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems». En: *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02), Austin, Texas, USA, December 3-5, 2002*. IEEE Computer Society, 2002, págs. 114-123. DOI: 10.1109/REAL.2002.1181567.

[Ped+20a] Alba Pedro-Zapater, Clemente Rodríguez, Juan Segarra, Rubén Gran Tejero y Víctor Viñals-Yúfera. «Ideal and Predictable Hit Ratio for Matrix Transposition in Data Caches». En: *Mathematics* 8.2 (2020), pág. 184. DOI: 10.3390/math8020184.

[Ped+20b] Alba Pedro-Zapater, Juan Segarra, Rubén Gran Tejero, Víctor Viñals y Clemente Rodríguez. «Reducing the WCET and analysis time of systems with simple lockable instruction caches». En: *PLOS ONE* 15.3 (mar. de 2020), págs. 1-21. DOI: 10.1371/journal.pone.0229980.

[Pua06] Isabelle Puaut. «WCET-Centric Software-controlled Instruction Caches for Hard Real-Time Systems». En: *18th Euromicro Conference on Real-Time Systems, ECRTS'06, 5-7 July 2006, Dresden, Germany, Proceedings*. IEEE Computer Society, 2006, págs. 217-226. DOI: 10.1109/ECRTS.2006.32.

[Pye16] Larry Pyeatt. *Modern Assembly Language Programming with the ARM Processor*. Newnes, dic. de 2016, págs. 297-361. ISBN: 9780128036983.

[Rei+07] Jan Reineke, Daniel Grund, Christoph Berg y Reinhard Wilhelm. «Timing predictability of cache replacement policies». En: *Real-Time Systems* 37.2 (2007), págs. 99-122. DOI: 10.1007/s11241-007-9032-3.

[RJ16] Bradley T Rearden y Matthew Anderson Jessee. *SCALE code system*. Inf. téc. Oak Ridge National Laboratory (ORNL), 2016.

[SA00] Friedhelm Stappert y Peter Altenbernd. «Complete worst-case execution time analysis of straight-line hard real-time programs». En: *J. Syst. Archit.* 46.4 (2000), págs. 339-355. DOI: 10.1016/S1383-7621(99)00010-7.

[SEE01] Friedhelm Stappert, Andreas Ermedahl y Jakob Engblom. «Efficient longest executable path search for programs with complex flows and pipeline effects». En: *Proceedings of the 2001 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 2001, Atlanta, Georgia, USA, November 16-17, 2001*. Ed. por Guang R. Gao, Trevor N. Mudge y Krishna V. Palem. ACM, 2001, págs. 132-140. DOI: 10.1145/502217.502240.

[Seg+12] Juan Segarra, Clemente Rodríguez, Ruben Gran Tejero, Luis C. Aparicio y Víctor Viñals. «A Small and Effective Data Cache for Real-Time Multitasking Systems». En: *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium, Beijing, China, April 16-19, 2012*. Ed. por Marco Di Natale. IEEE Computer Society, 2012, págs. 45-54. DOI: 10.1109/RTAS.2012.11.

[Seg+15] Juan Segarra, Clemente Rodríguez, Ruben Gran, Luis C. Aparicio y Víctor Viñals. «ACDC: Small, Predictable and High-Performance Data Cache». En: *ACM Trans. Embedded Comput. Syst.* 14.2 (2015), 38:1-38:26. DOI: 10.1145/2677093.

[Sha+04] Lui Sha, Tarek F. Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore P. Baker, Alan Burns, Giorgio C. Buttazzo, Marco Caccamo, John P. Lehoczky y Aloysius K. Mok. «Real Time Scheduling Theory: A Historical Perspective». En: *Real-Time Systems* 28.2-3 (2004), págs. 101-155. DOI: 10.1023/B:TIME.0000045315.61234.1e.

[Sim] SimpleScalar LLC. *SimpleScalar*. URL: simplescalar.com (visitado 15-06-2020).

[The+03] Stephan Thesing, Jean Souyris, Reinhold Heckmann, Famantanantsoa Randimbivololona, Marc Langenbach, Reinhard Wilhelm y Christian Ferdinand. «An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software». En: *2003 International Conference on Dependable Systems and Networks (DSN 2003), 22-25 June 2003, San Francisco, CA, USA, Proceedings*. IEEE Computer Society, 2003, págs. 625-632. DOI: 10.1109/DSN.2003.1209972.

[Tid] Tidorum Ltd. *Bound-T*. URL: bound-t.com (visitado 15-06-2020).

[Tra] Traces, Irit Lab, University of Toulouse. *Otawa, Open Tool for Adaptive WCET Analyses*. URL: otawa.fr/ (visitado 25-04-2020).

[TRS04] D. Tsifakis, Alistair P. Rendell y Peter E. Strazdins. «Cache Oblivious Matrix Transposition: Simulation and Experiment». En: *Computational Science - ICCS 2004, 4th International Conference, Kraków, Poland, June 6-9, 2004, Proceedings, Part II*. Ed. por Marian Bubak, G. Dick van Albada, Peter M. A. Sloot y Jack J. Dongarra. Vol. 3037. Lecture Notes in Computer Science. Springer, 2004, págs. 17-25. DOI: 10.1007/978-3-540-24687-9_3.

[Ves07] Steve Vestal. «Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance». En: *Proceedings of the 28th IEEE Real-Time Systems Symposium (RTSS 2007), 3-6 December 2007, Tucson, Arizona, USA*. IEEE Computer Society, 2007, págs. 239-243. DOI: 10.1109/RTSS.2007.47.

[Wil+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat y Per Stenström. «The worst-case execution-time problem - overview of methods and survey of tools». En: *ACM Trans. Embedded Comput. Syst.* 7.3 (2008), 36:1-36:53. DOI: 10.1145/1347375.1347389.

[Xil] Xilinx. *MPSoC*. URL: xilinx.com/products/technology/ultrascale-mpsoc.html (visitado 14-07-2020).

[Yot+07] Kamen Yotov, Thomas Roeder, Keshav Pingali, John A. Gunnels y Fred G. Gustavson. «An experimental comparison of cache-oblivious and cache-conscious programs». En: *SPAA 2007: Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Diego, California, USA, June 9-11, 2007*. Ed. por Phillip B. Gibbons y Christian Scheideler. ACM, 2007, págs. 93-104. DOI: 10.1145/1248377.1248394.

[ZWY17] Wenguang Zheng, Hui Wu y Qing Yang. «WCET-Aware Dynamic I-Cache Locking for a Single Task». En: *TACO* 14.1 (2017), 4:1-4:26. DOI: 10.1145/3046683.