TESIS DE LA UNIVERSIDAD DE ZARAGOZA

2013

17

Luis Carlos Aparicio Cardiel

Jerarquía de memoria para instrucciones y cálculo del WCET

Departamento
Informática e Ingeniería de Sistemas

Director/es
Rodríguez Lafuente, Clemente
Segarra Flor, Juan

ISSN 2254-7606

Prensas de la Universidad
Universidad Zaragoza

Universidad Zaragoza
1542

Tesis Doctoral

JERARQUÍA DE MEMORIA PARA INSTRUCCIONES Y CÁLCULO DEL WCET

Autor

Luis Carlos Aparicio Cardiel

Director/es

Rodríguez Lafuente, Clemente
Segarra Flor, Juan

UNIVERSIDAD DE ZARAGOZA

Informática e Ingeniería de Sistemas

2013

Universidad de Zaragoza

Dpto. Informática e Ingeniería de Sistemas

Tesis Doctoral

Jerarquía de Memoria para Instrucciones y Cálculo del WCET

D. Luis C. Aparicio Cardiel

Directores:

Dr. Juan Segarra Flor

Dr. Clemente Rodríguez Lafuente

Zaragoza, Octubre 2012

Jerarquía de Memoria para Instrucciones y Cálculo del WCET

Memoria presentada por

D. Luis C. Aparicio Cardiel

para obtener el título de Doctor en Informática

Dirigida por:

Dr. Juan Segarra Flor

Dr. Clemente Rodríguez Lafuente

Universidad de Zaragoza

Dpto. Informática e Ingeniería de Sistemas

Zaragoza, Octubre 2012

Resumen

Uno de los principales retos de los sistemas de tiempo real es el cálculo del tiempo de ejecución del peor caso (WCET/ *Worst Case Execution Time*), es decir, determinar el tiempo de ejecución del *camino más largo*. El cálculo del WCET tiene que ser seguro y también preciso, ya que la *planificabilidad* del sistema debe estar garantizada antes de su ejecución.

El mercado de los sistemas de tiempo real añade una restricción importante en el diseño de la jerarquía de memoria, la necesidad de conocer un límite máximo del tiempo de ejecución, ya que este tiempo depende en gran medida del número máximo de fallos de cache que se producirán durante la ejecución. Pero, el análisis del comportamiento temporal en el peor caso de la cache es complejo, por lo tanto los diseñadores de sistemas de tiempo real descartan su utilización.

En esta Tesis se analiza el comportamiento en el peor caso de varias jerarquías de memoria para instrucciones. En concreto se estudia, tanto una cache de instrucciones *convencional*, como una cache que pueda fijar su contenido. El principal objetivo de este análisis es conseguir el mejor rendimiento, en un sistema de tiempo real, de la jerarquía de memoria estudiada. Así pues, también se presentan diferentes técnicas de análisis y cálculo del WCET para cada una de las jerarquías de memoria estudiadas.

Para una cache de instrucciones *convencional* con algoritmo de reemplazo LRU, analizamos su comportamiento en el peor caso y demostramos que el número de caminos *relevantes* generado por estructuras condicionales dentro de bucles no depende del número de iteraciones del bucle, sino que depende del número de caminos del condicional. Esto permite obtener la contribución exacta al WCET de los accesos a memoria, cuando el número de caminos condicionales dentro de un bucle no es grande. Así pues, proponemos una técnica para determinar la contribución exacta al WCET de los accesos a memoria. A esta técnica la denominamos *poda dinámica de caminos*.

Estudiamos una jerarquía de memoria formada por un (LB/ *Line Buffer*) y una cache que pueda fijar su contenido (*Lockable iCache*). Para esta jerarquía de memoria proponemos un algoritmo óptimo que selecciona las líneas a fijar en la

cache durante la ejecución de cada tarea del sistema. A este algoritmo lo hemos denominado *Lock-MS* (*Lock for Maximize Schedulability*). Además, proponemos una nueva jerarquía de memoria en sistemas de tiempo real con hardware de prebúsqueda secuencial (PB/ *Prefetch Buffer*) y analizamos su influencia en el WCET de cada tarea. El LB y el PB capturan muy bien la localidad espacial y reducen considerablemente el WCET de las tareas. También permiten reducir la capacidad de la *Lockable iCache* sin comprometer la *planificabilidad* del sistema.

Dado un conjunto de tareas que podrían formar un sistema de tiempo real, para cada una de las jerarquías de memoria analizadas, proponemos técnicas de análisis y cálculo del WCET totalmente seguro y más preciso que el obtenido con las técnicas de análisis ya descritas en la literatura.

Finalmente, también se presenta un estudio sobre el consumo energético de una jerarquía de memoria formada por un LB, un PB y una *Lockable iCache*. Los resultados de este estudio indican que el camino del WCET de una tarea no coincide con el camino del WCEC (*Worst Case Energy Consumption*) de dicha tarea.

Palabras Clave: WCET, tiempo de ejecución en el peor caso, jerarquía de memoria, memoria cache, camino más largo, caminos relevantes, máxima *planificabilidad*, prebúsqueda secuencial, WCEC, consumo de energía en el peor caso.

Para Laura y
Moisés

Agradecimientos

Agradezco a mi familia y amigos los ánimos que me han dado durante la elaboración de esta Tesis, en especial a mis padres, ya que siempre me han apoyado en todos los estudios que he realizado.

Quiero agradecer a Víctor Viñals y a José Luis Villarroel la oportunidad que me dieron de trabajar con ellos cuando comencé mis estudios de doctorado y por supuesto a mis directores de Tesis, Juan y Clemente, que han dirigido esta aventura sin dudar.

Aunque no he tenido la oportunidad de convivir con ellos día a día, también me gustaría agradecer la ayuda que todos los miembros del gaZ me ofrecen cuando estoy en Zaragoza.

Muchas gracias a todos.

Finalmente, he de agradecer la financiación recibida de las siguientes entidades y proyectos que han hecho posible este trabajo:

- Jerarquía de Memoria de Alto Rendimiento. TIN2007-66423, Ministerio de Ciencia y Tecnología (2007 - 2010).
- Interconexión y Memoria en Computadores Escalables. TIN2011-21291, Ministerio de Ciencia e Innovación (2011 - 2013).
- gaZ: Reconocimiento Grupo Consolidado de Investigación, Diputación General de Aragón (2008 - 2012).
- HiPEAC2, HiPEAC3: European Network of Excellence on High Performance and Embedded Architecture and Compilation.

Índice general

Capítulo 1

Introducción

El empleo de los sistemas informáticos es cada vez más habitual en nuestra vida diaria. Desde un sencillo microondas hasta el complejo sistema de seguridad y control de un reactor nuclear dependen del correcto funcionamiento de un sistema informático. En concreto, la mayor parte de los sistemas informáticos son *sistemas empotrados* que no suelen ser visibles directamente por el usuario y forman parte de sistemas más grandes y complejos [28, 88, 125]. Se pueden encontrar sistemas empotrados en equipos de telecomunicación, en sistemas de transporte, en equipos de fabricación y en dispositivos electrónicos de uso diario.

Los sistemas empotrados deben ser fiables y seguros, con la garantía de que en caso de fallo la reparación sea posible; deben estar siempre disponibles; no deben causar daños y no deben generar pérdida de información. Igualmente han de ser eficientes en consumo energético, ya que muchos sistemas empotrados son dispositivos móviles que funcionan con baterías; en tamaño del código, puesto que todo el código debe almacenarse en la memoria del sistema; y en la utilización de los recursos del sistema. Finalmente, si son dispositivos portátiles o dispositivos electrónicos de uso diario, además de garantizar una calidad mínima de servicio para que resulten atractivos a los usuarios, han de ser ligeros y su coste debe ser competitivo en el mercado [88, 125].

Los sistemas empotrados adquieren una relevancia especial cuando se utilizan para responder temporalmente a un evento externo. En este caso, el sistema se denomina *Sistema de Tiempo Real* [28]. La corrección de un sistema de tiempo real, no sólo está en función de los resultados obtenidos, sino que también depende del instante en el que dichos resultados son generados, por lo tanto es esencial predecir su funcionamiento.

En los sistemas de tiempo real, tanto la fiabilidad como la seguridad adquieren una relevancia especial, ya que suelen ser sistemas críticos y un mal funcionamiento en estos sistemas puede provocar incluso graves daños personales. En los automóviles que conducimos encontramos ejemplos clásicos de sistemas de

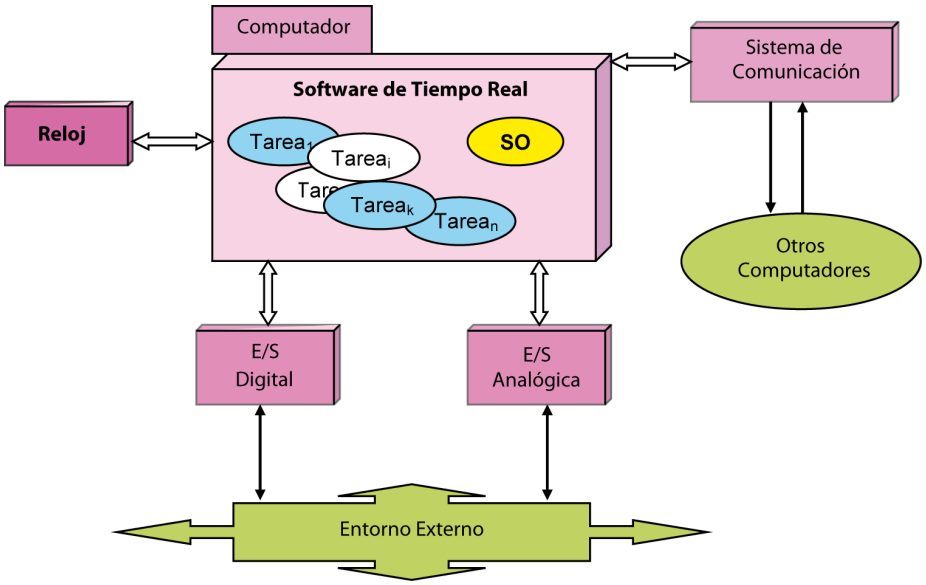


Figura 1.1: Sistema de Tiempo Real.

tiempo real, tales como el sistema de control del airbag, el sistema de control de frenado (ABS) y por supuesto el sistema de control de tracción del que ya disponen la mayoría de los vehículos actuales.

Los programas escritos para sistemas de tiempo real deben ser verificados para asegurar el correcto funcionamiento del sistema. Pero además también se debe verificar la corrección temporal del mismo, es decir, es obligatorio garantizar el tiempo de respuesta de las tareas del sistema en el peor caso. Por ejemplo, es obligatorio asegurar que el sistema de control del airbag, en caso de accidente, lanzará el airbag en un corto plazo de tiempo para prevenir los posibles daños sobre los ocupantes del vehículo.

En definitiva, un sistema de tiempo real debe responder a los diferentes eventos generados por éste en unos plazos de tiempo preestablecidos. El sistema se divide habitualmente en un conjunto de tareas que cooperan para conseguir una funcionalidad, y cada una de ellas se encarga de responder a un determinado evento o conjunto de eventos generados por el entorno (ver Figura 1.1). Para poder responder a dichos eventos en un determinado plazo de tiempo, es necesario determinar qué tarea o tareas se deben ejecutar en cada instante, y para ello es necesario definir algoritmos o políticas de planificación que determinarán si las restricciones temporales del sistema se pueden satisfacer. Por lo tanto, es necesario realizar un análisis de *planificabilidad* que tenga en cuenta: las tareas del sistema, sus plazos de finalización, sus periodos de ejecución y su tiempo de ejecución en el peor caso.

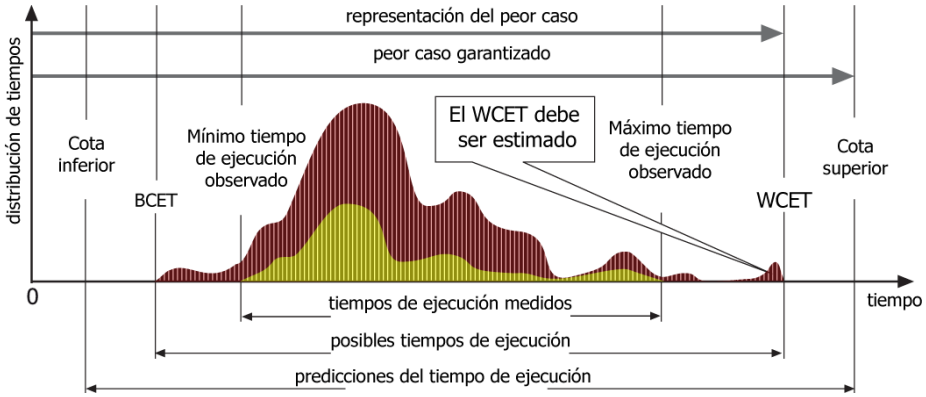


Figura 1.2: Análisis del tiempo de ejecución de una tarea [198].

Cálculo del WCET y *planificabilidad* en sistemas de tiempo real

Uno de los principales retos de los sistemas de tiempo real es el cálculo del tiempo de ejecución del peor caso (WCET/ *Worst Case Execution Time*). El WCET de un programa es el mayor tiempo de ejecución que una invocación del programa podría exhibir en una arquitectura hardware específica. Es decir, calcular el WCET es determinar el tiempo de ejecución del *camino más largo*. Obtener el WCET de una tarea de tiempo real es clave en el análisis de *planificabilidad* que garantiza el correcto funcionamiento temporal del sistema [28]. Por lo tanto, el cálculo del WCET tiene que ser seguro (*safe*), de tal forma que una sobrestimación mínima podría ser aceptada pero una subestimación no se aceptaría en ningún caso. Además, el WCET también tiene que ser preciso, ya que la *planificabilidad* del sistema debe estar garantizada antes de su ejecución. En la Figura 1.2 se muestra un exhaustivo análisis temporal de la ejecución de una tarea. Se han representado todos los posibles tiempos de ejecución de la tarea, pero sólo se han podido medir algunos de ellos. En la figura también aparecen reflejados el tiempo de ejecución del mejor caso (BCET/ *Best Case Execution Time*) y el WCET. Como se muestra en la Figura 1.2, cualquier aproximación al WCET basada en medida no es segura, ya que sólo considera un subconjunto de las posibles ejecuciones del programa.

Una vez garantizada la *planificabilidad* del sistema, también es necesario ordenar la ejecución de sus tareas. Uno de los algoritmos más sencillos y utilizados en planificación es el algoritmo *Ejecutivo Cíclico* que ordena la ejecución de las tareas mediante una tabla donde se indican los instantes en que cada tarea debe tomar y abandonar la CPU [13]. Este algoritmo es muy fácil de implementar y muy eficiente en tiempo de ejecución. Además, permite asegurar la *planificabilidad* del sistema desde el primer momento, ya que facilita la predicción de los instantes de ejecución de las tareas que son fijos y conocidos. Pero este algo-

ritmo es muy rígido y no permite incorporar nuevas tareas al sistema de forma sencilla. Además, al no existir un sistema operativo propiamente dicho, no es posible utilizar algunos servicios de comunicación del sistema.

Los sistemas operativos de tiempo real utilizan algoritmos de planificación en tiempo de ejecución basados en prioridades. En función del tipo de prioridad de las tareas, los algoritmos pueden seguir una política de planificación con expulsiones o sin ellas. Cuando en el sistema se permiten las expulsiones, una tarea abandona el procesador en cuanto otra tarea de mayor prioridad está lista para su ejecución. Si la prioridad de las tareas es constante, los algoritmos se denominan estáticos o de prioridades fijas. En la literatura encontramos algunos trabajos clásicos que definen este tipo de planificación, basada principalmente en asignar la prioridad más alta a la tarea más frecuente RMA (*Rate Monotonic Analysis*) o a la tarea más urgente DMA (*Deadline Monotonic Analysis*) [111]. La teoría subyacente en estos algoritmos también permite demostrar que la asignación óptima de prioridades debe ser inversamente proporcional a su plazo de ejecución, esto es, a menor plazo de ejecución mayor prioridad. Mediante estos algoritmos, también es posible determinar si un conjunto de tareas es *planificable*, es decir, si se cumplirán los requisitos temporales en forma de plazos de finalización marcados para ellas. Por ejemplo, un sistema con N tareas periódicas con prioridades fijas, donde C_i es el WCET y P_i es el periodo de activación de la tarea $Task_i$, será *planificable* si la utilización del procesador U verifica la siguiente expresión:

$$U = \sum_{i=i}^{N} \frac{C_i}{P_i} \leq N \cdot \left(2^{\frac{1}{N}} - 1\right) \tag{1.1}$$

Consideremos el ejemplo de la Tabla 1.1 formado por tres tareas. Para cada una de ellas se indica el tiempo de ejecución del peor caso C_i, su periodo P_i, que en este caso coincide también con su plazo de finalización D_i, y la utilización del procesador. La prioridad de cada tarea es fija y es inversamente proporcional a su periodo. Por lo tanto, según el análisis de *planificablidad* basado en medir la utilización del procesador, el sistema de la Tabla 1.1 es *planificable*, ya que $U = 0,75 < U(3) = 3 \cdot (2^{1/3} - 1) = 0,779$, es decir, se verifica la Ecuación 1.1.

Sistema de tiempo real			
Tarea	WCET	Periodo / Plazo Finalización	Utilización
$Task_1$	5	20	0,25
$Task_2$	10	40	0,25
$Task_3$	20	80	0,25

Tabla 1.1: Ejemplo de un sistema *planificable*.

Supongamos ahora que el WCET de la tarea $Task_1$ es 7. En este caso, la utilización del procesador es $U = 0,85 > U(3) = 3 \cdot (2^{1/3} - 1) = 0,779$, luego

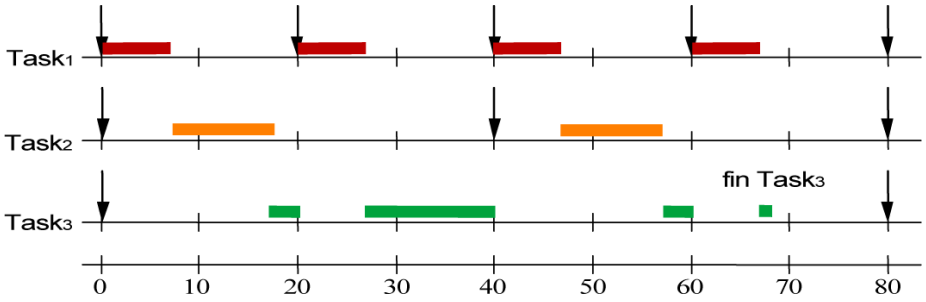


Figura 1.3: Planificación mediante RM de las nuevas tareas $Task_1$, $Task_2$ y $Task_3$.

no se verifica la Ecuación 1.1, por lo tanto no se puede asegurar que el sistema sea *planificable*. Sin embargo, esta condición es suficiente, pero no es necesaria para garantizar la *planificabilidad* del sistema. Es decir, en algunas ocasiones un sistema puede ser *planificable* sin verificar la expresión anterior, como por ejemplo en este caso. En la Figura 1.3 se muestra una ejecución del sistema planificando las tareas mediante RM (Rate Monotonic), y obviamente todas ellas acaban su ejecución antes de que termine su plazo de finalización.

Un sistema también es *planificable* si se puede garantizar, en cualquier caso, que el tiempo de respuesta de cada tarea $Task_i$ es menor que su plazo de finalización (RTA/ *Response Time Analysis*). Es decir una tarea $Task_i$ verifica sus restricciones temporales si $R_i \leq D_i$, siendo R_i su tiempo de respuesta y D_i su plazo de finalización. En este caso, el tiempo de respuesta R_i se determina mediante la expresión recursiva siguiente:

$$R_i^{n+1} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^n}{D_j} \right\rceil \cdot C_j \tag{1.2}$$

En la Figura 1.3 se observa que es posible planificar las tareas del ejemplo anterior. Pero para demostrar la *planificabilidad* del sistema debemos aplicar RTA. Para ello basta comprobar que el tiempo de respuesta de la tarea $Task_3$ es menor que su plazo de finalización, es decir $R_3 \leq D_3 = 80$. Aplicando la Ecuación 1.2 tenemos que:

$$\begin{aligned}
R_3^0 &= 0 \\
R_3^1 &= 20 \qquad\qquad (C_3 = 20) \\
R_3^2 &= \lceil 20/20 \rceil \cdot 7 + \lceil 20/40 \rceil \cdot 10 + 20 = 37 \\
R_3^3 &= \lceil 37/20 \rceil \cdot 7 + \lceil 37/40 \rceil \cdot 10 + 20 = 44 \\
R_3^4 &= \lceil 44/20 \rceil \cdot 7 + \lceil 44/40 \rceil \cdot 10 + 20 = 61 \\
R_3^5 &= \lceil 61/20 \rceil \cdot 7 + \lceil 61/40 \rceil \cdot 10 + 20 = 68 \\
R_3^6 &= \lceil 68/20 \rceil \cdot 7 + \lceil 68/40 \rceil \cdot 10 + 20 = 68
\end{aligned}$$

Así pues, como $R_3 = 68 \leq 80$, queda demostrado que el sistema es *planificable*.

Si por el contrario la prioridad de las tareas puede cambiar en función del estado del sistema, los algoritmos se denominan dinámicos o de prioridades dinámicas. Estos algoritmos basados en prioridades dinámicas presentan dos importantes ventajas. Por un lado aprovechan al máximo la potencia del procesador, haciendo que un conjunto de tareas sea *planificable* cuando no lo era utilizando un algoritmo de prioridades estáticas; y por otro, se adaptan perfectamente a entornos más dinámicos en los que la carga del sistema no puede ser conocida de antemano. Por ejemplo, el algoritmo EDF (*Earliest Deadline First*) asigna en cada instante la prioridad más alta a la tarea cuyo plazo de respuesta está más próximo; pero el plazo de ejecución de una tarea no es un valor constante y la prioridad de la misma va aumentando cuanto más cerca se encuentre de incumplir sus restricciones temporales [111]. El algoritmo LLF (*Least Laxity First*) asigna en cada instante la prioridad más alta a la tarea que menor holgura tiene para finalizar su ejecución. En este caso la prioridad de la tarea depende de su plazo de finalización y del tiempo de ejecución que todavía tiene pendiente [11].

Dificultades para calcular el WCET

La investigación sobre la *Jerarquía de Memoria* es uno de los más importantes y clásicos campos de la Arquitectura de Computadores. Las velocidades del procesador y de la memoria continúan creciendo a diferentes ritmos. Por otra parte, el desarrollo tecnológico permite integrar en un solo chip varios procesadores que pueden ejecutar uno o varios hilos de ejecución. La combinación de ambos factores obliga a realizar un sustancial rediseño de la jerarquía de memoria, para impedir que ésta llegue a ser un importante cuello de botella en los computadores del futuro. Los problemas relacionados con la creciente disparidad de velocidades entre procesador y memoria son objetivo de investigación desde todos los puntos de vista.

El mercado de los sistemas de tiempo real añade otra restricción en el diseño de la jerarquía de memoria, la necesidad de conocer un límite máximo del tiempo de ejecución, ya que, por ejemplo, este tiempo depende en gran medida del número máximo de fallos de cache que se producirán durante la ejecución. Aunque las memorias cache, tanto de instrucciones como de datos, reducen el tiempo medio de los accesos a la memoria principal y son muy utilizadas en los procesadores comerciales, la mayor parte de los diseñadores de sistemas de tiempo real descartan estos procesadores o proponen el apagado de las caches, debido a que el análisis de su comportamiento temporal, en el peor caso, es complejo. Pero las estimaciones pesimistas son poco prácticas, ya que el planificador asignará a cada tarea más tiempo del estrictamente necesario para su ejecución, y además gran parte de los recursos se desaprovechan, lo que genera

una gran desconfianza en el usuario. Por lo tanto es muy importante obtener valores seguros y precisos del WCET.

Los procesadores actuales disponen de una serie de componentes hardware, tales como la ejecución segmentada de instrucciones, los predictores de saltos, las memorias cache, etc., que permiten reducir la media del tiempo de ejecución de los programas [83]. Desafortunadamente estos componentes tienen latencia variable dependiente del pasado; así por ejemplo, la segmentación introduce los riesgos estructurales y de control que afectan al tiempo de ejecución de las instrucciones, el funcionamiento de los predictores de saltos depende de la historia local o global de ejecución, y las memorias cache pueden hacer aumentar el WCET en función del número de fallos. Todos estos componentes hardware hacen complejo el análisis del WCET y obligan a considerar la máxima latencia. Por lo tanto, el WCET es ampliamente sobrestimado forzando el incremento de los recursos disponibles del sistema para garantizar su funcionamiento temporal. El análisis temporal de los procesadores superescalares con ejecución fuera de orden donde los recursos del procesador se asignan dinámicamente, aún es más complejo, ya que podría alcanzar complejidad exponencial. Además, en este tipo de procesadores no es cierto suponer que la estimación del WCET es segura siempre que se asigna el tiempo de ejecución de peor caso a cada instrucción, debido a las *anomalías de distribución* (*timing anomalies*) [43, 115, 155, 193].

Una *anomalía de distribución* se produce cuando el peor caso local no está incluido en el peor caso global. Aunque es posible encontrar otros tipos de *anomalías de distribución*, los casos más significativos a tener en cuenta durante el análisis del WCET son los siguientes: las *anomalías de distribución* que se producen en la planificación o asignación de recursos, por ejemplo durante la asignación de las unidades funcionales del procesador; las generadas por la especulación, como por ejemplo las que producen los predictores de saltos; y las generadas por el funcionamiento de las memorias cache [155]. Pero, aunque las producidas en la asignación de recursos sólo aparecen en procesadores fuera de orden, las otras dos dependen del funcionamiento de los predictores de saltos y del comportamiento particular de las memorias cache.

En la Figura 1.4 se presentan dos casos típicos de anomalías de distribución. El caso a) muestra el efecto en la cache de un fallo durante la predicción de un salto. En este caso un fallo de cache en la instrucción A evita el fallo de predicción y mejora el tiempo de ejecución con respecto a un acierto de cache en dicha instrucción A. En el caso b) se muestra la planificación de un conjunto de instrucciones dependientes unas de otras. En este caso el tiempo de ejecución de la instrucción A varía. Curiosamente, el tiempo de ejecución de todas las instrucciones en conjunto es menor cuando el tiempo de ejecución de la instrucción A es mayor.

Aunque en la literatura se han propuesto diferentes técnicas para analizar el WCET de un programa, sólo el análisis estático permite determinar el WCET

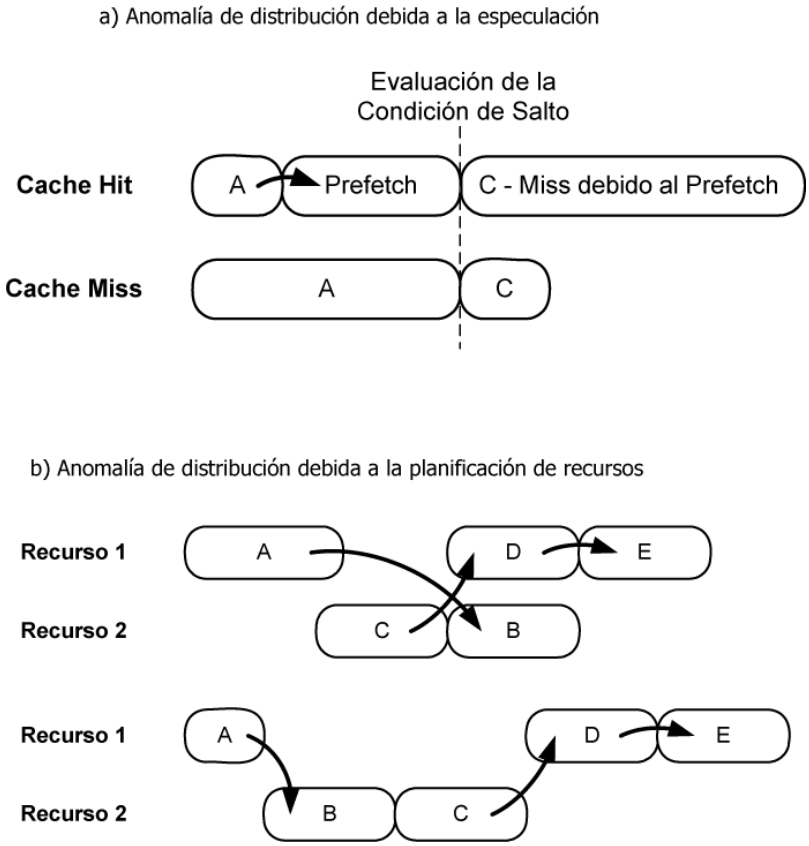


Figura 1.4: Anomalías de distribución [155].

de forma segura. Sin embargo, el hardware moderno, que cada vez es más complejo, supone una gran limitación para estas técnicas de análisis. Por ejemplo, el tiempo de ejecución de algunos segmentos del programa puede variar en función de si se utiliza hardware que actúa en paralelo o de si se emplean componentes que dependen de la historia de ejecución.

En definitiva, el cálculo del WCET es complejo, ya que depende tanto del software (por ejemplo el compilador, la arquitectura del lenguaje máquina, etc.), como del hardware (por ejemplo la jerarquía de memoria, los predictores de saltos, la ejecución segmentada de las instrucciones, etc.) [198]. Debido a la complejidad del hardware moderno, el tiempo de análisis del WCET podría exceder de lo razonable, obligando a sobrestimar el WCET de forma muy pesimista.

En general, sólo es posible obtener un WCET preciso si todos estos factores se consideran a la vez. En particular, se debe analizar el funcionamiento del procesador de la forma más exacta posible. El número de iteraciones de los bucles

y el número de llamadas recursivas han de estar acotados y las cotas tienen que ser conocidas. Por eso en la literatura, el término WCET hace referencia a una cota superior del tiempo de ejecución en el peor caso del programa, ya que se puede afirmar que analíticamente sólo es posible obtener cotas del tiempo de ejecución del programa. Por lo tanto, el WCET es la mínima cota superior obtenida durante el análisis.

Contribuciones de la Tesis

En esta Tesis se analiza el comportamiento, en el peor caso, de varias jerarquías de memoria para instrucciones. El principal objetivo de este análisis es conseguir el mejor rendimiento de la jerarquía de memoria estudiada en un sistema de tiempo real. Además se presentan diferentes técnicas de análisis y cálculo del WCET para cada una de las jerarquías de memoria analizadas. En concreto, para un conjunto de tareas que podrían formar un sistema de tiempo real, mediante las técnicas de análisis y cálculo que presentamos se consigue calcular un WCET totalmente seguro y más preciso que el obtenido con las técnicas de análisis ya descritas en la literatura.

A continuación comentamos brevemente las contribuciones más importantes de esta Tesis:

Las estructuras condicionales dentro de bucles hacen que el análisis del WCET en presencia de caches adquiera complejidad exponencial, debido a las interferencias intrínsecas de la cache. Como primera contribución demostramos que el número de caminos que es necesario analizar para determinar el comportamiento exacto de una cache de instrucciones con algoritmo de reemplazo LRU, no depende del número de iteraciones de los bucles, sino que está en función del número de caminos del condicional. Cuando el número de caminos alternativos de un bucle no es grande, la complejidad del problema se reduce considerablemente y en muchos casos se puede predecir de forma exacta el comportamiento de la cache de instrucciones en el peor caso [7]. Así pues, proponemos una técnica de poda que permite analizar y calcular el WCET de una tarea que se ejecuta de forma aislada en presencia de una cache de instrucciones. Este análisis, centrado en el comportamiento de una cache de instrucciones *convencional*, permite determinar la contribución exacta de los accesos a memoria al WCET. Por lo tanto, el WCET obtenido es más preciso [7].

En un sistema multitarea analizamos una cache de instrucciones que pueda fijar o bloquear su contenido durante algunos periodos de la ejecución. Como segunda contribución se presenta el algoritmo *Lock-MS* (Lock for Maximize Schedulability) para optimizar el rendimiento de una jerarquía de memoria formada por un LB (*Line Buffer*) y una cache de instrucciones que pueda fijar su contenido (*Lockable iCache*) durante algunos periodos de la ejecución del

sistema. Al fijar el contenido de la cache su comportamiento es totalmente predecible. Además, si el procesador considerado no dispone de otros componentes hardware de latencia variable, se evita la explosión combinatoria de los caminos condicionales dentro de bucles. Finalmente, también se evitan las interferencias de cache, tanto las intrínsecas, como las extrínsecas. El algoritmo *Lock-MS* está basado en ILP (*Integer Linear Programming*) y permite obtener un WCET seguro y preciso de cada una de las tareas del sistema. El objetivo de *Lock-MS* es seleccionar las líneas de memoria más adecuadas que se bloquearán en la cache, para obtener la máxima *planificabilidad* del sistema en esta jerarquía de memoria, teniendo en cuenta además el WCET de cada tarea y el coste de los cambios de contexto del sistema [6].

Sin embargo, cuando el número de caminos de un programa es grande, no es posible representar todos los caminos mediante restricciones lineales. Como tercera contribución se presenta un modelo compacto del algoritmo *Lock-MS* que permite reducir el número de caminos del problema ILP, sin perder precisión en el WCET obtenido [6].

Como cuarta contribución presentamos la posibilidad de predecir el WCET con un hardware de prebúsqueda secuencial (PB/ *Prefetch Buffer*). Así pues, se propone una nueva jerarquía de memoria con prebúsqueda para un sistema de tiempo real formada por un LB, un PB y una *Lockable iCache*. Para obtener la máxima *planificabilidad* del sistema, en esta nueva jerarquía de memoria, proponemos, tanto la extensión del algoritmo *Lock-MS*, como la extensión del modelo compacto de dicho algoritmo. En general el hardware de prebúsqueda permite reducir el WCET y la capacidad de la cache de instrucciones. En particular, la prebúsqueda reduce el WCET en programas de código *plano*, llegando a obtener un rendimiento equivalente al caso ideal [5].

Finalmente, y dado que el consumo de energía también es un aspecto importante en los sistemas de tiempo real, se presenta un estudio sobre el consumo energético en el peor caso (WCEC/ *Worst Case Energy Consumption*). En este estudio se pone de manifiesto que no existe una correspondencia lineal entre el WCEC y el WCET de una tarea. Así pues, se introduce la posibilidad de que el diseñador del sistema decida si su objetivo es obtener un WCET más preciso o reducir el WCEC. También se muestra que la prebúsqueda aumenta considerablemente el consumo de energía del sistema y, en algunos casos, no consigue reducir significativamente el WCET de las tareas analizadas.

Actualmente, estamos analizando el comportamiento en el peor caso de la cache de datos, ya que podría reducir aún más el WCET de una tarea. Sin embargo, predecir el funcionamiento de la cache de datos es un problema bien distinto, ya que para una misma instrucción de acceso a datos, las direcciones de memoria a las que se accede pueden cambiar a lo largo de la ejecución. En concreto, estamos estudiando el comportamiento de una nueva estructura hardware predecible para la cache de datos en sistemas de tiempo real. Como

primera solución al problema ya hemos propuesto una estructura denominada ACDC (*Address-Cache/Data-Cache*). Esta nueva estructura está formada por una pequeña cache de datos (DC/ *Data-Cache*) y por una tabla (AC/ *Address-Cache*) que guarda las direcciones de las instrucciones que pueden actualizar el contenido de la DC [162]. Por lo tanto, nuestro trabajo futuro, en sistemas de tiempo real, se centrará en la predicción de los accesos a datos para explotar la localidad espacial y temporal en este tipo de accesos.

Organización de la memoria de la Tesis

En el Capítulo 2 se presenta el problema del análisis y cálculo del WCET y se revisan muchos de los trabajos de investigación más relevantes relacionados con este problema.

En el Capítulo 3 se estudia el comportamiento en el peor caso de una cache de instrucciones *convencional*. En este capítulo se demuestra que, en presencia de una cache de instrucciones con algoritmo de reemplazo LRU, se puede obtener la contribución exacta al WCET de los accesos a memoria, analizando los *caminos relevantes* de un programa.

En el Capítulo 4, se estudia el comportamiento en el peor caso de una jerarquía de memoria formada por una LB y una *Lockable iCache* en un sistema de tiempo real. También se propone el algoritmo *Lock-MS* para seleccionar las líneas de memoria a fijar en la cache y obtener la máxima *planificabilidad* del sistema.

En el Capítulo 5 se presenta una nueva jerarquía de memoria para instrucciones en sistemas de tiempo real. Esta jerarquía de memoria dispone de una LB, un PB y una *Lockable iCache*. Además, para obtener la máxima *planificabilidad* del sistema, se extiende el algoritmo *Lock-MS* a esta nueva jerarquía de memoria. Finalmente, en este capítulo se realiza un estudio sobre el consumo energético de esta jerarquía de memoria.

En el Capítulo 6 se resumen las conclusiones de esta Tesis y se comentan los objetivos actuales y futuros de la líneas de investigación abiertas.

Capítulo 2

Análisis y cálculo del WCET

Una gran cantidad de trabajos de investigación se han presentado para intentar resolver el problema del análisis y cálculo del WCET de un programa, desde que en 1989 se propusiera el primer método concreto para este fin [149]. Debido a la dificultad que entraña esta cuestión, los diferentes trabajos propuestos se centran en algunos problemas concretos del análisis. Generalmente en la literatura se describen dos técnicas para analizar el WCET de un programa: el análisis basado en medida y el análisis estático [198].

En la Figura 2.1 se muestran los principales esquemas de trabajo para analizar y calcular el WCET de un programa. Tanto el análisis basado en medida, como el análisis estático, suelen dividir el programa en segmentos o bloques básicos y tratan de determinar el tiempo de ejecución de cada uno de ellos. Un bloque básico es un conjunto de instrucciones que se ejecutan de forma secuencial de principio a fin, es decir, ninguna de las instrucciones del conjunto puede ser un salto excepto la última instrucción. A partir del tiempo de ejecución obtenido en la fase de análisis y mediante algún método de cálculo o expresión matemática se determina el WCET del programa.

El análisis de flujo de control, también denominado *análisis de alto nivel*, permite obtener información sobre la ejecución del programa en tiempo de compilación, como por ejemplo el número máximo de iteraciones de los bucles, los caminos imposibles o el valor de algunas variables en puntos específicos de la ejecución. Cuanto más exacta sea la información recopilada durante este análisis, más precisa será la cota del WCET calculada.

El análisis basado en medida (MBA/ *Measurement-Based Analysis*) determina el tiempo de ejecución del programa en un hardware específico, para un conjunto de entradas concreto. En este tipo de análisis, el modelado del hard-

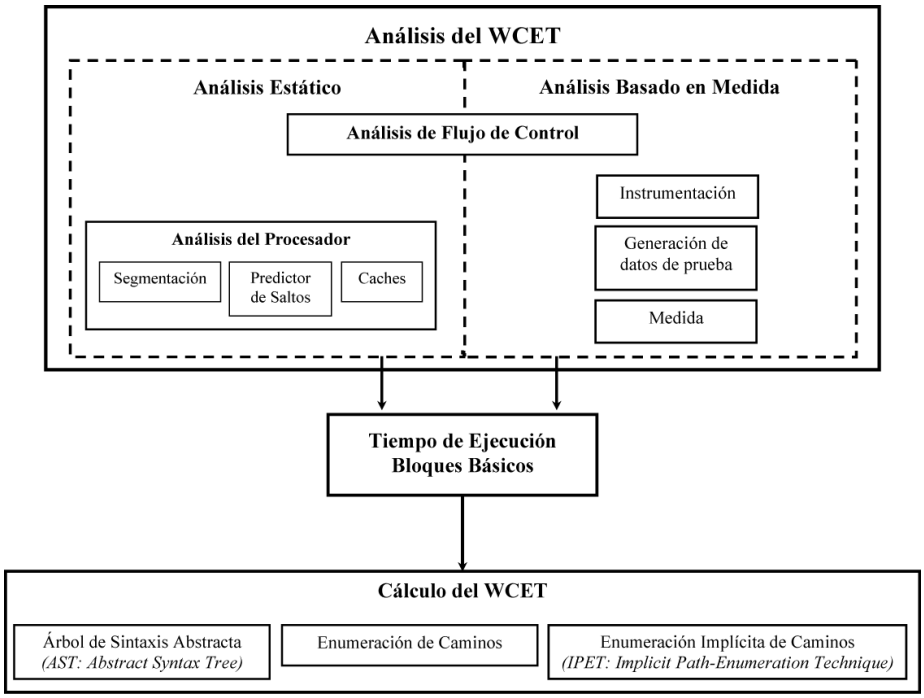


Figura 2.1: Elementos que intervienen en el análisis y cálculo del WCET.

ware se sustituye por la instrumentación del código, por la generación de los datos de prueba, y por la medida real del tiempo de ejecución de los trozos o segmentos del programa cuando se ejecutan con los datos de prueba generados previamente. Aunque el análisis basado en medida no garantiza que el WCET conseguido sea seguro, ya que es imposible medir el tiempo de ejecución de todos los caminos, sí evita tener que realizar un análisis exhaustivo del comportamiento del procesador. A su vez, permite obtener el WCET analizando por separado los diferentes segmentos en los que se puede dividir el programa durante el análisis. Este método es muy utilizado en la industria, ya que permite verificar el funcionamiento y determinar una aproximación al WCET del programa.

El análisis estático proporciona un método totalmente seguro para determinar el WCET del programa. En este caso, la precisión del WCET depende en gran medida de la exactitud del análisis de flujo de control y del análisis del funcionamiento del procesador o modelado hardware, también denominado *análisis de bajo nivel*. El análisis del comportamiento del procesador permite determinar el WCET de los bloques básicos, considerando características del hardware subyacente tales como la ejecución segmentada de las instrucciones, los predictores de saltos y las memorias cache. Sin embargo, el tiempo de análisis podría exceder de lo razonable obligando a sobrestimar el WCET de forma pesimista.

Finalmente, el tiempo de ejecución de cada bloque básico del programa se introduce en una herramienta o motor de cálculo, para obtener el WCET del programa. En definitiva, el WCET se determina a partir de la información obtenida durante el análisis estático o basado en medida, aplicando posteriormente alguna técnica de cálculo como las indicadas en la Figura 2.1 y que se describen en la siguiente sección.

2.1. Cálculo del WCET

En la literatura se describen tres técnicas de cálculo que permiten determinar el WCET de un programa: el cálculo basado en el árbol de sintaxis abstracta [23, 38], la enumeración de todos los posibles caminos de ejecución [79, 168], y la enumeración implícita de caminos [105, 152].

Habitualmente, la expresión matemática que determina el WCET está basada en el grafo de flujo control (CFG/ *Control Flow Graph*) o en el árbol de sintaxis abstracta (AST/ *Abstract Syntax Tree*) del programa. El WCET se calcula a partir de las restricciones de flujo y de los tiempos de ejecución de los trozos o segmentos en los que se haya dividido el programa [130]. En la parte superior de la Figura 2.2 se muestra el CFG asociado a un programa, junto con los tiempos de ejecución de sus bloques básicos, y también se presenta el AST asociado a dicho programa. En la parte inferior de la Figura 2.2 se indican el *camino más largo* en el CFG del programa y el CFG con las restricciones de conservación de flujo, que se utilizan para calcular el WCET del programa mediante la enumeración implícita de caminos (IPET/ *Implicit Path-Enumeration Technique*).

Cálculo del WCET basado en el AST

El cálculo del WCET basado en el árbol de sintaxis abstracta es muy rápido, pero carece de una visión global que es necesaria para que el WCET obtenido sea preciso, por lo tanto la cota del WCET conseguida suele ser pesimista. Como se muestra en la Figura 2.3, el cálculo basado en árbol utiliza un formato del AST con información del tiempo de ejecución de los segmentos o bloques básicos para representar la estructura temporal del programa [149, 151].

Para determinar el tiempo de ejecución de un segmento del programa, en función del tiempo de ejecución de sus bloques básicos $Tiempo_{Exe}(A)$, se utilizan las siguientes reglas:

- El tiempo de ejecución de una secuencia de bloques básicos de un programa es la suma del tiempo de ejecución de cada uno de ellos. Por ejemplo, si A y B son bloques en secuencia:

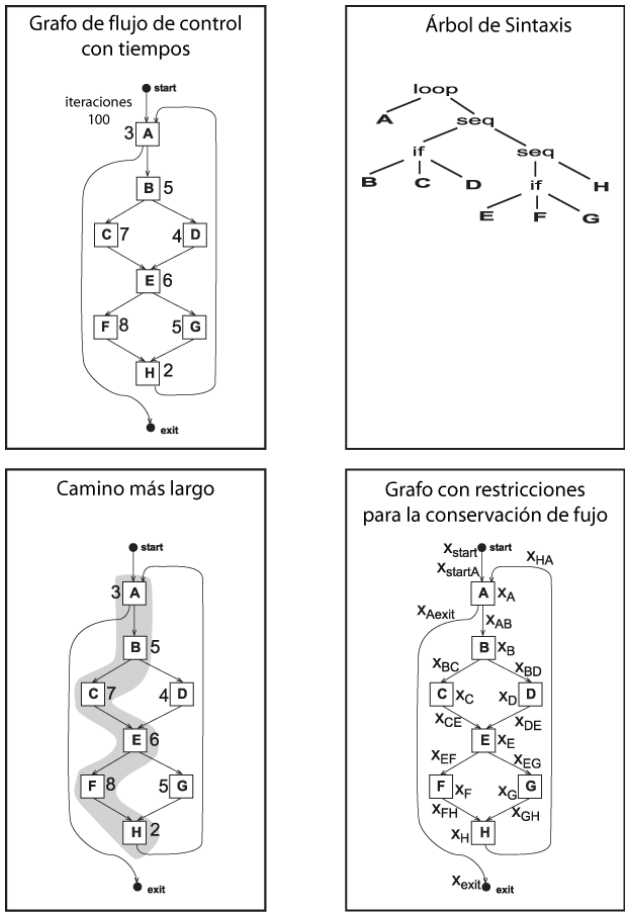


Figura 2.2: Grafo de flujo de control, árbol de sintaxis abstracta, *camino más largo* y grafo de flujo de control con restricciones [198].

$$Tiempo_{Exe}(A, B) = Tiempo_{Exe}(A) + Tiempo_{Exe}(B)$$

- El tiempo de ejecución de un bucle es la suma del tiempo de ejecución del cuerpo, más el tiempo de ejecución de la condición o guarda del bucle, multiplicado por el número máximo de iteraciones. Por ejemplo, en el caso de un bucle donde N es el número máximo de iteraciones:

$$Tiempo_{Exe}(for \;\; E \;\; loop \;\; A) =$$

$$Tiempo_{Exe}(E) + N \cdot (\; Tiempo_{Exe}(A) + Tiempo_{Exe}(E) \;)$$

- El tiempo de ejecución de una estructura condicional es la suma del tiempo de ejecución de la condición, más el máximo de los tiempos de ejecución

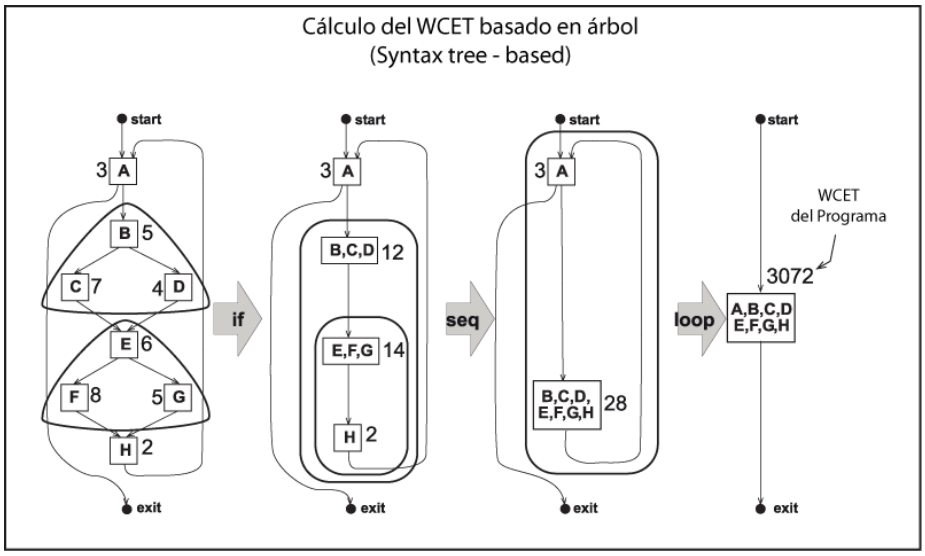


Figura 2.3: Esquema que sigue el cálculo del WCET basado en AST [198].

de cada una de las alternativas del condicional. Por ejemplo, en el caso de un condicional:

$$Tiempo_{Exe}(if \;\; E \;\; then \;\; A \;\; else \;\; B) =$$

$$Tiempo_{Exe}(E) + \text{máx}\,(\; Tiempo_{Exe}(A), Tiempo_{Exe}(B)\;)$$

Por lo tanto, aplicando las reglas anteriores, como se indica en la Figura 2.3, al esquema del programa mostrado en la Figura 2.2 se tiene que:

$$WCET \;\; = \;\; 3072$$

En general, los métodos basados en árbol no pueden capturar las restricciones de flujo de control complejas o los tiempos de ejecución variables para un mismo segmento o bloque básico. El cálculo basado en árbol es local, es decir, el tiempo de ejecución de los bloques básicos se obtiene de forma independiente y luego se utiliza en las estructuras de programación de las que forman parte dichos bloques.

Cálculo del WCET basado en caminos

En los métodos de cálculo del WCET basados en caminos se enumeran todos los caminos que puede seguir el programa durante su ejecución. El WCET se calcula como el máximo tiempo de ejecución asociado a dichos caminos. Conviene

indicar que un camino es una secuencia de segmentos o bloques básicos que no contiene estructuras condicionales ni bucles. Esta técnica puede alcanzar complejidad exponencial cuando se aplica a programas con sentencias condicionales dentro de bucles, en especial en presencia de componentes hardware con latencia variable durante la ejecución, como por ejemplo los predictores de saltos y las memorias cache. En estos casos, esta técnica sólo se puede utilizar para determinar el tiempo de ejecución de un trozo o segmento del programa, pero el WCET del programa completo se podría sobrestimar ampliamente debido a la pérdida de información que se produce al evitar la complejidad exponencial que presenta.

Considerando el ejemplo de la Figura 2.2 donde se indica el *camino más largo* de un programa, el WCET basado en caminos se determina a partir de los siguientes cálculos:

Enumeración de caminos

$$
\begin{aligned}
path1 &: A + B + C + E + F + H \\
path1 &: A + B + C + E + G + H \\
path2 &: A + B + D + E + F + H \\
path3 &: A + B + D + E + G + H
\end{aligned}
$$

Unidades de tiempo de programa

$$
\begin{aligned}
Tiempo_{path1} &= 31 \\
Tiempo_{path2} &= 28 \\
Tiempo_{path3} &= 28 \\
Tiempo_{path4} &= 25 \\
Tiempo_{header} &= 3
\end{aligned}
$$

Cálculo del WCET

$$
\begin{aligned}
WCET &= Tiempo_{header} + Tiempo_{path1} \cdot (Iteraciones - 1) \\
WCET &= 3 + 31 \cdot 99 = 3072
\end{aligned}
$$

Cálculo del WCET basado en IPET

En los métodos de cálculo del WCET basados en la enumeración implícita de caminos (IPET/ *Implicit Path-Enumeration Technique*), el flujo de control del programa y el tiempo de ejecución de los bloques básicos se transforman en un problema de Programación Lineal Entera (ILP/ *Integer Linear Programming*) [105, 152]. El método IPET permite expresar mediante restricciones lineales, tanto las dependencias temporales, como las del flujo de control, y con una herramienta de cálculo adecuada o *solver* se resuelve el problema ILP de

forma muy efectiva [161].

El cálculo del WCET basado en IPET es más complejo, y en él se deben considerar las restricciones de conservación de flujo asociadas al CFG. Así pues, el número de veces que se ejecutará cada bloque básico debe cumplir las reglas de conservación de flujo, garantizándose que el número de veces que se ejecutan los bloques de entrada es igual al número de veces que se ejecutan los bloques de salida. A estos enlaces de los bloques básicos se les denomina restricciones estructurales y forman parte de las restricciones de flujo de control del programa.

Por ejemplo, si consideramos el CFG con restricciones de conservación de flujo de la Figura 2.2, las restricciones que modelan el problema ILP son las siguientes:

Restricciones de Inicio y Finalización

$$\begin{aligned} X_{start} &= 1 \\ X_{exit} &= 1 \end{aligned}$$

Restricciones estructurales

$$\begin{aligned} X_{start} &= X_{statA} \\ X_A &= X_{startA} + X_{HA} = X_{Aexit} + X_{AB} \\ X_B &= X_{AB} = X_{BC} + X_{BD} \\ X_C &= X_{BC} = X_{CE} \\ \dots &\quad \dots \\ X_H &= X_{FH} + X_{GH} = X_{HA} \\ X_{exit} &= X_{Aexit} \end{aligned}$$

Restricciones asociadas a los límites de los bucles

$$X_A \leq 100$$

Expresión que determina el WCET del programa

$$\begin{aligned} WCET &= \text{máx}\,(3 \cdot X_A + 5 \cdot X_B + 7 \cdot X_C \dots 2 \cdot X_H) \\ WCET &= 3072 \end{aligned}$$

Los datos necesarios para determinar un WCET seguro y preciso mediante IPET son las restricciones estructurales y el tiempo de ejecución de cada uno de los bloques básicos del programa. Además, teniendo en cuenta las condiciones de flujo de control, se determinan las *restricciones de funcionalidad* del programa. En muchas ocasiones, estas restricciones se obtienen de forma automática, otras veces el usuario las añade directamente al problema ILP. Las restricciones de

funcionalidad permiten estimar de una forma más precisa el WCET. Así pues, el cálculo del WCET se modela como un conjunto de restricciones lineales cuya función objetivo maximiza el tiempo de ejecución del programa. El cálculo el WCET también se podría formular como un problema de programación lineal (LP/ *Linear Programming*) pero, en algunas ocasiones, la solución no sería válida, ya que en este caso la solución no tiene por qué ser entera. Sin embargo, el WCET obtenido al resolver el problema LP suele ser una buena aproximación al WCET exacto, proporcionando una cota inferior.

La ejecución de programas complejos en procesadores actuales no se puede modelar de una forma sencilla, como un simple problema de conservación de flujo. Así por ejemplo, para considerar las funcionalidades del programa es necesario añadir restricciones de flujo complejas. Pero si además el tiempo de ejecución de los bloques básicos puede ser variable, por ejemplo debido a la utilización de memorias cache, se deben añadir nuevas restricciones relativas a la historia de ejecución del programa.

Así pues, aunque los métodos de cálculo del WCET basados en IPET pueden tener algunas limitaciones a la hora de describir el problema ILP, el WCET obtenido es seguro y suele ser mucho más preciso que el logrado mediante AST o mediante la enumeración de caminos, ya que las limitaciones de estos métodos todavía son mayores.

Cálculo del WCET: Un ejemplo basado en ARM v7

Como ejemplo de los comentarios anteriores, en este apartado se calcula el WCET de un programa escrito en C. En la Figura 2.4 se muestra el código ensamblador del programa compilado con GCC 2.95.2 -O2 para ARM v7. En el código ensamblador se han marcado los bloques básicos del programa y su coste de ejecución. Suponemos, por sencillez, que cada instrucción tiene un coste de ejecución de un ciclo, por lo tanto los tiempos de ejecución de cada bloque básico son constantes para todos los posibles caminos de ejecución.

Asociado al programa de la Figura 2.4, se muestra en la Figura 2.5 el árbol de sintaxis abstracta, el grafo de flujo de control con el *camino más largo* marcado y el grafo de flujo de control con las restricciones de conservación de flujo necesarias.

El WCET del programa de la Figura 2.4, que se obtiene a partir del AST de la Figura 2.5 a), se determina en función del tiempo de ejecución de cada bloque básico mediante la siguiente expresión:

$$\begin{aligned} WCET &= B1 + 10 \cdot (\ B2 + \text{máx}\,(B3, B4) + B5\ ) + B6 \\ WCET &= 8 + 10 \cdot (\ 4 + \text{máx}\,(7, 2) + 7\ ) + 1 = 189 \end{aligned}$$

```
                    @ Generated by gcc 2.95.2 19991024 (release) for ARM/elf
                            .file   "programa.c"
                    gcc2_compiled.:
                            .global n
                    .data
                            .align  2
                            .type    n,object
                            .size    n,4
                    n:
                            .word   10
                            .global z
                            .align  2
                            .type    z,object
                            .size    z,4
                    z:
                            .word   0
                    .text
                            .align  2
                            .global main
                            .type    main,function
                    main:
                            @ args = 0, pretend = 0, frame = 0
                            @ frame_needed = 1, current_function_anonymous_args = 0
 _______
|-> B1: 8 ciclos            mov     ip, sp
|                           stmfd   sp!, {fp, ip, lr, pc}
|                           sub     fp, ip, #4
|                           mov     r1, #0
|                           mov     lr, r1
|                           mov     r2, r1
|                           mov     r0, r2
|_______                    ldr     ip, .L10
 _______                .L6:
|-> B2: 4 ciclos            add     r3, r2, #1
|                           cmp     r3, #5
|                           mov     r2, r3
|_______                    bgt     .L7
|-> B3: 7 ciclos            add     lr, lr, #1
|                           add     r1, r1, #2
|                           ldr     r3, [ip, #0]
|                           add     r0, r0, #3
|                           add     r3, r3, #4
|                           str     r3, [ip, #0]
|_______                    b       .L8
 _______                .L7:
|-> B4: 2 ciclos            mov     r1, r1, asl #1
|_______                    mov     r0, r0, asl #2
 _______                .L8:
|-> B5: 7 ciclos            ldr     r3, [ip, #0]
|                           cmp     r2, #9
|                           add     r3, r3, lr
|                           add     r3, r3, r1
|                           add     r3, r3, r0
|                           str     r3, [ip, #0]
|_______                    ble     .L6
|-> B6: 1 ciclo             ldmea   fp, {fp, sp, pc}
                        .L11:
                            .align  2
                        .L10:
                            .word   z
                        .Lfe1:
                            .size    main,.Lfe1-main
                            .ident  "GCC: (GNU) 2.95.2 19991024 (release)"
```

```
/*  programa.c  */

int n = 10;
int z = 0;

int main()
{
   int i, j;
   int v, x, y;

   v = 0;
   x = 0;
   y = 0;

  for (i = 0; i < 10; i++)
  {
     j = i + 1;
     if (j <= 5)
     {
        v = v + 1;
        x = x + 2;
        y = y + 3;
        z = z + 4;
     }
     else
     {
        x = x * 2;
        y = y * 4;
     }
     z = z + v + x + y;
  }
}
```

Figura 2.4: Código ensamblador ARM y bloques básicos asociados a un programa en C.

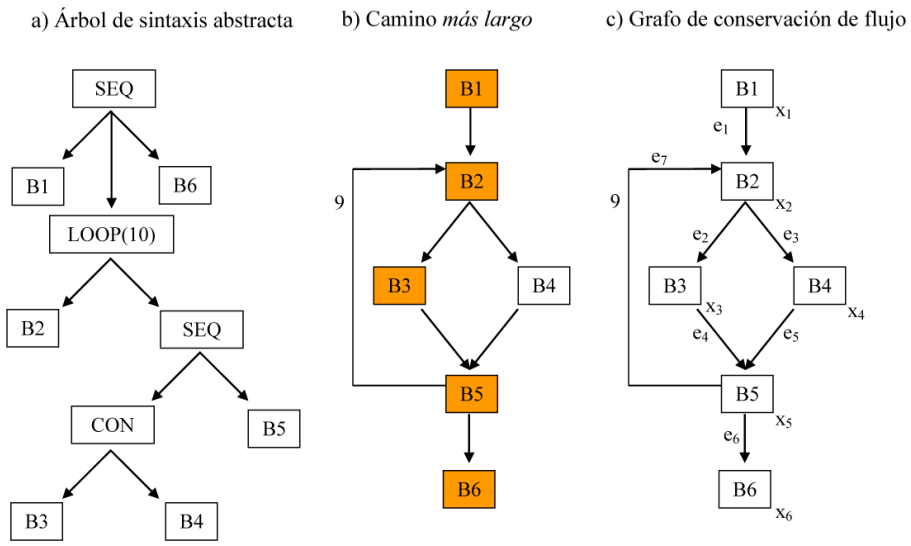


Figura 2.5: Árbol de sintaxis abstracta, grafo de flujo de control y grafo de flujo con restricciones del programa de la Figura 2.4.

El programa de la Figura 2.4 tiene 2^{10} posibles caminos de ejecución que deben ser explorados para determinar el WCET. Sin embargo, cuando el tiempo de ejecución de cada bloque básico es fijo, se puede simplificar calculando los máximos locales a los diferentes *subcaminos* condicionales que presenta el programa. Así pues, el cálculo del WCET del programa basado en la enumeración de caminos se puede simplificar de la siguiente manera:

$$\begin{aligned} WCET &= B1 + 10 \cdot (\ B2 + \text{máx}\,(subPath_{B3},\ subPath_{B4}) + B5\ ) + B6 \\ WCET &= 8 + 10 \cdot (\ 4 + \text{máx}\,(7, 2) + 7\ ) + 1 \\ WCET &= 8 + 10 \cdot (\ 4 + 7 + 7\ ) + 1 = 189 \end{aligned}$$

Finalmente, en la Tabla 2.1 se muestran las restricciones estructurales y de funcionalidad asociadas al grafo de flujo de control con la información de conservación de flujo de la Figura 2.5 c). También se indica el tiempo de ejecución de los bloques básicos del programa de la Figura 2.4. A partir de esta información se modela el problema ILP que determina el WCET del programa mediante IPET. En este caso los tiempos de ejecución de cada bloque básico son constantes y las variables del problema ILP son el número de veces que se ejecuta cada uno de los bloques básicos.

El tiempo de ejecución del *camino más largo* del programa se obtiene resol-

Restricciones estructurales	Restricciones funcionales	Ciclos de ejecución de bloques básicos
$x_1 = 1$	$x_3 \leq \quad 5$	$B1 = 8$
$x_1 = e_1$	$x_5 \leq 10$	$B2 = 4$
$x_2 = e_1 + e_6$		$B3 = 7$
$x_2 = e_2 + e_3$		$B4 = 2$
$x_3 = e_2$		$B5 = 7$
$x_3 = e_4$		$B6 = 1$
$x_4 = e_3$		
$x_4 = e_5$		
$x_5 = e_4 + e_3$		
$x_5 = e_6 + e_7$		
$x_6 = e_7$		
$x_6 = 1$		

Tabla 2.1: Restricciones IPET y tiempo de ejecución de los bloques básicos del programa de la Figura 2.4.

viendo el siguiente problema ILP.

$$\text{máx} : \sum_{k=1}^{6} Bk \cdot x_i$$

Cálculo del WCET basado en parámetros

El cálculo del WCET basado en parámetros es otra técnica que proporciona una estimación precisa del tiempo de ejecución en el peor caso. Se trata de evaluar una expresión simbólica que depende de una serie de variables asociadas al programa. Mediante este cálculo no se proporciona una cota fija del WCET, sino que se obtiene una función dependiente de una serie de parámetros asociados al programa. Cuando se asigne un valor a cada uno de los parámetros y se evalúe la función, se obtendrá una cota del WCET.

Existen diferentes razones para justificar este tipo de cálculo. En algunos casos, el valor de ciertos parámetros sólo es conocido en tiempo de ejecución, por ejemplo el número máximo de iteraciones de un bucle para una ejecución particular del programa. En otros casos, los datos de entrada del programa determinan el camino a seguir durante la ejecución, y por lo tanto pueden fijar el tiempo de ejecución de algunas subrutinas. Por último, se justifica el uso de este tipo de cálculo, por la rapidez con la que se obtiene el WCET del programa cuando ya se han asignado los valores a todos los parámetros.

Como ya se ha comentado, para conseguir una cota precisa del WCET de un programa es necesario indicar el número máximo de iteraciones de cada uno de

sus bucles. Si esta información no se conoce de forma exacta, el análisis estático no es efectivo y puede llegar a producir una sobrestimación inadmisible en el WCET calculado. En algunos trabajos se propone el análisis del WCET basado en parámetros para evitar esta sobrestimación [37, 184]. En este caso, la función obtenida en el análisis depende del número de iteraciones que se decide durante la ejecución. Los beneficios de este tipo de análisis son claros, ya que el WCET se calcula de una forma muy rápida, se mejora la planificación dinámica del sistema y se gestiona de forma más efectiva la utilización de los recursos del sistema. Por otra parte, cuando la función que determina el WCET depende de los datos de entrada del programa o del tiempo de ejecución de alguna función, el WCET conseguido es más sensible al contexto. Puesto que el tiempo de ejecución de la función no es constante, la estimación del WCET se puede obtener evaluando dicha función para los valores extremos [19].

Finalmente, dado el interés suscitado por esta técnica, en algunos trabajos de investigación incluso se ha propuesto utilizar Programación Paramétrica Entera (PIP/ *Parametric Integer Programming*) [52]. Se trata de transformar el cálculo del WCET basado en IPET, que se soluciona mediante un problema ILP, en un problema basado en parámetros que se resuelve mediante PIP [4, 33, 110].

2.2. Análisis de flujo de control

El análisis de flujo de control es una de las fases que más influencia tiene en el análisis y cálculo del WCET. Durante esta fase se intenta obtener la mayor información sobre la estructura del programa, generalmente, a partir del código fuente. Posteriormente es necesario combinar la información obtenida durante el análisis de flujo de control y el análisis del funcionamiento del procesador, para lograr una estimación del WCET del programa lo más precisa posible.

El principal objetivo del análisis de flujo de control es determinar la estructura del programa mediante el árbol de sintaxis abstracta (AST/ *Abstract Sysntax Tree*) o mediante el grafo de flujo de control (CFG/ *Control Flow Graph*). En algunos casos, el compilador puede generar la estructura del programa automáticamente, por lo que, tanto la construcción del AST, como la del CFG suelen ser directas. Además, durante el análisis de flujo de control del programa también es de gran importancia determinar el número máximo de iteraciones de los bucles y detectar los *caminos imposibles*. En este ámbito de investigación, los *caminos imposibles* son aquellos caminos que nunca pueden ser recorridos durante la ejecución del programa.

Análisis de la estructura del programa

El análisis de flujo de control es complejo, ya que depende del tamaño del código, de los valores de los datos de entrada e incluso del tamaño del domi-

nio de las variables del programa. No obstante, si queremos realizar un análisis exhaustivo de la estructura del programa, son también de interés las técnicas que intentan reducir la complejidad del análisis a costa de perder cierta información [38, 73, 159]. Así, por ejemplo, estas técnicas intentan fusionar caminos o eliminar las instrucciones que no influyen directamente en el flujo de control del programa.

Para determinar la estructura del programa se utiliza, tanto el código fuente, como el código objeto. Si la información de flujo de control se obtiene a partir del código fuente es necesario trazar un mapa de la estructura del programa sobre el código objeto, aunque en la mayor parte de los casos, tanto el AST, como el CFG se suelen generar de forma directa a partir del código fuente.

Cuando el compilador no realiza optimizaciones destructivas es relativamente sencillo trazar este mapa sobre el código objeto, aunque no es suficiente para obtener un WCET preciso. En la literatura se han presentado diferentes técnicas que consiguen la información más relevante del programa de forma automática para calcular el WCET. Estas técnicas analizan el código objeto del programa, y con la información obtenida completan el AST o el CFG. La mayor parte de estas técnicas utilizan ejecución simbólica, para analizar el código objeto del programa o transformar la información conseguida durante el análisis en restricciones lineales que luego incorporan al cálculo del WCET [34, 48, 49, 50, 168].

Pero cuando el compilador puede aplicar optimizaciones destructivas, trazar un mapa de la estructura del programa sobre el código objeto es mucho más difícil. De hecho, en muchos trabajos de investigación se prohiben este tipo de optimizaciones para evitar este problema. Puesto que la estructura del código objeto compilado con optimizaciones puede ser muy diferente a la estructura del código fuente, establecer la relación entre el código objeto y el código fuente es más complejo, y por lo tanto el análisis del WCET también lo es.

Cuando la información de flujo de control se obtiene a partir del código objeto, ya no es necesario trazar un mapa de la estructura del programa sobre el código objeto. Además, se aprovechan todas las optimizaciones del compilador que, en general, mejoran el funcionamiento del programa y reducen el tamaño del código. Ambos factores disminuyen el coste del sistema y, no sólo mantienen las prestaciones del mismo, sino que las incrementan. Por lo tanto, a la hora de calcular el WCET de un programa no se deberían desaprovechar las optimizaciones del compilador, ya que la cota del WCET obtenida será mucho más precisa.

En la literatura se han propuesto diferentes trabajos de investigación dedicados a resolver este problema [44, 46, 93, 94, 95, 96, 109, 148, 185]. En general todos estos trabajos tratan de añadir, en el código objeto, la información que se obtiene del código fuente, de tal forma que se pueda aprovechar el trabajo realizado durante el proceso de compilación.

La programación orientada a objetos añade aún más complejidad al análisis de flujo de control. Obviamente con este tipo de programación es difícil establecer una relación entre el código fuente y el código objeto del programa. Este problema ha sido tratado mediante interpretación abstracta en diferentes trabajos de investigación [67, 68, 69, 70, 71, 72].

Iteraciones de un bucle y caminos imposibles

El análisis de flujo de control también trata de determinar el número máximo de iteraciones de los bucles y los *caminos imposibles*, por ejemplo, mediante alguna técnica de análisis formal como la interpretación abstracta [40].

En principio es habitual que el programador indique en el código fuente el número de iteraciones de los bucles, pero estas indicaciones pueden llevar asociado algún tipo de error. Además, cuando se permite al compilador realizar optimizaciones, estas indicaciones pueden no ser exactas, por ejemplo, cuando el compilador desenrolla un bucle, el número de iteraciones se reduce. Por lo tanto, también son importantes los trabajos de investigación que determinan automáticamente el número de iteraciones de los bucles [41, 51, 75, 76, 77, 80, 81, 84, 89, 112, 126]. La mayor parte de estas propuestas tratan de determinar directamente los invariantes de los bucles, por ejemplo mediante la interpretación abstracta [51, 126]. En otros casos particulares se ha propuesto el análisis de flujo de datos [41], o el análisis sintáctico [76, 77]. Aunque también es habitual utilizar ejecución simbólica [80, 81, 84, 89].

Un *camino imposible* puede hacer que la cota obtenida durante el cálculo del WCET no sea precisa, ya que durante el análisis estático dicho camino se podría considerar como el *camino más largo*. Dicho camino nunca será tomado durante la ejecución del programa y, por lo tanto, el WCET obtenido, aunque seguro, no será preciso. En la literatura también se han presentado soluciones para detectar los *caminos imposibles* o *falsos caminos*, y evitar una posible sobrestimación del WCET [2, 3, 35, 57, 73, 74, 75, 89, 99, 172]. Estos trabajos utilizan ejecución simbólica, ejecución abstracta, o programación lineal con restricciones.

El ejemplo de la Figura 2.6 muestra distintos tipos de caminos imposibles y dos bucles anidados cuyos límites dependen de un parámetro [75]. El programa contiene dos funciones, la función *foo* que tiene que estudiarse dos veces dependiendo del punto del programa desde donde se invoca, y la función *bar* que tiene dos bucles anidados dependientes de un parámetro. La función *main* puede seguir algunas de las siguientes combinaciones de caminos: $path_A$ o $path_B$, $path_C$ o $path_D$ y $path_E$ o $path_F$. La funcion *foo* puede seguir los caminos: $path_G$ o $path_H$ y $path_I$ o $path_J$. Analizando este código mediante ejecución abstracta, se pueden detectar los caminos imposibles y determinar el número máximo de iteraciones del bucle [75].

```
// x = [ 0 . . 100 ]

int main ( int x )
{
  if ( x < 10 )  A
  else  B

  if ( x < 5 )  C
  else  D

  foo ( x );

  if ( x < 0 )  E
  else  F

  foo ( x + 50 );

  bar ( x );

 return 1;
}
```

```
void foo ( int y )
{
  int i;

  for ( i = 0; i < 10; i++ )
  {
    if ( y > 50 )  G
    else  H

    if ( y < 50 )  I
    else  J
  }

void bar ( int n )
{
  int i, j;
  for ( i = 0; i < n; i++ )
    for ( j = i; j < n; j++ )
      K
}
```

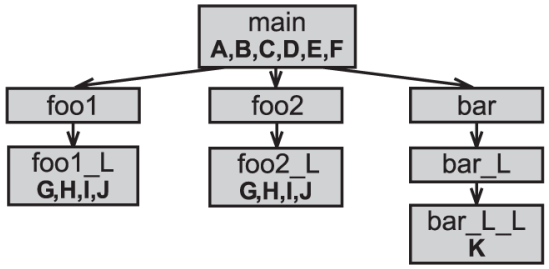


Figura 2.6: Programa con diversos *caminos imposibles* [75].

Por ejemplo, en la función *main*, estudiando los condicionales excluyentes, se observa que las siguientes combinaciones de caminos son imposibles: $path_B$ y $path_C$; $path_B$ y $path_E$; $path_D$ y $path_E$; $path_G$ y $path_I$; y también $path_H$ y $path_J$. También los siguientes tríos: $path_A$, $path_D$ y $path_E$; $path_B$, $path_C$ y $path_E$; y finalmente $path_B$, $path_D$ y $path_E$. Si ahora analizamos los datos de entrada, se observa que las siguientes combinaciones también representan caminos imposibles: $path_E$; $path_A$ y $path_E$; $path_C$ y $path_E$; y $path_A$, $path_C$ y $path_E$. Además, en la segunda llamada la función *foo*, la combinación de los caminos $path_H$ y $path_I$, es un camino imposible. Finalmente, es necesario determinar el número máximo de iteraciones del bucle anidado de la función *bar*. Se podría tomar como primera aproximación el valor $100 \cdot 100 = 10000$, pero si aplicamos de nuevo las técnicas de ejecución abstracta se puede concluir que el número máximo de iteraciones del bucle es 5050.

Aunque, en general, no hay una solución óptima para identificar los *caminos imposibles*, ya que es un problema NP-completo, algunos métodos basados en ejecución abstracta proponen técnicas para reducir la complejidad del problema [73, 74, 75]. Pero esta reducción suele llevar asociada una pérdida de información y el WCET será sobrestimado.

2.3. Análisis del WCET basado en medida

El desarrollo de sistemas de tiempo real es una actividad esencial para la industria. El análisis dinámico es el único método que permite verificar el comportamiento del sistema en el entorno donde se ejecutará. Además, el correcto funcionamiento del sistema no sólo depende de la exactitud de los resultados obtenidos, sino que también depende del instante en el que dichos resultados se han generado, por lo tanto, es necesario verificar su comportamiento temporal.

Un método habitual para determinar el WCET de un programa es ejecutar el programa, o un trozo de éste, en un hardware propuesto, y medir el tiempo de la ejecución. La principal ventaja de los métodos basados en medida (MBA/ *Measurement-Based Analysis*) es que no es necesario analizar el comportamiento del procesador, ya que se mide el tiempo de ejecución del programa sobre el hardware real. La principal dificultad es que hay que generar los datos de prueba para la ejecución. El MBA utiliza información estática sobre el código y el hardware subyacente, para mejorar las estimaciones del WCET, así como el análisis de flujo de control para guiar el proceso de generación de los datos de prueba.

El WCET obtenido mediante las técnicas de análisis basado en medida no es seguro, a no ser que las pruebas de medición sean completas, pero esto suele ser imposible. El WCET conseguido sólo es una aproximación optimista, frente al WCET obtenido mediante el análisis estático, que es una aproximación totalmente segura pero pesimista.

Además, para evitar la explosión combinatoria de los posibles caminos de ejecución, también es necesario definir algún tipo de heurística que garantice cierta precisión en el WCET logrado. Así pues, suele ser habitual suponer que:

- Cada camino en el código fuente determina, al menos, un camino de ejecución en el código objeto.
- El tiempo de ejecución de un camino no depende de los datos de entrada que determinan ese camino.
- Para cada entrada es posible inicializar el estado de la máquina con el peor caso.

- Las variaciones externas, como la actividad del bus o el refresco de la *DRAM*, no influyen en el tiempo de ejecución.

Por ejemplo, bajo las hipótesis anteriores, si un bucle de n iteraciones contiene un condicional con p caminos alternativos, el número de posibles caminos de ejecución que es necesario analizar es p^n. Pero si en las m primeras iteraciones un camino $path_i$ tiene un coste de ejecución menor que cualquier otro camino $path_j$, se puede descartar del análisis. Esto evita la explosión de caminos y permite finalizar el análisis, obteniendo un WCET más o menos preciso. Desafortunadamente, esta forma de evitar la explosión combinatoria de caminos no es correcta en presencia de componentes con latencia variable dependientes de la historia de ejecución, como por ejemplo las memorias cache.

Examen del programa de principio a fin

Las técnicas que determinan el WCET, midiendo el tiempo de ejecución del programa de principio a fin, aprovechan los métodos utilizados en la industria para verificar el funcionamiento del software en sistemas críticos. Estas técnicas, que intentan descubrir los errores del software, también se pueden utilizar para obtener el tiempo de ejecución del *camino más largo* de un programa.

Mediante estas técnicas para determinar el WCET, se generan los datos de entrada necesarios para ejecutar todos los posibles caminos de ejecución del programa [199, 200, 205]. El análisis del WCET comienza seleccionando unos datos de entrada para el programa. A partir de estos datos de entrada, se ejecuta el programa y se mide el tiempo de ejecución de este camino particular. El proceso se repite hasta que todos los posibles caminos de ejecución se hayan analizado. Pero en los procesadores actuales la dificultad de generar todos los datos de entrada es cada vez mayor, debido a la explosión combinatoria de los posibles caminos de ejecución, que generan los componentes hardware que dependen de la historia de ejecución, como por ejemplo las memorias cache.

Utilización de algoritmos genéticos

Especialmente los algoritmos genéticos, dentro de la investigación evolutiva [59], han sido muy utilizados para determinar el tiempo de ejecución del *camino más largo* en programas de tiempo real [10, 60, 61, 62, 63, 64, 135, 140, 144, 150, 177, 178, 186, 187, 188, 189, 190, 191].

Si entendemos el cálculo del WCET de un programa como un problema de optimización, los algoritmos genéticos se pueden utilizar para automatizar el proceso de análisis. La idea básica es aplicar algoritmos genéticos para determinar los datos de entrada del *camino más largo*. Se trata por lo tanto de ejecutar el programa para una determinada entrada, que en estos casos se denomina población, y desarrollar durante el análisis una serie de mutaciones y una selección

de las nuevas entradas a ejecutar. Si no se produce una violación en las restricciones temporales del programa, el proceso se realiza un número predeterminado de veces, o hasta que el tiempo de ejecución observado no empeore. Al final del análisis, se toma como el WCET del programa el mayor tiempo de ejecución obtenido.

En general, el principal objetivo del análisis basado en algoritmos genéticos es buscar los datos de entrada que generan una violación en las restricciones temporales del programa. Si para una determinada entrada se encuentra un error temporal, el análisis finaliza inmediatamente. Pero si no es así, se hace considerablemente difícil decidir cuándo debe finalizar el análisis para garantizar la corrección temporal del programa. Así pues, uno de los principales problemas de utilizar algoritmos genéticos para calcular dinámicamente el WCET del programa es determinar cuándo debe finalizar el análisis, como ponen de manifiesto algunos trabajos de investigación [64, 140, 144, 186, 187, 188, 190, 191].

Puesto que, debido a la explosión de combinaciones de los datos de entrada del programa, no es viable analizar dinámicamente todos los posibles caminos de ejecución, el análisis dinámico nunca puede garantizar que el tiempo máximo obtenido durante la ejecución del programa sea el WCET. Aun así, algunos investigadores proponen utilizar algoritmos evolutivos para evitar dicha explosión de combinaciones y hacer posible el cálculo del WCET de forma dinámica [60, 61, 62, 63]. En estos trabajos también se presentan ciertas características que dificultan el análisis dinámico del WCET, como por ejemplo: que el programa tenga un gran número de bucles anidados, que haya una baja probabilidad de que se tome un determinado camino debido a la selección de los datos de entrada, que el tiempo de ejecución dependa de los datos de entrada o que el número de iteraciones de los bucles dependa de dichos datos.

Más recientemente se ha propuesto una técnica de búsqueda evolutiva para determinar el WCET de un programa, teniendo en cuenta los componentes hardware habituales en los procesadores actuales [15, 90]. En esta aproximación se presentan las propiedades, tanto del software, como del hardware del procesador, que tienen un impacto más significativo en el cálculo del WCET. Teniendo en cuenta estas características se determina el WCET del programa, definiendo una función objetivo donde tendrán mayor o menor importancia, por ejemplo, el tiempo de ejecución de las instrucciones, los fallos de cache, etc. Aunque el análisis se hace cada vez más complejo al añadir nuevas características, las técnicas de análisis dinámico evitan las *anomalías de distribución* (*timing anomalies*) [43, 115, 155, 193] y permiten considerar algunos componentes hardware, que dependen de la historia de ejecución, sin aumentar la complejidad de dicho análisis.

Cobertura local de caminos

En la literatura también podemos encontrar técnicas para calcular el WCET de un programa, que combinan el análisis de flujo de control para determinar las estructuras del programa, con el análisis basado en medida que nos permite obtener el tiempo de ejecución de los segmentos del programa en el hardware real [97, 98, 192, 194, 195, 196].

A continuación se enumeran los pasos que suelen seguir los métodos de análisis de cobertura local de caminos para analizar y calcular el WCET de una tarea:

1. Se realiza un análisis de flujo de control para determinar la estructura del programa. Este análisis se lleva a cabo en alto nivel, porque es más sencillo que analizar el código objeto.

2. El CFG del programa se divide en segmentos que abarcan todos los posibles caminos de ejecución.

3. Se generan los datos de prueba para medir el tiempo de ejecución de los segmentos.

4. El WCET se calcula mediante IPET teniendo en cuenta el tiempo de ejecución de cada uno de los segmentos y la información de la estructura del programa obtenida mediante el análisis de flujo de control.

En definitiva, las estimaciones del WCET basado en medida no son seguras, aunque para una entrada concreta son precisas y pueden estar más cerca del WCET que las obtenidas mediante el análisis estático, que son más pesimistas pero totalmente seguras. De hecho, sólo hemos encontrado un trabajo de investigación donde se describe un método basado en medida que puede generar, en determinadas circunstancias, resultados seguros [42]. En esta nueva propuesta se mide el tiempo de ejecución de un camino completo, pero si la medición no finaliza en un determinado tiempo, el camino se va dividiendo en trozos o segmentos cada vez más pequeños hasta que se puede medir el tiempo de ejecución de estos trozos, pero el tiempo de análisis podría exceder de lo razonable.

Análisis basado en probabilidades

El análisis del WCET basado en probabilidades se puede utilizar en sistemas en los que, aunque sea necesario conocer el WCET del programa, no sea estrictamente obligatorio garantizar el cumplimiento de sus restricciones temporales. Este tipo de análisis se presenta para determinar el WCET de un programa, garantizando con una alta probabilidad sus plazos de ejecución [18, 20, 21, 22]. Este análisis también puede tener un interés especial en la industria cuando sea muy costoso determinar el WCET de una tarea, o cuando la probabilidad de que se ejecute el *camino más largo* sea pequeña. Aunque el WCET obtenido mediante el análisis basado en probabilidades no sea seguro, este análisis permite

reducir la complejidad del problema y también la sobreestimación del WCET que se produce en los métodos de cálculo seguros.

A continuación se describen los pasos más habituales que sigue el análisis basado en probabilidades para conseguir el WCET:

1. Se construye el árbol de sintaxis abstracta que representa el programa.
2. Se determinan la distribución de probabilidad de los bloques básicos y sus dependencias.
3. Se define el álgebra de probabilidades que permite manipular las diferentes distribuciones.
4. Se indica cómo se combinarán las distribuciones de probabilidad de los bloques básicos con los nodos del árbol de sintaxis abstracta, para calcular el WCET del programa.
5. Finalmente se presentan los resultados probabilísticos asociados al WCET obtenido.

2.4. Análisis del procesador

Analizar el comportamiento del procesador, en el peor caso, es esencial para obtener un WCET preciso, ya que permite tener en cuenta características del hardware subyacente tales como la ejecución segmentada de las instrucciones, los predictores de saltos, las memorias cache, etc. Estos mecanismos hardware se utilizan para incrementar la velocidad de ejecución de un programa y son habituales en los procesadores actuales. Pero modelar el comportamiento del procesador, en el peor caso, es un problema complejo, validar el modelo resulta complicado, podría contener errores y además consume gran cantidad de tiempo. Así pues, analizar el funcionamiento del procesador se ha convertido en una de las principales áreas de investigación en el análisis y cálculo del WCET. Durante este análisis se intenta determinar el WCET de los bloques básicos o segmentos del programa, modelando de la forma más exacta posible y segura el comportamiento del hardware del procesador en el peor caso. Conviene indicar que si el tiempo de ejecución de todos los bloques básicos es constante, el cálculo del WCET del programa se obtiene de forma directa considerando el *camino más largo*, por lo que únicamente se requiere determinar el tiempo de ejecución de cada uno de los posibles caminos del programa y seleccionar el máximo.

Los primeros trabajos que estudian el comportamiento del procesador en el peor caso, suponen que el procesador no dispone de memorias cache o que éstas se han deshabilitado [34, 142, 145, 149, 151, 185]. Aunque en la actualidad estas suposiciones todavía se puedan considerar correctas, porque muchos sistemas de

tiempo real utilizan procesadores muy simples sin caches, los nuevos diseños, por ejemplo en telecomunicaciones, intentan utilizar la última tecnología hardware para incrementar las prestaciones. Por lo tanto, el tiempo de ejecución de los bloques básicos debe ser más sensible al hardware en el que se ejecuta el programa. Si no se aprovechan adecuadamente los recursos hardware disponibles, las estimaciones del WCET pueden ser muy pesimistas.

Una vez realizado el análisis del procesador es necesario integrar dicho análisis en las técnicas de cálculo del WCET presentadas en la Sección 2.1, pero esta integración no está exenta de dificultades. En la literatura se han propuesto técnicas que analizan y determinan el cálculo del WCET como un solo paso. Así por ejemplo, mediante ejecución simbólica se puede analizar el comportamiento del procesador, en particular la ejecución segmentada de instrucciones, y el comportamiento de las memorias cache [114, 116]. Estos trabajos, para evitar la complejidad exponencial del análisis, reducen el número de caminos mediante técnicas de fusión (*path merging*), pero la fusión de caminos lleva asociada una importante pérdida de información y el WCET suele ser ampliamente sobrestimado.

A continuación se describen algunas técnicas que modelan estáticamente los principales componentes del procesador en el peor caso, y también se indican algunos detalles para incorporar estos modelos al cálculo del WCET.

Análisis de la segmentación de instrucciones

La segmentación consiste en dividir la ejecución de cada instrucción en una serie de etapas con un tiempo de duración fijo. Por ejemplo una implementación típica del conjunto de instrucciones RISC se divide en cinco etapas o ciclos: capturar la instrucción (*fetch*), decodificar la instrucción (*decode*), ejecución (*execution*), acceso a memoria (*memory access*), y escritura del resultado (*write-back*) [83].

En el caso ideal, durante la ejecución de un bloque básico, se puede conseguir ejecutar hasta una instrucción por ciclo. Pero durante la ejecución, pueden surgir algunos problemas que impidan aprovechar este solapamiento, por ejemplo los riesgos estructurales que aparecen cuando los recursos hardware no son suficientes para mantener todas las instrucciones en ejecución; los riesgos de datos que surgen cuando una instrucción depende del resultado de la ejecución de una instrucción previa con la que ha solapado; y finalmente los riesgos de control que aparecen con las instrucciones de salto que modifican el contador de programa [83].

Así pues, en un procesador segmentado también se debe tener en cuenta el tiempo de ejecución de las instrucciones solapadas, para que la estimación del WCET sea precisa. Por lo tanto, es necesario analizar, por un lado, dentro de un

bloque básico, los retardos producidos por los riesgos de datos y estructurales; y por otro, entre dos bloques básicos consecutivos, los riesgos de control. En la literatura se han presentado diferentes trabajos que estudian el efecto de la segmentación en el cálculo del WCET [17, 47, 82, 79, 108, 141, 160, 169, 204]. La forma más habitual de analizar la influencia de la segmentación en el WCET es determinar el tiempo de ejecución de las instrucciones mediante tablas de tiempos o mediante simulación de la ejecución. El tiempo de ejecución de cada instrucción se utiliza posteriormente para determinar el WCET mediante las técnicas de cálculo ya comentadas en la Sección 2.1, por ejemplo mediante IPET [47, 169].

Mucho más ambiciosas son las propuestas para analizar el WCET en procesadores fuera de orden [103, 104]. La dificultad de estas propuestas reside principalmente en determinar el tiempo máximo de ejecución de cada uno de los bloques básicos durante la asignación dinámica de recursos en un procesador fuera de orden, evitando las *anomalías de distribución*. Esta propuesta modela, mediante restricciones, un problema ILP que determina el tiempo de ejecución de peor caso, de cada uno de los bloques básicos del programa.

Finalmente conviene indicar que algunas propuestas, además de estudiar la ejecución segmentada de instrucciones, también modelan el comportamiento de una cache de instrucciones [82, 79] o de un predictor de saltos [17].

Análisis del predictor de saltos

El predictor de saltos tiene como objetivo principal reducir el retardo que se puede producir después de una instrucción de salto. Para cada instrucción de control este mecanismo de predicción determina, estática o dinámicamente, si se producirá una interrupción en la secuencia de ejecución de las instrucciones del programa (salto tomado) o no (salto no tomado), mientras se calcula la dirección de la siguiente instrucción a ejecutar. Aunque los predictores de saltos mejoran la media del tiempo de ejecución del programa, en los sistemas de tiempo real, debido a las *anomalías de distribución*, la predicción del salto debe ser totalmente segura, asumiendo un fallo cuando no lo es.

Las estrategias utilizadas en la predicción de saltos se clasifican en estáticas, cuando la predicción del salto es siempre la misma, y en dinámicas, cuando la predicción del salto depende de la historia de ejecución [165]. A continuación se resumen brevemente algunas de las estrategias más sencillas utilizadas.

Estrategias estáticas:

- Predicen todos los saltos como tomados.
- Predicen los saltos tomados en función del código de operación.
- Predicen los saltos hacia atrás como tomados.

Estrategias dinámicas:

- Predicen el salto igual a como se realizó en la última ejecución.
- Predicen los saltos como tomados si aparecen en una tabla que se actualiza durante la ejecución.
- Predicen los saltos de acuerdo a uno o más bits que se actualizan durante la ejecución.

Los predictores dinámicos necesitan una pequeña memoria para guardar la historia de la ejecución a partir de la cual se realiza la predicción [83]. Esta memoria se denomina habitualmente *branch-prediction buffer* o *branch-history table*. Además, si para predecir el salto sólo se tiene en cuenta la propia instrucción de salto, los predictores dinámicos se denominan predictores locales, y si tienen en cuenta el comportamiento de todos los saltos realizados hasta ese instante se denominan predictores globales. Aunque los predictores de saltos dinámicos presentan mejor rendimiento que los estáticos, la predicción de su funcionamiento en el peor caso es más compleja, y algunos investigadores desaconsejan su uso en sistemas de tiempo real [45].

En la literatura se han presentado técnicas para utilizar predictores de saltos estáticos en sistemas de tiempo real [24, 26]. Sin embargo, las propuestas para determinar el funcionamiento en el peor caso de algunos predictores de saltos dinámicos son habituales. Por ejemplo, para guardar la historia de ejecución del programa se ha propuesto utilizar una tabla de destinos de saltos (BTB/ *Branch Target Buffer*). En algunas casos, para predecir el comportamiento de los saltos, se ha simulado estáticamente el comportamiento del BTB [39]. En otros se ha creado un entorno que permite analizar el WCET en presencia de algunos BTBs particulares [65, 66].

Especial relevancia adquiere la predicción de los saltos en bucles y mucho más en bucles anidados. Este problema se ha tratado con más detalle en diferentes trabajos, donde se presentan métodos de análisis estático para clasificar las instrucciones de salto en el código fuente, ampliando su análisis, tanto a un predictor local (*bimodal banch predictor*), como a un predictor global (*global-history branch predictor*) [16, 17, 25, 157].

En otros trabajos de investigación se modela el impacto sobre el WCET de un predictor de saltos genérico. A partir del grafo de flujo de control se determinan de forma automática el conjunto de restricciones lineales que modela el número de fallos del predictor de saltos. Con estas restricciones, y resolviendo un problema basado en ILP, se calcula el WCET del programa analizado [102, 127, 128]. Se trata en definitiva de maximizar el tiempo de ejecución de cada uno de los bloques básicos del programa, teniendo en cuenta los fallos en la predicción de los saltos. Si el bloque básico no contiene ninguna instrucción de salto, el tiempo de ejecución no se verá afectado por la predicción. Además, este

modelo también permite medir el impacto de los fallos de predicción en la cache de instrucciones [106].

2.5. El WCET con caches de instrucciones

Uno de los componentes hardware del procesador que más influyen en el tiempo de ejecución de una tarea son las memorias cache. Las memorias cache se utilizan para incrementar la velocidad media de acceso a la memoria principal, y por lo tanto reducen significativamente el tiempo de ejecución. Predecir el comportamiento de las memorias cache es esencial para poder calcular una cota del WCET precisa, pero analizar su comportamiento es complejo ya que depende de la historia de ejecución del programa.

Las memorias cache son almacenes pequeños de rápido acceso que contienen instrucciones y datos. Las memorias cache se dividen en conjuntos de líneas donde cada línea sólo puede almacenar un bloque de memoria. Cuando al referenciar a una instrucción o a un dato, el bloque que los contiene está en la cache, se produce un acierto de cache. En caso contrario, se produce un fallo de cache. Como el tamaño de la cache es mucho menor que la memoria principal, una línea de cache puede guardar varios bloques de memoria distintos, por lo tanto es necesario definir una función de correspondencia para decidir en que línea de cache se guarda un bloque de la memoria.

Existen tres tipos de correspondencia: directa, totalmente asociativa y asociativa por conjuntos. La correspondencia directa asigna a cada bloque de memoria una única línea de cache. Aunque esta correspondencia es simple, si durante la ejecución de un programa se referencia varias veces a instrucciones o datos de bloques diferentes asignados a una misma línea, se producirán muchos fallos de cache porque dichos bloques se estarán reemplazando continuamente. La correspondencia totalmente asociativa permite que cada bloque de memoria se pueda cargar en cualquier línea de cache. Pero en este caso, es necesario examinar todas las línea de la cache para ver si la instrucción o el dato referenciado está en la cache. Finalmente, la correspondencia asociativa por conjuntos es una solución de compromiso, ya que la cache se divide en S conjuntos de líneas. En este caso un bloque de memoria puede cargarse en un solo conjunto, y dentro de éste puede hacerlo en cualquiera de sus líneas. El número de líneas w que pertenecen a un conjunto se denominan vías. Así pues, la capacidad de una cache asociativa por conjuntos viene determinada por el producto $S \cdot w$. Lo mismo sucede para una cache de correspondencia directa, que tiene una única vía ($w = 1$); y para una cache totalmente asociativa, que tiene un único conjunto ($S = 1$). En la Figura 2.7 se muestra un ejemplo de cada uno de los tipos de correspondencia descritos.

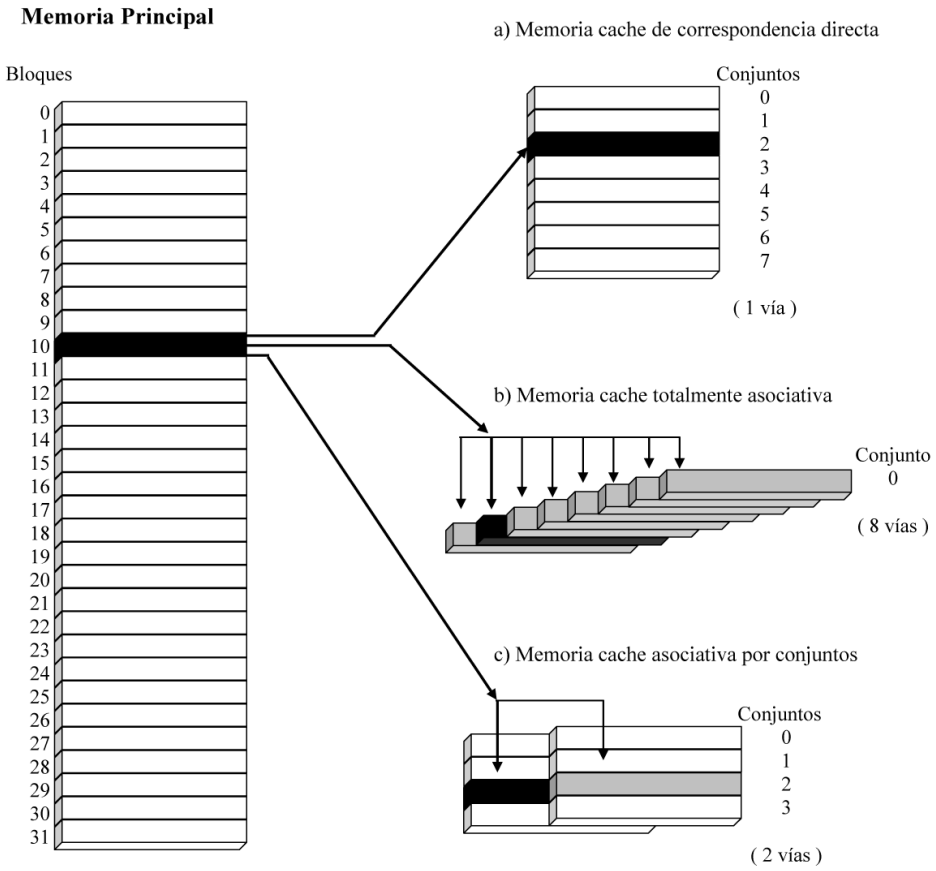


Figura 2.7: Posibles tipos de correspondencia entre bloques de memoria y líneas de cache.

Cuando se carga un nuevo bloque de memoria en la cache hay que determinar el conjunto en el que se guardará dicho bloque, y esto depende del tipo de correspondencia de la cache. Así pues, si denotamos con j el número de bloque de memoria principal que se cargará en la cache, el conjunto i donde se guardará dicho bloque se obtiene mediante la siguiente expresión:

$$i \quad = \quad j \mod S$$

Pero también es posible que haya que reemplazar alguna de las líneas del conjunto por el nuevo bloque de memoria. Para una cache de correspondencia directa no hay elección, ya que cada conjunto sólo tiene una línea de cache. Pero para las caches totalmente asociativas o asociativas por conjuntos, es necesario definir una política de reemplazo para poder seleccionar la línea de cache

a reemplazar. Existen diferentes algoritmos de reemplazo, no obstante, el más efectivo es el algoritmo LRU (*Least-Recently Used*) que sustituye, dentro de un conjunto, a la línea de cache con el bloque de memoria que más tiempo lleva sin ser referenciado. En el caso de que la línea de cache contenga datos, también se habrá de tener en cuenta la posibilidad de que dichos datos hayan sido actualizados durante la ejecución, ya que en este caso se debe actualizar la memoria principal. No obstante, existen dos políticas de escritura en la cache. Por un lado, la escritura inmediata (*write through*), en la que se actualiza a la vez la línea de cache y el bloque de memoria que contiene el dato, por lo tanto la memoria principal siempre está actualizada. Por otro, la escritura retardada (*write back*), en la que sólo se actualiza el dato en la cache, y sólo en caso de que sea reemplazada esa línea se actualiza el bloque de memoria.

En definitiva, el funcionamiento de las caches no es fácilmente predecible en tiempo de compilación. El contenido de sus líneas depende totalmente del camino seguido por el programa durante la ejecución. Determinar estáticamente las líneas de memoria presentes en la cache en todo instante de ejecución es difícil, puesto que equivale a calcular en cada referencia a memoria si va a producirse un fallo obligatorio, un fallo de capacidad o un fallo de conflicto [83].

Un fallo de cache obligatorio se produce siempre que se accede por primera vez a un bloque de memoria, ya que éste no puede estar en la cache. Un fallo de capacidad se produce porque la cache no puede contener todos los bloques del programa a los que se accede durante su ejecución. Así pues, algunos de estos bloques son descartados y posteriormente son otra vez solicitados durante la ejecución. Un fallo de conflicto se produce porque algunos bloques son reemplazados por otros, dentro de un mismo conjunto, y luego son otra vez requeridos durante la ejecución. Los fallos de conflicto se producen en caches asociativas por conjuntos o en caches de correspondencia directa. Así pues, algunos aciertos en caches totalmente asociativas, son fallos en caches asociativas por conjuntos en función del grado de asociatividad. Los fallos por conflicto son debidos a las interferencias, denominadas interferencias intrínsecas de cache, que se producen en la cache durante la ejecución de una tarea. En un sistema multitarea aparecen fallos por conflictos entre las diferentes tareas cuando se produce una expulsión, en este caso estos fallos de cache también se denominan interferencias extrínsecas de cache. En el momento en que la tarea, que ha sido expulsada de la CPU, reanude su ejecución, puede ser necesario volver a cargar algunas instrucciones y datos que tenía en la cache antes de la expulsión, esto provoca, en la ejecución de la tarea, un retardo adicional (*cache-related preemption delay*).

Las memorias cache en los procesadores de altas prestaciones se organizan en varios niveles integrados en el chip. Hoy en día podemos encontrar hasta tres niveles integrados, el primero pequeño y con una latencia de uno o dos ciclos y el resto progresivamente más grandes y lentos. En la Tabla 2.2 se muestran algunos ejemplos de procesadores clásicos con sus configuraciones de cache.

Procesadores de propósito general

Procesador	Arquitectura	Cache en chip			
		L_1 i	L_1 d	L_2 u	L_3 u
Intel Itanium 2 Madison	Itanium	16KB	16KB	256KB	6MB
AMD Opteron	IA-32	64KB	64KB	1MB	
Intel P4 Xeon		96KB	8KB	512KB	6MB

Procesadores para sistemas empotrados

Procesador	Arquitectura	Cache en chip			
		L_1 i	L_1 d	L_2 u	L_3 u
IBM PPC 750GX	PowerPC	32KB	32KB	1MB	
PMC-Sierra RM9000*2GL	MIPS64	16KB	16KB	256KB	
ARM 1020E	ARM	32KB	32KB		
F. PowerPC MPC 7448	PowerPC	32KB	32KB		

Tabla 2.2: Configuraciones de cache integrada en chip en procesadores clásicos. L_1, L_2 y L_3 indican el nivel de cache. Se utiliza i para instrucciones, d para datos, u para instrucciones y datos.

Los diseñadores de sistemas de tiempo real, en una primera aproximación, han decido no utilizar memorias cache, debido a su comportamiento no predecible. No obstante, cuando en el diseño del sistema se propone un procesador con caches, se determina el WCET de las tareas asumiendo que cada acceso a memoria será un fallo de cache. Pero el WCET de cada tarea es ampliamente sobrestimado y el análisis de *planificabilidad* puede fallar cuando en realidad el sistema es capaz de cumplir todos sus requisitos temporales.

El análisis del WCET de una tarea en presencia de memorias cache presenta dos tipos de retos bien distintos. Por un lado, es necesario determinar las interferencias propias de la tarea o *interferencias intrínsecas*, que aparecen cuando durante la ejecución de una tarea, en la cache se reemplazan algunos de sus propios bloques de memoria debido a los fallos de capacidad o de conflicto. Por otro lado, en un sistema multitarea con expulsiones hay que determinar las interferencias que se producen entre las distintas tareas, también denominadas *interferencias extrínsecas*. Estas interferencias aparecen cuando, durante la ejecución de una tarea, en la cache se reemplazan bloques de memoria que pertenecen a otras tareas del sistema [14].

En la literatura se han estudiado ampliamente ambas cuestiones. Para determinar las interferencias intrínsecas se ha modelado el comportamiento de la cache en el peor caso, cuando las tareas se ejecutan de forma aislada, y para determinar las interferencias extrínsecas se ha calculado el retardo adicional en el tiempo de ejecución de la tarea, que se puede producir si es necesario cargar algunas instrucciones y datos que fueron expulsados durante la ejecución de otras tareas.

No obstante, la obtención de una cota ajustada y segura del WCET de una tarea en presencia de memorias cache, depende de la exactitud de la predicción de su funcionamiento. Es decir, se debe predecir si el acceso a una instrucción o dato será un acierto o un fallo, asumiendo fallo en caso de duda.

Interferencias extrínsecas y *planificabilidad*

En este apartado se comentan algunas de las propuestas más interesantes para determinar, en un sistema multitarea, el impacto producido por las interferencias extrínsecas de la cache sobre el WCET de una tarea.

Una primera aproximación, suponiendo conocido el WCET de una tarea que se ejecuta de forma aislada C_i, añade dos nuevos términos constantes en el análisis de *planificabilidad* del sistema [14]. El primer término está asociado al coste de los cambios de contexto δ que se producen en los sistemas multitarea [129]. El segundo término está asociado al retardo adicional γ que se producirá como consecuencia de las interferencias extrínsecas de la cache. Por lo tanto, el nuevo WCET de la tarea $C_i^{'}$ queda determinado por la siguiente ecuación:

$$C_i^{'} = C_i + 2\delta + \gamma \tag{2.1}$$

Así pues, la expresión que calcula la utilización del procesador U, y en definitiva proporciona una condición suficiente para determinar la *planificabilidad* de un sistema con prioridades estáticas planificado mediante RM (*Rate Monotonic*) [111], también se debe modificar como se muestra en la siguiente ecuación:

$$U = \sum_{i=i}^{N} \frac{C_i^{'}}{D_i} \leq N \cdot \left(2^{\frac{1}{N}} - 1\right) \tag{2.2}$$

La Ecuación 2.2 es una condición suficiente pero no necesaria, por lo que este análisis de *planificabilidad* introduce cierto pesimismo, ya que un sistema puede ser *planificable* sin verificar dicha expresión. Así pues, también se ha propuesto analizar el tiempo de respuesta R_i de cada tarea $Task_i$ del sistema utilizando RTA (*Response Time Analysis*) [27, 78, 87, 111]. Es decir, las tareas de un sistema verifican sus restricciones temporales si $R_i \leq D_i, \forall i$, siendo D_i su plazo de finalización. Para ello, es necesario modificar el WCET de las tareas que pueden expulsar a la tarea analizada $Task_i$ añadiendo la penalización por la recarga de la cache, como se indica en la siguiente ecuación recursiva que permite calcular el tiempo de respuesta de cada tarea del sistema teniendo en cuenta dicha penalización.

$$R_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{D}_j \right\rceil \cdot (C_j + \gamma_j) \tag{2.3}$$

La principal dificultad de las propuestas, que estudian el tiempo de respuesta, reside en determinar de una forma más precisa la penalización por la recarga de la cache, aunque el análisis de *planificabilidad* es equivalente al propuesto inicialmente [27, 78, 87, 111]. A continuación se enumeran algunas de las propuestas para obtener la penalización por la recarga de la cache en un cambio de contexto:

- Considerar el tiempo de recarga de la cache completa.
- Calcular el tiempo de recarga de los bloques reemplazados por la tarea preferente.
- Determinar el tiempo de recarga de los bloques activos de la tarea expulsada.
- Determinar el tiempo de recarga para el máximo número de bloques activos que puede tener la tarea expulsada en cada instante.
- Considerar el tiempo de recarga de los bloques de cache compartidos entre la tarea expulsada y la preferente.

La penalización por recarga más sencilla de obtener es calcular el tiempo de recarga de la cache completa, ya que en cualquiera de los otros casos enumerados es necesario tener un conocimiento más detallado del contenido de la cache en cada instante de la ejecución del sistema. Esta penalización es la mayor que se debe considerar, y puede hacer que el análisis de *planificabilidad* falle. Así pues, en los primeros trabajos de investigación ya se optó por utilizar como penalización el tiempo de recarga de los bloques reemplazados por la tarea preferente [29, 32, 30].

Sin embargo, en trabajos posteriores se han mejorado y calculado cotas más precisas del tiempo de recarga de la cache en los cambios de contexto [100, 101, 139, 170, 171]. Por ejemplo, en un sistema de prioridades fijas se ha propuesto analizar los estados de la cache cuando una tarea es expulsada, y los estados de la cache asociados a las tareas que se ejecutarán durante su expulsión [100, 101]. De esta forma, se determina una cota más ajustada del tiempo de recarga de la cache, ya que por un lado se obtiene el tiempo de expulsión en cada punto del programa, teniendo en cuenta el estado o contenido de la cache, y por otro se consigue el número de expulsiones y los puntos de expulsión en el peor caso.

También son de interés, las propuestas que analizan los posibles caminos de ejecución para determinar los estados de cache asociados a la tarea expulsada y a la tarea preferente [139, 170, 171]. Este análisis de los estados de cache asociados a los caminos permite obtener de una forma más exacta el tiempo de penalización por recarga de la cache. En concreto, proponen calcular la penalización, como el tiempo de recarga de las líneas de cache compartidas entre la tarea expulsada y la preferente. Aunque esta propuesta supone que las posibles

expulsiones sólo tienen lugar al final de cada bloque básico, esta simplificación queda perfectamente justificada [170, 171].

No obstante, a pesar de estas consideraciones, sigue siendo difícil conseguir un tiempo de penalización ajustado por recarga de la cache en sistemas multitarea con expulsiones. Así pues, para evitar el problema de las interferencias extrínsecas de cache, también se han propuesto métodos para repartir o dividir la cache (*cache partitioning*) entre las diferentes tareas del sistema [91, 92, 202]. Se trata de asignar a cada tarea una porción o trozo de la cache para su uso exclusivo. Obviamente, al dividir la cache se eliminan las interferencias extrínsecas, pero pueden aumentar las interferencias intrínsecas, ya que el tamaño de cache disponible para cada tarea se reduce considerablemente, por lo tanto podría aumentar el número de fallos de cache y también el WCET de las tareas. Pero determinar el tamaño de cache más adecuado que se asignará a cada tarea del sistema es un problema NP-completo y suele ser habitual dividir la cache en trozos iguales. Por lo tanto, todas las tareas, independientemente de su prioridad, estructura y tamaño, disponen de un trozo de cache de igual capacidad.

Finalmente también tienen interés las técnicas híbridas que combinan, por un lado la división de la cache entre algunas tareas, y por otro el cálculo del tiempo de recarga de la cache para las tareas que comparten un mismo trozo, como se propone en [31].

Análisis estático de la cache

Modelar de forma estática el comportamiento en el peor caso de una cache de instrucciones es difícil, ya que es necesario predecir como serán todos los accesos a memoria durante la ejecución del programa. A continuación, para un sistema de tiempo real, se describen algunas técnicas de análisis estático que modelan el comportamiento de una cache de instrucciones con algoritmo de reemplazo LRU. El análisis estático del comportamiento en el peor caso de la cache siempre es seguro, pero este análisis, que es pesimista por definición, puede añadir una importante sobrestimación si no se tiene en cuenta la historia de ejecución del programa. Una vez clasificados todos los accesos a memoria, el tiempo de ejecución en el peor caso de cada bloque básico del programa que se obtiene es seguro, aunque su precisión depende de la exactitud en la predicción de los accesos a memoria realizados en el peor caso.

En la literatura se han propuesto dos técnicas diferentes para analizar el comportamiento en el peor caso de la cache de instrucciones. La primera de las técnicas utiliza la simulación estática de la cache (SCS/ *Static Cache Simulation*) para clasificar en el peor caso los accesos a la cache de instrucciones [9, 79, 82, 108, 131, 132, 133, 134, 137, 136]. La segunda técnica utiliza la interpretación abstracta para analizar formalmente el comportamiento en el

peor caso de la cache de instrucciones [53, 54, 55, 56, 174, 175, 176].

Como ya se habrá observado en otros apartados de esta Tesis, tanto la simulación estática, como la interpretación abstracta se han utilizado ampliamente en el análisis y cálculo del WCET de un programa, puesto que los resultados obtenidos mediante estas técnicas de análisis estático son seguros. La interpretación abstracta es una teoría general, basada en semánticas, que permite definir diferentes análisis estáticos de los programas que por construcción son correctos [40]. El objetivo de la interpretación abstracta es extraer información segura y correcta de forma automática, sobre el comportamiento dinámico de los programas, sin necesidad de ejecutarlos. Aunque mediante la interpretación abstracta siempre se obtiene una aproximación, es posible garantizar que los resultados del análisis describen de forma segura el comportamiento del programa en algunas ejecuciones y que dicho análisis finaliza en un tiempo determinado.

Predicción de aciertos y fallos con la simulación estática de la cache

La técnica *SCS* permite identificar estáticamente los bloques de memoria que contiene la cache en cada instante de la ejecución. El contenido de la cache se obtiene mediante simulación a partir de los bloques básicos del programa y del grafo de flujo de control. Pero para evitar la explosión combinatoria de caminos, el contenido de la cache se considera abstracto, ya que agrupa, hasta un determinado punto del programa, todos los estados de la cache asociados a cada uno de los posibles caminos de ejecución. Además, cualquier conflicto entre las líneas de memoria que pueda contener la cache se analiza de forma pesimista, considerando siempre el peor caso de ejecución. La clasificación final de cada acceso a memoria se determina en función del contenido de los estados abstractos de la cache [9, 79, 82, 108, 131, 132, 133, 134, 137, 136].

Mediante un ejemplo sencillo, en la Figura 2.8 se muestran, en la parte izquierda, los estados concretos de cache alcanzados mediante la simulación estática de cache, y en la parte derecha, los estados abstractos obtenidos mediante *SCS* en un condicional (ver Figura 2.8 a)) y en un condicional dentro de un bucle (ver Figura 2.8 b)). La cache estudiada tiene 4 conjuntos con 2 vías.

La clasificación propuesta por *SCS* para cada uno de los accesos a memoria realizados durante la ejecución de un programa es la siguiente:

- *Siempre acierto*: una instrucción se clasifica como siempre acierto si se puede asegurar que siempre está en cache.
- *Siempre fallo*: una instrucción que siempre se puede garantizar que no está en cache se clasifica como siempre fallo.
- *Primer acierto*: son instrucciones de las que sólo podemos garantizar que están en cache cuando se accede por primera vez a la instrucción. Por

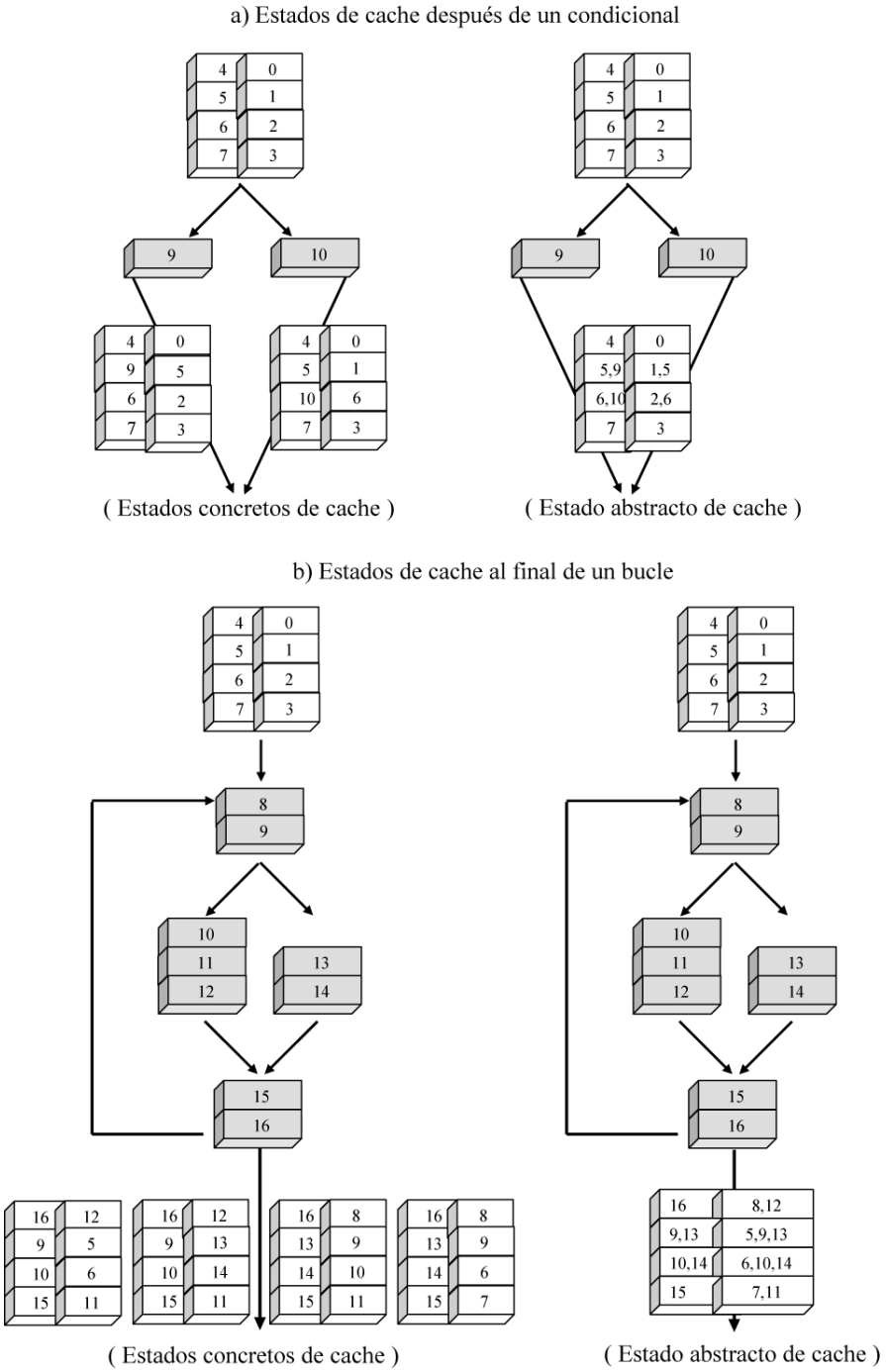


Figura 2.8: Estados concretos vs. estado abstracto conseguido mediante *SCS*.

ejemplo, dentro de un bucle, una instrucción cuyo bloque de memoria será expulsado durante la ejecución.

- *Primer fallo*: son instrucciones de las que no se puede garantizar que estén en la cache en su primer acceso, pero sí se puede asegurar que estarán en los siguientes accesos. Por ejemplo, las instrucciones de un bucle cuyos bloques de memoria no son expulsados durante la ejecución del bucle.

- *Sin clasificar*: son las instrucciones que no se han podido clasificar como ninguna de las anteriores. Por lo tanto son instrucciones que se deben clasificar como *siempre fallo* para que el análisis del WCET sea seguro.

La clasificación de accesos a memoria que se obtiene mediante el método *SCS* es totalmente segura, aunque en general el análisis siempre es pesimista al considerar como fallos todos los accesos que no se han podido clasificar. Como las direcciones de memoria de las instrucciones de un programa se pueden conocer estáticamente, esta técnica permite clasificar los accesos a instrucciones de forma precisa en códigos sin condicionales dentro de bucles, ya que simulando todos los accesos a memoria realizados por el programa se puede garantizar si un bloque de memoria está en la cache. Pero en códigos con estructuras condicionales dentro de bucles esto ya no es tan sencillo, debido a la pérdida de información que se produce al crear los estados abstractos de cache, como se muestra en la Figura 2.8 b). Por ejemplo, *SCS* clasifica el acceso a la primera instrucción del bloque de memoria 9 como *primer fallo*, por lo tanto la predicción es correcta. Sin embargo, el acceso a la primera instrucción del bloque 8 se clasifica como *siempre fallo*. En este caso, la predicción no es precisa, ya que cuando el programa sigue la rama condicional de la izquierda es *siempre fallo*, pero cuando el programa sigue la rama de la derecha es *siempre acierto*.

En definitiva, esta técnica permite predecir el comportamiento de la cache y determinar el tiempo de ejecución de cada bloque básico, en algunos casos de forma segura y precisa, incluso sin tener en cuenta la historia reciente de ejecución. En la mayoría de los trabajos que utilizan *SCS*, una vez obtenido el tiempo de ejecución de cada bloque básico, se calcula el WCET del programa mediante el árbol de sintaxis abstracta.

Finalmente conviene indicar que la deficiencia más significativa de esta técnica es que los accesos a memoria, que no se pueden clasificar, dependen, tanto de la estructura de programa, por ejemplo el número de posibles caminos de ejecución dentro de un bucle, como de la propia organización de la cache, principalmente del tamaño y de la asociatividad. Así, por ejemplo si durante la ejecución del programa se producen pocos conflictos de cache, o si los diferentes caminos de ejecución que puede seguir el programa tienen entre sí pocas líneas de cache compartidas, el número de accesos a memoria que no se pueda clasificar será pequeño, y por lo tanto el WCET obtenido será más preciso. Pero si por el contrario hay muchos conflictos de cache entre los diferentes caminos

de ejecución del programa, o una instrucción de un bucle se clasifica de forma pesimista, el WCET obtenido puede ser ampliamente sobrestimado.

Predicción de aciertos y fallos mediante interpretación abstracta de la cache

Analizando la estructura del programa mediante interpretación abstracta también se pueden clasificar, en el peor caso, los accesos a memoria en presencia de una cache de instrucciones. Así pues, una vez clasificados todos los accesos a instrucciones, se obtiene el tiempo de ejecución, en el peor caso, de los bloques básicos. A partir de estos tiempos de ejecución y con la información de flujo de control, se puede obtener un WCET seguro del programa.

Las técnicas basadas en interpretación abstracta permiten clasificar los accesos a memoria, realizando un análisis formal que garantiza si un determinado bloque *debe* o *puede* estar en la cache, en un punto concreto de la ejecución del programa. Si el análisis, en un punto de la ejecución, determina que un bloque de memoria *debe* estar en la cache, todos los accesos a las instrucciones de dicho bloque se consideran aciertos. Los bloques de memoria que *pueden* estar en la cache se utilizan para garantizar la ausencia en la cache de otros bloques. Así pues, todos los accesos a los bloques que no *pueden* estar en la cache se consideran fallos. No obstante, pueden quedar bloques sin clasificar, es decir, algunos accesos a memoria nunca pueden ser clasificados ni como aciertos ni como fallos. Por lo tanto, para que el WCET obtenido sea seguro, todos los accesos que no se han clasificado se consideran fallos [53, 54, 55, 56, 174, 175, 176].

En la Figuras 2.9 a) y b) se muestra respectivamente como se construyen los estados abstractos de cache que permiten determinar los bloque de memoria que *deben* o *pueden* estar en la cache, mediante interpretación abstracta.

En principio, la clasificación de los accesos a la cache de instrucciones obtenida con interpretación abstracta no es tan exacta como la obtenida mediante *SCS*. Sin embargo esta clasificación se puede mejorar, por un lado, añadiendo información relacionada con la estructura del programa, como por ejemplo analizando la primera iteración de los bucles de forma separada, ya que se consigue predecir de una forma más exacta el comportamiento de la cache; por otro, realizando un análisis *persistente* (ver Figura 2.9 c)) que permite predecir de una forma más exacta aquellas instrucciones que en su primer acceso no estaban en la cache, pero que sí estarán en los siguientes accesos. Conviene indicar que el análisis *persistente* permite predecir los accesos clasificados como *primer fallo*, mediante *SCS*. Así pues, cuanto más precisa sea la predicción de los accesos a memoria, más ajustado será el WCET calculado.

Las propuestas que utilizan interpretación abstracta determinan el WCET del programa mediante ILP. La función objetivo del problema ILP permite

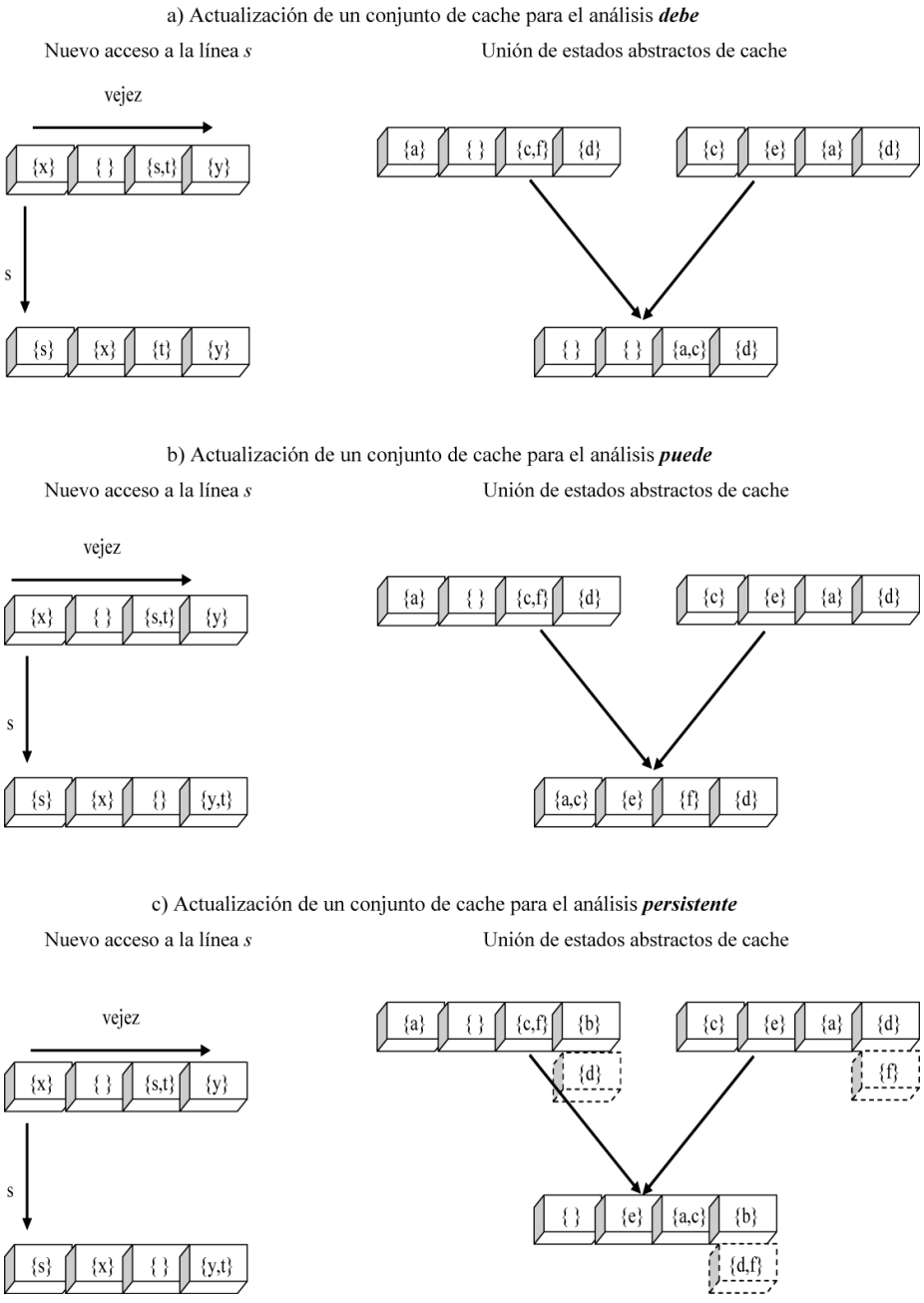


Figura 2.9: Interpretación abstracta: configuración de estados abstractos de cache.

tener en cuenta bloques básicos con tiempos de ejecución diferentes. Así, por ejemplo, el primer acceso a una instrucción de un bucle se puede considerar fallo y el resto de accesos a dicha instrucción se pueden considerar aciertos. En este caso, el bloque básico que contiene esta instrucción tendrá varios tiempos de ejecución asociados. No obstante, si cada instrucción de un bloque básico siempre se clasifica como acierto o como fallo, el bloque básico sólo tiene asociado un tiempo de ejecución. Además, formular un problema ILP para obtener el WCET permite añadir nuevas restricciones, tanto estructurales, como funcionales, que permiten determinar el WCET de una forma más precisa que el obtenido a partir del árbol de sintaxis abstracta.

Fijar el contenido de la cache

Analizar el comportamiento en el peor caso, de una cache de instrucciones, puede presentar complejidad exponencial, en concreto en bucles que contengan varios caminos condicionales. Para evitar en gran medida esta complejidad, tanto la simulación estática, como la interpretación abstracta, realizan algunas simplificaciones durante el análisis, como por ejemplo unir varios estados concretos de cache en un único estado abstracto [9, 79, 82, 108, 131, 132, 133, 134, 137, 136]. Pero cualquier técnica que reduce o simplifica la complejidad del problema lleva asociada una importante pérdida de información que se traduce en una sobrestimación en el número de fallos. Si esto es así, esta pérdida de información hace que el WCET obtenido sea sobrestimado.

Una técnica que permite predecir de forma exacta el comportamiento de la cache de instrucciones es fijar o bloquear su contenido. Fijar el contenido de la cache no presenta gran dificultad, ya que por ejemplo bastaría con deshabilitar el algoritmo de reemplazo. Muchas familias de procesadores clásicos permiten fijar el contenido de la cache, como por ejemplo: de la familia Intel, el Intel 960; de la familia MIPS el MIPS32; de la familia ARM el 940 y 946E-S; de la familia Motorola, el ColdFire, PowerPC, MPC7451 y MPC7400; y de la familia *Integrated Device Technology* el 79R4650 y 79RC64574; etc. [147, 118, 182].

Aunque fijar el contenido de la cache puede producir una degradación de su rendimiento, si el contenido de la cache está bloqueado todos los accesos a memoria ya son predecibles, y por lo tanto el cálculo del WCET del programa es seguro y más sencillo. Pero seleccionar los contenidos a fijar en la cache, para conseguir un WCET preciso, es complejo, ya que es necesario analizar de forma estática todos los accesos a memoria y tener en cuenta el impacto de la selección propuesta en el WCET. Así pues, la clave para poder utilizar hardware no predecible, como las memorias cache, en un sistema de tiempo real, está en adquirir un compromiso entre la predicción y las prestaciones de su funcionamiento.

En la literatura se han propuesto y evaluado dos técnicas diferentes para bloquear el contenido de la cache de instrucciones durante la ejecución de una

tarea en un sistema de tiempo real. La primera técnica propone bloquear la cache durante toda la ejecución del sistema. En la literatura, esta técnica se conoce como *static loking cache*. La segunda técnica propone fijar el contenido de la cache durante algunos periodos de la ejecución del sistema, por lo tanto su contenido puede ser actualizado bajo algunas circunstancias especiales. En la literatura, a esta técnica se la denomina *dynamic locking cache*.

Fijar el contenido durante toda la ejecución

Fijar el contenido de la cache durante toda la ejecución del sistema permite utilizar caches en sistemas de tiempo real multitarea. No obstante, para poder fijar su contenido, es necesario realizar un análisis estático y determinar los contenidos a cargar en la cache durante el arranque del sistema, para después bloquear su contenido. La predicción del funcionamiento de la cache es totalmente segura y exacta, ya que su contenido no cambia durante toda la vida del sistema. Sin embargo su rendimiento se reduce considerablemente, ya que todas las tareas comparten la cache simultáneamente y su contenido no puede ser actualizado.

En la literatura se han propuesto diferentes técnicas para seleccionar los contenidos a fijar en la cache. En algunos trabajos se ha propuesto la utilización de algoritmos genéticos [117, 118, 120, 121, 122, 124]. El objetivo de los algoritmos genéticos es seleccionar las líneas de memoria que se deben cargar en la cache para que el tiempo de ejecución de cada tarea del sistema sea el menor posible. Pero el coste computacional de estos algoritmos genéticos suele ser muy alto y, aunque los resultados obtenidos pueden ser interesantes, la selección propuesta no tiene por qué ser la mejor.

En sistemas multitarea con expulsiones, también se han propuesto algoritmos de baja complejidad cuyo objetivo se centra en reducir la utilización del procesador para cada una de las tareas del sistema. Además, esto permite que el tiempo de respuesta de cada tarea sea mucho más preciso [8, 146, 147]. El principal objetivo de estos algoritmos de baja complejidad es conseguir que un sistema sin cache de instrucciones que no es *planificable*, lo sea utilizando una cache que pueda fijar su contenido durante toda la vida del sistema.

En concreto, se han propuesto dos tipos de algoritmos: por un lado, se han seleccionado las líneas de memoria con el mayor número de accesos, para minimizar la utilización del procesador *Lock-MU* (*Algorithm for Minimize Utilization*); y por otro se ha intentado minimizar las interferencias entre las diferentes tareas del sistema *Lock-MI* (*Algorithm for Minimize Interferentes*). Aunque los resultados mostrados son muy parecidos, el algoritmo *Lock-MU* siempre obtiene mejores resultados que el algoritmo *Lock-MI* [147]. Sin embargo el conjunto de líneas a fijar en la cache, propuesto por estos algoritmos, no es la solución óptima al problema, aunque consigan que el sistema sea *planificable*. Así pues, con una

selección adecuada de las líneas a fijar en la cache, se obtiene un WCET seguro y más preciso para cada una de las tareas, haciendo que el sistema también sea *planificable*.

Finalmente los algoritmos genéticos se han comparado con los algoritmos de baja complejidad [123]. Los algoritmos genéticos analizan cada tarea del sistema de forma individual, mientras que los algoritmos de baja complejidad tienen en cuenta el sistema completo. Aunque los resultados de ambas propuestas puedan parecer equivalentes, el coste computacional de los algoritmos genéticos es mucho mayor que el coste computacional de *Lock-MU* y *Lock-MI*.

Fijar el contenido durante algunos periodos de la ejecución

Como ya se ha comentado anteriormente, fijar la cache durante toda la ejecución del sistema lleva asociada una importante pérdida en su rendimiento. Por un lado, al fijar su contenido se restringe su funcionamiento habitual, por otro se comparte la cache entre todas las tareas del sistema a la vez. Así pues, también es interesante fijar el contenido de la cache durante algunos periodos de la ejecución del sistema, por ejemplo cuando se ejecutan las tareas. Mientras, en los cambios de contexto, el contenido de la cache se actualizará para que cada tarea pueda aprovechar toda la capacidad de la cache y su comportamiento sea más dinámico, como se propone en varios trabajos [117, 119, 120, 122, 173]. Pero en ninguno de estos trabajos se ha presentado una solución óptima para seleccionar los contenidos a fijar en la cache durante la ejecución particular de cada tarea del sistema.

Bloquear el contenido de la cache durante algunos periodos de la ejecución puede mejorar de forma significativa el rendimiento del sistema. Pero en general, el análisis estático que selecciona las líneas de memoria, que se cargarán y bloquearán en la cache en cada cambio de contexto, debe ser más detallado para cada tarea del sistema, y por lo tanto será más complejo. Por ejemplo, si en cada cambio de contexto se permite actualizar el contenido de la cache, la tarea que reanude su ejecución podrá aprovechar toda la cache. Sin embargo, cuando se realice el análisis de *planificabilidad* es necesario tener en cuenta el coste de cargar las líneas de memoria de cada tarea en la cache, antes de volver a bloquear su contenido.

Contenidos a fijar en la cache

En la Figura 2.10 se presenta un ejemplo con los posibles contenidos a fijar en una cache con 4 conjuntos de correspondencia directa. En la parte superior de la figura se muestran las líneas de memoria asociadas a las dos tareas $Task_1$ y $Task_2$ que forman un sistema de tiempo real. La tarea $Task_1$ contiene un condicional con dos posibles caminos de ejecución dentro de un bucle. En el

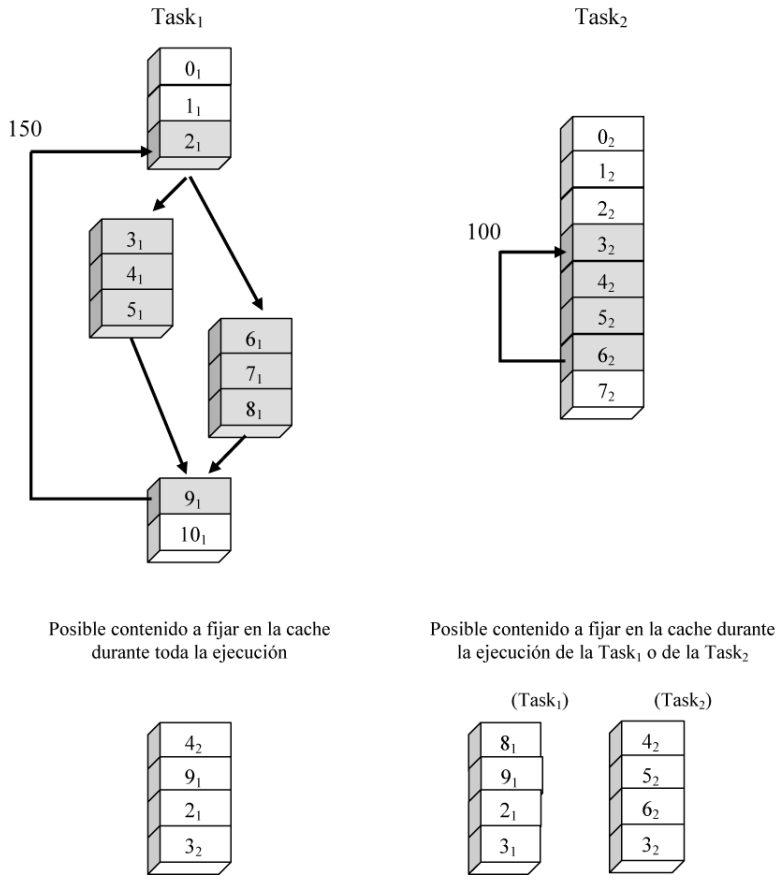


Figura 2.10: Posibles contenidos a fijar en la cache: durante toda la ejecución del sistema vs. durante la ejecución de cada tarea.

ejemplo, el número de iteraciones del bucle es 150. La tarea $Task_2$ contiene un bucle de 100 iteraciones con un único camino de ejecución. En la Figura 2.10 se han marcado las líneas de memoria más adecuadas para bloquear en la cache, pero debido a su capacidad, no es posible fijar todas estas líneas a la vez.

Supongamos que el contenido de la cache se fijará durante toda la ejecución del sistema. Una posible selección de líneas de memoria a fijar en la cache podría ser la formada por las líneas $line_2$ y $line_9$ de la tarea $Task_1$ y las líneas $line_3$ y $line_4$ de la tarea $Task_2$, como se muestra en la parte inferior izquierda de la Figura 2.10. Se han seleccionado las líneas de memoria a las que más veces se accede durante la ejecución. No se han elegido más líneas de la tarea $Task_1$ porque no se conoce con exactitud el número de accesos a dichas líneas, ya que depende del camino seguido durante la ejecución. Aunque ya se puede predecir

de forma exacta el comportamiento de la cache, la tarea $Task_2$ no puede aprovechar toda la cache. Además, es posible que esta selección no sea la óptima, por ejemplo cuando la rama izquierda o la rama derecha del condicional de la tarea $Task_1$ se ejecute más de 100 veces.

Supongamos ahora que la cache quedará fijada durante la ejecución de cada tarea y que en los cambios de contexto podemos actualizar su contenido. En este caso, para calcular un WCET preciso, es necesario tener en cuenta el coste de actualizar el contenido de la cache cuando una tarea comience o reanude su ejecución. Una posible selección de líneas de memoria a fijar en la cache durante la ejecución de la tarea $Task_1$ y de la tarea $Task_2$ se muestra en la parte inferior derecha de la Figura 2.10. De nuevo se han seleccionado las líneas de memoria, de cada tarea, a las que más veces se accederá durante la ejecución. En este caso, la selección de líneas de memoria de la tarea $Task_2$ es la óptima. Pero no ocurre lo mismo con la selección de líneas de la tarea $Task_1$, ya que no se conoce el número de veces que se ejecuta cada uno de los caminos del condicional. En este caso se ha optado por elegir una línea de memoria de cada camino.

En definitiva, bloquear la cache, durante algunos periodos de la ejecución del sistema, permite su utilización de una forma más dinámica en sistemas de tiempo real, ya que su predicción sigue siendo segura aunque resulte más difícil seleccionar las líneas de cada tarea a fijar en la cache, durante su ejecución. Sin embargo, es necesario tener en cuenta el coste de actualizar el contenido de la cache para obtener de forma precisa el WCET de cada tarea.

2.6. Conclusiones

El análisis y cálculo del WCET es complejo y difícil. Así pues, los trabajos de investigación que tratan este problema se centran en algunos aspectos concretos del análisis. En la literatura se han propuesto técnicas de análisis de flujo de control, análisis del comportamiento del procesador y técnicas para medir el tiempo de ejecución de un programa. El WCET obtenido mediante las técnicas basadas en medida no es seguro. Las técnicas de análisis estático son seguras, pero en algunas ocasiones sobrestiman el cálculo del WCET. Para obtener un WCET preciso y seguro, es necesario analizar estáticamente, tanto la estructura del programa mediante el análisis de flujo de control, como el comportamiento en el peor caso de los componentes hardware del procesador.

Uno de los componentes hardware más utilizados en los procesadores actuales son las memorias cache. Las memorias cache reducen la media del tiempo de acceso a la memoria principal. Pero predecir su comportamiento en el peor caso es complejo, ya que depende de la historia de ejecución. Para analizar el comportamiento en el peor caso de las memorias cache es necesario tener en cuenta, tanto las interferencias intrínsecas, como las interferencias extrínsecas

de la cache.

Para resolver el problema de las interferencias intrínsecas de cache, en la literatura se han propuesto técnicas como la simulación estática de la cache, o métodos basados en la interpretación abstracta. Estas técnicas de análisis son seguras y permiten determinar la contribución al WCET de los accesos a memoria, pero en muchas ocasiones esta contribución es sobrestimada. En el Capítulo 3, cuando el número de caminos condicionales dentro de un bucle no es grande, presentamos una técnica para determinar la contribución exacta al WCET de los accesos a memoria. En este capítulo también comparamos nuestros resultados con los resultados obtenidos mediante la simulación estática de la cache (SCS/ *Static Cache Simulation*) [134].

Con el objeto de resolver el problema de las interferencias extrínsecas de cache, en la literatura se han propuesto métodos para medirlas. También se ha propuesto la utilización de caches que puedan fijar su contenido durante toda la ejecución del sistema o durante algunos periodos concretos de la misma. En el Capítulo 4 presentamos un algoritmo *Lock-MS* (*Lock for Maximize Schedulability*) para seleccionar la líneas a fijar en la cache. Para una jerarquía de memoria formada por un almacén de línea (LB/ *Line Buffer*) y una cache que puede fijar su contenido (*Lockable iCache*), el algoritmo *Lock-MS* consigue maximizar la *planificabilidad* del sistema. En el Capítulo 5 añadimos a la jerarquía de memoria una prebúsqueda secuencial que reduce considerablemente el WCET de las tareas mejorando aún más la *planificabilidad* del sistema.

Además, tanto en el Capítulo 4, como en el Capítulo 5 comparamos los resultados conseguidos mediante el algoritmo *Lock-MS*, cuando se permite actualizar el contenido de la cache en los cambios de contexto, con los resultados obtenidos mediante el algoritmo *Lock-MU* (*Algorithm for Minimize Utilization*) cuando se fija la cache durante toda la ejecución del sistema [147].

Capítulo 3

El WCET con caches en *caminos relevantes*

Un sistema de tiempo real está formado por un conjunto de tareas que cooperan con el fin de conseguir un objetivo. Para garantizar que las tareas se ejecuten siempre en un determinado plazo de tiempo, es necesario definir algoritmos o políticas de planificación que determinan si las restricciones temporales del sistema se pueden satisfacer. Por lo tanto, antes de poner en funcionamiento un sistema de tiempo real es necesario realizar un análisis de *planificabilidad* que tenga en cuenta: las tareas del sistema, sus plazos de finalización, sus periodos de ejecución y su tiempo de ejecución en el peor caso. Es decir, cualquier análisis de *planificabilidad* depende del WCET de cada una de las tareas del sistema.

En este capítulo se describe una técnica de análisis para determinar de forma exacta el coste de los accesos a memoria en presencia de una cache de instrucciones con algoritmo de reemplazo LRU (*Least Recently Used*). El algoritmo de reemplazo LRU es totalmente determinista, por lo tanto está perfectamente indicado en un sistema de tiempo real, ya que evita cualquier incertidumbre respecto al contenido de la cache durante la ejecución de la tarea [154]. La base de esta nueva propuesta reside en analizar todos los caminos de ejecución que pueden alcanzar el WCET, descartando, sin pérdida de información, todos aquellos caminos sobre los que durante el análisis se pueda asegurar que en ningún caso determinarán el WCET. En los experimentos realizados se ha verificado que mediante esta propuesta se reduce espectacularmente el número de caminos que se deben analizar de principio a fin para determinar el WCET del programa. Así pues, en presencia de una cache de instrucciones *convencional*, si el número de caminos condicionales dentro de los bucles en un programa no es demasiado grande, es posible computacionalmente analizar de forma exacta el coste de los accesos a memoria en el peor caso.

3.1. Caminos de ejecución en bucles con condicionales

Determinar la contribución exacta al WCET de los accesos a memoria, en presencia de una cache de instrucciones, es muy costoso debido a la complejidad exponencial del problema, ya que es necesario analizar todos y cada uno de los posibles caminos de ejecución del programa. Por ejemplo, en un bucle de 100 iteraciones que contenga un condicional con dos caminos se generan 2^{100} caminos de ejecución diferentes. Puesto que el contenido de la cache depende del camino seguido durante la ejecución, conocer el contenido de la cache en cada instante hace que el problema sea demasiado complejo computacionalmente, tanto en tiempo, como en espacio.

En la Figura 3.1 se muestra con un ejemplo sencillo la dificultad que conlleva calcular el WCET exacto de un bucle con un condicional. Teniendo en cuenta el comportamiento de una cache de instrucciones, en la Tabla 3.1 se indica el tiempo de ejecución asociado a los caminos A y B que forman el bucle de la Figura 3.1 a).

Casos de Ejecución	**Camino A**	**Camino B**
Primera ejecución	30	40
Ejecuciones alternativas	20	28
Dos ejecuciones consecutivas	10	14

Tabla 3.1: Coste de ejecución con una cache de instrucciones.

La primera fila de la Tabla 3.1 indica el coste de la primera ejecución de cada uno de los caminos cuando la cache está vacía. En la segunda fila se muestra el coste de ejecución de cada uno de los caminos cuando en la iteración anterior del bucle se ejecutó el otro camino, es decir, primero se ejecuta el camino A y después el B o primero se ejecuta el camino B y después el A. En la tercera fila se indica el coste de la ejecución de cada uno de los caminos cuando en la iteración anterior del bucle se ejecutó ese mismo camino.

Una vez clasificados todos los accesos a memoria de las instrucciones, se determina el tiempo de ejecución de cada bloque básico, y finalmente se calcula el WCET del programa, por ejemplo a partir del árbol sintáctico que se puede obtener durante el proceso de compilación.

La Figura 3.1 b) muestra el cálculo del WCET en presencia de una cache de instrucciones donde sólo aprovechamos su localidad espacial, es decir, suponemos que no sabemos predecir la localidad temporal. Finalmente, en la Figura 3.1 c) se indica el cálculo exacto del WCET considerando todos los caminos posibles de ejecución, aunque sólo se muestran las 4 primeras iteraciones del bucle.

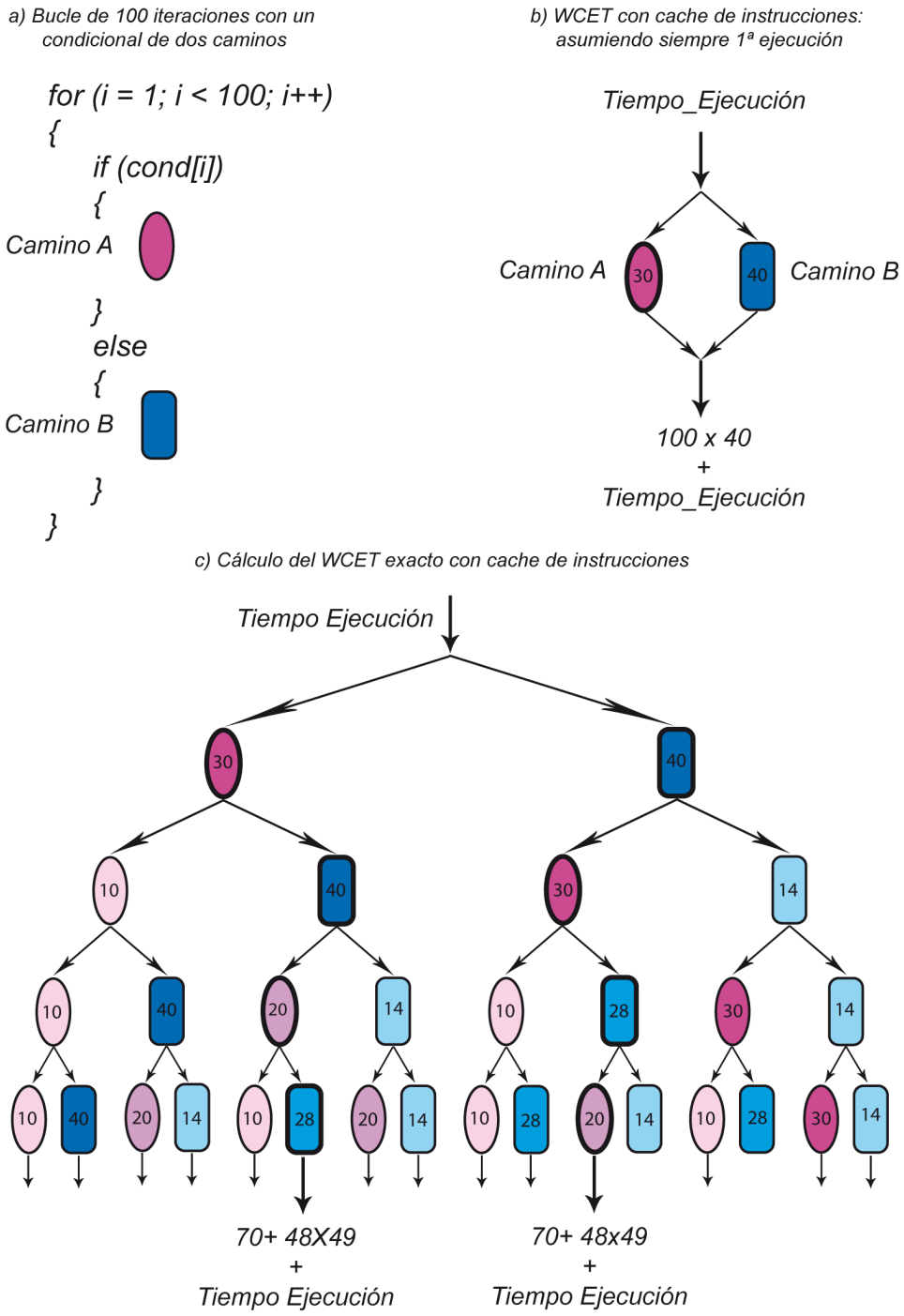


Figura 3.1: Cálculo del WCET en un bucle con un condicional.

La diferencia en el WCET obtenido es considerable, por lo tanto no es una buena estrategia simplificar excesivamente dicho análisis y mucho menos descartar la utilización de caches en sistemas de tiempo real.

En la siguiente sección se demuestra que, para determinar la contribución exacta al WCET de los accesos a memoria, en presencia de una cache de instrucciones *convencional*, el número de caminos que se deben analizar está acotado, y en general es mucho más pequeño que el número de posibles caminos de ejecución de la tarea. Esto permite realizar un análisis exacto del comportamiento, en el peor caso, de la cache de instrucciones, siempre y cuando el número de caminos que deben ser analizados no sea excesivamente grande.

3.2. Número máximo de *caminos relevantes*

Como aparece en la Figura 3.1 c), durante el análisis del WCET se produce una explosión combinatoria de los posibles caminos de ejecución que puede seguir un programa debido a las estructuras condicionales dentro de bucles. Por lo tanto, es necesario acotar el número de caminos posibles de ejecución de la tarea en estas estructuras de programación. Por ejemplo, en un bucle de n iteraciones con un condicional, el número de caminos posibles de ejecución es p^n donde p representa el número de caminos del condicional. No obstante, si se analizan los diferentes estados de cache que se pueden alcanzar durante la ejecución de todos estos caminos, es decir, si se analiza el contenido de la cache, se observa que muchos de estos estados son iguales. Por lo tanto, para determinar el coste de los accesos a memoria de la tarea sólo será necesario considerar aquellos estados de cache que sean diferentes. Además, para calcular el WCET de un bucle será suficiente considerar los caminos con el mayor tiempo de ejecución acumulado para cada uno de los estados diferentes de cache alcanzados durante el análisis. Así pues, se trata de analizar únicamente los *caminos relevantes*, que son aquellos caminos que acumulan el mayor tiempo de ejecución para cada estado diferente de cache alcanzado hasta un determinado punto del programa.

En esta sección se demuestra que, en presencia de una cache de instrucciones con algoritmo de reemplazo LRU, el número máximo de *caminos relevantes* en un bucle con un condicional está acotado por la siguiente expresión, en la que n es el número de iteraciones del bucle y p representa el número de caminos de la estructura condicional.

$$\sum_{i=1}^{min(p,n)} P_p^i = \sum_{i=1}^{min(p,n)} \frac{p!}{(p-i)!} \tag{3.1}$$

En general, el número de iteraciones de un bucle suele ser mucho mayor que el número de caminos de las estructuras condicionales y, aunque la Ecuación 3.1

sigue teniendo carácter exponencial, el número de *caminos relevantes* es mucho menor que el número de posibles caminos de ejecución. Por ejemplo, en un bucle de n iteraciones, que tenga un condicional con 2 caminos, el número de caminos posibles de ejecución es 2^n, pero el número máximo de estados diferentes de cache, es decir, el número máximo de *caminos relevantes* es 4.

Para determinar el número máximo de estados de cache de una tarea se propone una cache de instrucciones con algoritmo de reemplazo LRU, donde el contenido de la cache depende del orden en el que se realizan los accesos a memoria. Además, sin pérdida de generalidad, se considera una cache totalmente asociativa, para evitar los fallos por conflicto, y de capacidad infinita para evitar los fallos por capacidad. Esta organización de la cache genera el número máximo de estados de cache durante el análisis de la tarea, ya que cualquier limitación, tanto en la capacidad, como en la asociatividad de la cache reduce el número de estados diferentes de cache que se pueden alcanzar. Se trata en definitiva de considerar la organización de cache que genera el mayor número de *caminos relevantes* durante el análisis del WCET de la tarea.

Para la demostración formal de la Ecuación 3.1, es decir, que el número máximo de *caminos relevantes* durante el análisis del WCET de un programa está acotado, se consideran las definiciones, proposiciones y reglas que se detallan a continuación.

Definición 1. *(OSB/ Ordered Set of Blocks)*
Se define el conjunto ordenado de bloques OSB como el conjunto de bloques de memoria ordenados a los que se accederá durante la ejecución de un segmento o camino del programa.

Es decir, dado un segmento o camino A de un programa, y $A_@$ la secuencia ordenada de accesos a memoria que se generan durante la ejecución de A, el OSB_A es el conjunto ordenado de bloques de memoria distintos a los que se accede al considerar dicha secuencia ordenada de accesos a memoria. El orden de un bloque de memoria dentro de un conjunto OSB depende únicamente del instante en el que se accedió a dicho bloque por última vez, por lo tanto, dicho orden es equivalente al orden establecido en una cache con política de reemplazo LRU, como ya se ha indicado.

Por ejemplo, sea $A_@$ una secuencia de accesos a memoria del camino A, formada por las direcciones: a_{i1}, a_{i2}, a_{i3}, a_{j1}, a_{j2}, a_{j3}, a_{j4}, a_{k1}, a_{k2}, a_{k3}, a_{k4} que pertenecen a los bloques b_i, b_j y b_k respectivamente con i, j, k distintos. El conjunto OSB_A asociado a dicha secuencia es el conjunto ordenado de bloques $\{b_i, b_j, b_k\}$. Si durante la ejecución del camino A, suponemos ahora que se accede a las direcciones: a_{i1}, a_{i2}, a_{i3}, a_{j1}, a_{j2}, a_{j3}, a_{j4}, a_{k1}, a_{k2}, a_{k3}, a_{k4}, a_{j3}, a_{j4}, que pertenecen a los bloques b_i, b_j, b_k y b_j respectivamente con i, j, k distintos. El conjunto OSB_A asociado a dicha secuencia será el conjunto ordenado

de bloques $\{b_i, b_k, b_j\}$.

Las siguientes propiedades se deducen trivialmente a partir de la definición de OSB como un conjunto ordenado de elementos, que en este caso son bloques de memoria.

Propiedad 1. $OSB_A \cup OSB_A = OSB_A$

Propiedad 2. $OSB_A \cup OSB_B \neq OSB_B \cup OSB_A$

Propiedad 3. $OSB_A \cup OSB_B \cup OSB_A = OSB_B \cup OSB_A$

Para simplificar la notación, y siempre que dos secuencias de accesos $A_@$ y $B_@$ sean distintas, se denota $OSB_A \cup OSB_B$ como OSB_{AB}.

Definición 2. *(CS/ Cache State)*
Sea $A_@$ una secuencia de accesos a memoria asociada al camino de ejecución A de un programa, CS(A) representa el estado de cache que se alcanzará al considerar la secuencia de accesos $A_@$.

Definición 3. *Sea OSB_A el conjunto ordenado de bloques de memoria asociado a la secuencia de accesos $A_@$, CS(OSB_A) representa el estado de cache que se alcanzará al realizar el acceso ordenado a los bloques de memoria del conjunto OSB_A.*

Proposición 1. *CS(A) = CS(OSB_A)*
Sea la secuencia de accesos a memoria $A_@$ asociada a un camino de ejecución A de un programa, y el conjunto OSB_A asociado a dicha secuencia de accesos, entonces el CS(A) será igual al CS(OSB_A).

Demostración. La igualdad se deduce de forma trivial de las Definiciones 1, 2, 3, de la construcción de los conjuntos OSB y del algoritmo de reemplazo LRU utilizado por la cache.

Tanto el CS(OSB_A), como el CS(A), están formados por los mismos bloques de memoria y la vejez asociada a cada línea de cache coincidirá con el orden de los bloques que forman el conjunto OSB. □

Proposición 2. *CS(A, B) = CS(OSB_{AB})*
El estado de cache que se alcanza al realizar primero la secuencia de accesos $A_@$ y después la secuencia de accesos $B_@$, es igual al CS(OSB_{AB}).

Demostración. Sean $A_@$ y $B_@$ secuencias de acceso a memoria distintas asociadas a un camino de ejecución de un programa. Se consideran los conjuntos OSB_A y OSB_B, asociados a dichas secuencias. El conjunto OSB asociado a la realización de, primero la secuencia $A_@$ y después la secuencia $B_@$, vendrá determinado por $OSB_A \cup OSB_B$.

Por lo tanto, CS(A, B) = CS($OSB_A \cup OSB_B$) = CS(OSB_{AB}). □

Se establecen a continuación algunas reglas generales para la obtención de los diferentes estados de cache cuando se combinan distintas secuencias de accesos a memoria.

Regla 1. *CS(A, A) = CS(A)*
El estado de cache que se alcanza al realizar dos veces consecutivas una secuencia de accesos $A_@$, es igual al estado de cache que se consigue al realizar dicha secuencia de accesos $A_@$ una sola vez.

Demostración. Es consecuencia de la Proposición 1, de la propia definición del conjunto OSB y de sus propiedades demostradas anteriormente.
$\mathrm{CS}(A, A) = \mathrm{CS}(OSB_{AA})$
$\mathrm{CS}(OSB_{AA}) = \mathrm{CS}(OSB_A)$
$\mathrm{CS}(OSB_A) = \mathrm{CS}(A)$ □

Regla 2. *CS(A, $\cdots$ n) $\cdots$, A) = CS(A)*
El estado de cache, que se alcanza al realizar n veces una secuencia de accesos $A_@$, es igual al estado de cache que se consigue al realizar dicha secuencia de accesos una sola vez.

Demostración. Por inducción aplicando la Regla 1. □

Regla 3. *CS(A, B) $\neq$ CS(B, A)*
En general, el estado de cache que se alcanza al realizar primero la secuencia de accesos $A_@$ y después la secuencia $B_@$, es distinto al estado de cache que se consigue al realizar primero la secuencia de accesos $B_@$ y después la secuencia $A_@$.

Demostración. Se deduce de la Proposición 2 y de las propiedades de los conjuntos OSB. □

Regla 4. *CS(A, $\cdots$ n) $\cdots$, A, B, $\cdots$ m) $\cdots$, B) = CS(A, B)*
El estado de cache que se alcanza al realizar n veces la secuencia de accesos $A_@$, seguida de realizar m veces la secuencia de accesos $B_@$, es igual al estado de cache que se consigue al realizar la secuencia de accesos $A_@$ y después la secuencia de accesos $B_@$.

Demostración. Aplicando la Proposición 2 tenemos que
$\mathrm{CS}(A, \cdots n) \cdots, A, B, \cdots m) \cdots, B) = \mathrm{CS}(OSB_{A \cdots n) \cdots AB \cdots m) \cdots B})$
Aplicando inducción y la Propiedad 1 obtendremos que
$\mathrm{CS}(OSB_{A \cdots n) \cdots AB \cdots m) \cdots B}) = \mathrm{CS}(OSB_{AB})$
Finalmente, por la Proposición 2 podemos concluir que
$\mathrm{CS}(OSB_{AB}) = \mathrm{CS}(\mathrm{A}, \mathrm{B})$ □

Regla 5. *CS(A, $\cdots$ i) $\cdots$, A, B, $\cdots$ j) $\cdots$, B, A, $\cdots$ k) $\cdots$, A) = CS(B, A)*
El estado de cache que se alcanza al realizar i veces la secuencia de accesos $A_@$, seguida de j veces la secuencia de accesos $B_@$ y seguida de k veces la secuencia de accesos $A_@$, es igual al estado de cache que se consigue al realizar la secuencia de accesos $B_@$ y después la secuencia de accesos $A_@$.

Demostración. Aplicando la Proposición 2 obtendremos que
$\mathrm{CS}(A, \cdots i) \cdots , A, B, \cdots j) \cdots , B, A, \cdots k) \cdots , A) =$
$\mathrm{CS}(OSB_{A\cdots i)\cdots AB\cdots j)\cdots BA\cdots k)\cdots A})$
Aplicando inducción y la Propiedad 1 de los conjuntos OSB tenemos que
$CS(OSB_{A\cdots i)\cdots AB\cdots j)\cdots BA\cdots k)\cdots A}) = CS(OSB_{ABA})$
Aplicando la Propiedad 3 de los conjuntos OSB obtendremos que
$\mathrm{CS}(OSB_{ABA}) = \mathrm{CS}(OSB_{BA})$
Por último, aplicando la Proposición 2 concluimos que
$CS(OSB_{BA}) = CS(B, A)$ □

Proposición 3. *Dada una estructura condicional, con p caminos de ejecución alternativos, dentro de un bucle L de n iteraciones, el número máximo de estados distintos de cache que se pueden alcanzar al final del bucle está acotado por la siguiente expresión matemática:*

$$\sum_{i=1}^{min(p,n)} P_p^i = \sum_{i=1}^{min(p,n)} \frac{p!}{(p-i)!}$$

Demostración. Sin pérdida de generalidad, se supone que $p < n$ y se consideran las p primeras iteraciones del bucle L. Sea $i = 1, ..., p$ un camino dentro del bucle L, y sea $i_@$ la secuencia de accesos generada por dicho camino.

Para la primera iteración sólo es posible realizar la secuencia de accesos $i_@$ con $i = 1, ..., p$, una sola vez.

Por lo tanto, el número de estados de cache $\mathrm{CS}(i_@)$ que podemos alcanzar es P_p^1.

En la segunda iteración se puede realizar cualquier combinación de secuencias de accesos $i_@$ seguida de $j_@$ con $i, j = 1, ..., p$ cualesquiera.

Para cada pareja de índices i, j iguales ($i = j$), el número de estados de cache $\mathrm{CS}(i_@)$ que podemos alcanzar si aplicamos la Regla 2 es P_p^1.
Para cada pareja de índices i, j distintos ($i \neq j$), el número de estados de cache $\mathrm{CS}(i_@, j_@)$ y $\mathrm{CS}(j_@, i_@)$ que podemos alcanzar si aplicamos la Regla 3 es P_p^2.

Por lo tanto, en la segunda iteración podemos alcanzar $\sum_{i=1}^{2} P_2^i$ estados de cache diferentes.

En la tercera iteración se puede realizar cualquier combinación de secuencias de accesos $i_@$ seguida de $j_@$ y seguida de $k_@$ con i, j, $k = 1, ..., p$ cualesquiera.

Para cada terna de índices i, j, k iguales ($i = j = k$), el número de estados de cache CS($i_@$) que podemos alcanzar si aplicamos la Regla 2 es P_p^1.

Para cada terna de índices i, j, k con $i \neq j$ y $k = i$ o $k = j$ tenemos que:
Si $k = i$, el número de estados de cache que podemos alcanzar teniendo en cuenta la Regla 5 es igual al logrado por las secuencias de accesos $j_@$ seguido de la secuencia de accesos $i_@$, es decir CS($j_@$, $i_@$).
Si $k = j$, el número de estados de cache que podemos alcanzar teniendo en cuenta la Regla 5 es igual al logrado por las secuencias de accesos $i_@$ seguido de la secuencia de accesos $j_@$, es decir CS($i_@$, $j_@$).
Por lo tanto, para cada terna de índices i, j, k con $i \neq j$ y $k = i$ o $k = j$ podemos alcanzar P_p^2 estados de cache diferentes.

Y para la terna de índices i, j, k con $i \neq j \neq k$, el número de estados diferentes de cache CS($i_@$, $j_@$, $k_@$) que podemos alcanzar si aplicamos la Regla 3 es P_p^3.

Por lo tanto en la tercera iteración podemos lograr $\sum_{i=1}^{3} P_3^i$ estados de cache diferentes.

Finalmente, si aplicamos este razonamiento de forma inductiva sobre el número p de iteraciones del bucle, tenemos que el número de estados diferentes de cache que podemos alcanzar es $\sum_{i=1}^{p} P_p^i$.
No obstante, si $n < p$, el número de estados diferentes de cache que podemos conseguir será $\sum_{i=1}^{n} P_p^i$.
Por lo tanto, el número de estados diferentes de cache que podemos alcanzar es:

$$\sum_{i=1}^{min(p,n)} P_p^i = \sum_{i=1}^{min(p,n)} \frac{p!}{(p-i)!}$$

□

En la Figura 3.2, a escala logarítmica, se muestra una comparativa entre el número de caminos posibles de ejecución y el número de estados diferentes de cache en función del número de iteraciones de un bucle y del número de caminos alternativos dentro del mismo. Como se puede observar, el número de estados de cache siempre es menor, en todas y cada una de las iteraciones, que el número de caminos posibles de ejecución. No obstante, el número de estados de cache diferentes inicialmente crece de forma exponencial, pero permanece constante cuando alcanza el máximo teórico propuesto.

Así pues, si se consideran todos los estados diferentes de cache, se puede determinar el coste exacto de los accesos a memoria durante el análisis del WCET

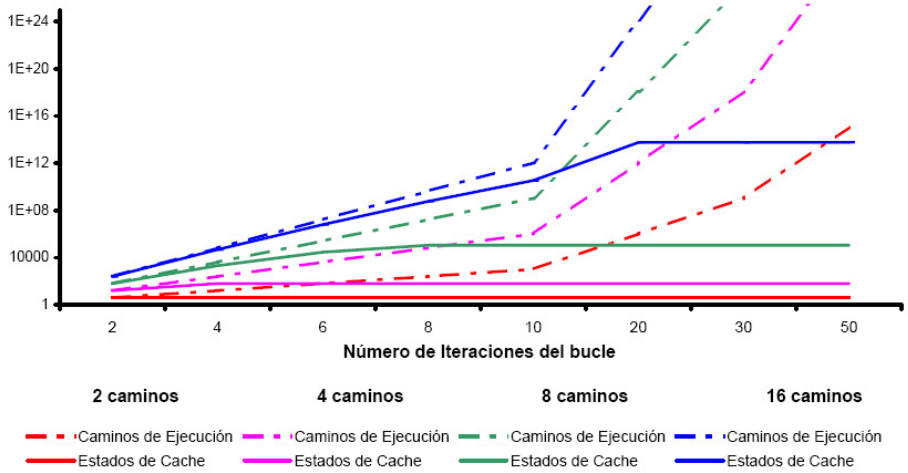


Figura 3.2: Número de caminos posibles de ejecución vs. Número de estados diferentes de cache.

de una tarea, evitando en gran medida la explosión combinatoria de los posibles caminos de ejecución. Para ello, en determinadas instrucciones comunes a dos o más caminos, se deben seleccionar los caminos con mayor tiempo de ejecución acumulado para cada uno de los diferentes estados de cache alcanzados. El resto de caminos se puede *podar* o descartar del análisis sin pérdida de información. A las instrucciones comunes a dos o más caminos donde se pueden eliminar del análisis las ramas o caminos que no determinan el WCET del programa, las denominamos *instrucciones poda*. A partir de estas *instrucciones poda*, cada uno de los caminos se debe analizar por separado hasta alcanzar de nuevo otra *instrucción poda*.

Corolario 1. *Para determinar el coste exacto de los accesos a memoria, durante el análisis del WCET de una tarea, sólo es necesario analizar los* caminos relevantes.

Corolario 2. *Dada una estructura condicional, con p caminos de ejecución alternativos, dentro de un bucle L de n iteraciones, el número máximo de estados diferentes de cache que se pueden alcanzar al final del bucle está acotado por la siguiente expresión:*

$$\sum_{i=1}^{\infty} P_p^i = e \cdot p!$$

Es decir, el número de estados diferentes de cache que se pueden alcanzar en un bucle que contiene p caminos de ejecución diferentes no depende del número

de iteraciones del bucle.

Además, para determinar el WCET del programa no es necesario analizar todas la iteraciones del bucle L, es suficiente con analizar las $\lceil log_e(p!) \rceil + p$ primeras iteraciones del bucle.

Corolario 3. *Dada una estructura condicional, con p caminos de ejecución diferentes, dentro de un bucle L de n iteraciones, el número máximo de estados distintos de cache que se pueden alcanzar para una cache asociativa por conjuntos está acotado por:*

$$\sum_{i=1}^{min(p,n)} P_p^i$$

Es decir, el número de estados diferentes de cache que se pueden conseguir, en un bucle que contiene p caminos de ejecución diferentes, no depende del grado de asociatividad de la cache.

Corolario 4. *Dada una estructura condicional, con p caminos de ejecución alternativos, dentro de un bucle L_n de n iteraciones, donde a su vez el bucle L_n está incluido en otro bucle L_m de m iteraciones, el número máximo de estados diferentes de cache que se pueden alcanzar en este bucle anidado está acotado por la siguiente expresión:*

$$\sum_{i=1}^{min(p,\ n \cdot m)} P_p^i$$

Es decir, los resultados de los corolarios anteriores se pueden aplicar directamente en bucles anidados.

Corolario 5. *Dados N estados diferentes de cache, si durante el análisis del WCET alcanzamos un bucle de m iteraciones que contiene una estructura condicional con p caminos de ejecución alternativos, el número máximo de estados diferentes de cache que se pueden lograr al final del bucle está acotado por la siguiente expresión:*

$$N \cdot \sum_{i=1}^{min(p,m)} P_p^i$$

3.3. Contribución exacta de los accesos a memoria al WCET

En esta sección se describe el funcionamiento de un nueva técnica de análisis y cálculo del WCET que permite calcular la contribución exacta de los accesos a

memoria de una tarea en presencia de una cache de instrucciones con algoritmo de reemplazo LRU. Esta técnica explora todos los caminos posibles de ejecución de una tarea podando o descartando, de forma segura y sin perder información, los caminos de ejecución que no puedan determinar en ningún caso el WCET.

Para describir esta técnica de análisis se considera un procesador sencillo que dispone de una cache de instrucciones. Es decir, el procesador no es segmentado, no dispone de predictor de saltos ni de cache de datos. La técnica predice el coste de *fetch* de cada una de las instrucciones de todos los caminos de ejecución que pueden alcanzar el WCET del programa. Esta propuesta también permite estudiar cómo afecta al WCET un fallo en la predicción del coste de un acceso a una instrucción en el *camino más largo*. Además, para evitar cualquier tipo de pérdida de información durante el análisis, se han integrado el análisis de flujo de control, el comportamiento del procesador y el propio cálculo del WCET de todos los *caminos relevantes*. El WCET obtenido, en este procesador sencillo, es exacto y se determina en un único paso, es decir, de forma integral.

Esta técnica de análisis basa su funcionamiento en seguir el flujo de control del programa, decodificando en secuencia las instrucciones y siguiendo el camino indicado por los saltos obligatorios. Sin embargo, el programador debe añadir las indicaciones necesarias para tratar de la forma más adecuada y correcta las instrucciones de salto condicional. Por ejemplo, debe indicar el número de iteraciones cuando el condicional está asociado a un bucle, y también especificar el descarte del análisis de una de las ramas del condicional, cuando representa un camino imposible[1]. Si en una instrucción de salto condicional no hay ningún tipo de anotación, el análisis se divide, ya que debemos analizar todos los caminos posibles de ejecución. A partir de ese instante, se estudian los dos posibles caminos de ejecución por separado de forma totalmente independiente. Pero, como ya se ha mostrado anteriormente, si la instrucción condicional está dentro de un bucle, en unas cuantas iteraciones, el análisis será inviable ya que el número de caminos posibles de ejecución crece exponencialmente.

Para evitar la complejidad exponencial del análisis debemos aplicar la teoría desarrollada en la sección anterior (Sección 3.2). Así pues, para analizar el comportamiento de la cache de instrucciones también se debe guardar el estado de cache asociado a cada uno de los posibles caminos de ejecución. Además, antes de iniciar el análisis, es necesario marcar las *instrucciones poda* para detener dicho análisis y comparar los estados de cache de todos los caminos que alcanzan estas *instrucciones poda*. Si en una *instrucción poda* dos o más caminos tienen el mismo estado de cache, se podan o descartan todos aquellos caminos que acumulen el menor tiempo de ejecución hasta ese instante. Estos caminos no son *relevantes*, ya que en ningún caso pueden llegar a determinar el WCET. Para cada estado diferente de cache, sólo el camino con mayor tiempo de ejecución

[1]Esta técnica permite descartar del análisis los caminos imposibles añadiendo las anotaciones oportunas en las estructuras condicionales, pero no los puede reconocer.

acumulado puede alcanzar el WCET del programa. Como la arquitectura del procesador considerada es sencilla, sin perdida de información podemos aplicar el Corolario 1. Es decir, para determinar el coste exacto de los accesos a memoria sólo es necesario analizar los *caminos relevantes*.

Como ejemplo, en la Figura 3.3 se muestra el código en lenguaje C de un programa y su código ensamblador generado con GCC 2.95.2 -O2 para ARM v7. En el código ensamblador se han marcado los caminos de ejecución. En este caso, el programa puede seguir dos posible caminos durante la ejecución. También se han marcado las instrucciones de saltos condicionales y obligatorios. En particular, se ha marcado la instrucción condicional asociada al bucle indicando el número de iteraciones. Finalmente, el conjunto de posibles instrucciones poda más eficientes se ha indicado mediante llaves. Por lo tanto, modificando adecuadamente una herramienta de simulación, como por ejemplo SimpleScalar [12], para que siga las indicaciones marcadas por el programador, se puede conseguir el WCET exacto del programa.

Aunque no se ha definido una política para determinar la elección más adecuada de las *instrucciones poda*, y cualquier instrucción común a varios caminos podría ser marcada como tal, parece obvio que el coste computacional de esta técnica dinámica de poda depende, tanto del número de *instrucciones poda*, como de su efectividad. Puesto que el número de *caminos relevantes* en los bucles con condicionales está acotado, como se demostró en la Proposición 3, podar o eliminar durante el análisis los *caminos no relevantes* tiene especial importancia en estas estructuras de programación. Así pues, es necesario marcar como *instrucción poda* alguna de las instrucciones al final del bucle, para que en cada iteración podamos eliminar el mayor número de caminos posibles de ejecución y sólo se analicen los *caminos relevantes*. Por otro lado, cuantos más estados de cache sean comparados en las *instrucciones poda*, mayor número de *caminos no relevantes* pueden ser descartados. Por lo tanto, también es interesante marcar *instrucciones poda* al final de los bloques básicos que contengan muchas instrucciones, ya que todos los caminos que ejecuten estos bloques básicos alcanzarán estados de cache muy parecidos y la poda puede llegar a ser verdaderamente efectiva.

Si consideramos de nuevo el ejemplo representado en la Figura 3.1 a), se puede observar gráficamente en la Figura 3.4 el funcionamiento de la técnica de poda dinámica presentada. En la Figura 3.4 a) se representa el tiempo de ejecución acumulado hasta ese instante para cada uno de los posibles caminos de ejecución y su estados de cache asociados. En la Figura 3.4 b) se pone de manifiesto la efectividad de la poda a la hora de reducir el número de *caminos relevantes* durante el análisis. En ambas figuras se observa que el número máximo de caminos analizados en cualquier instante siempre se puede reducir a 4, cifra que coincide con la cota establecida en la Proposición 3.

También puede ser interesante marcar otras *instrucciones poda*, aunque la

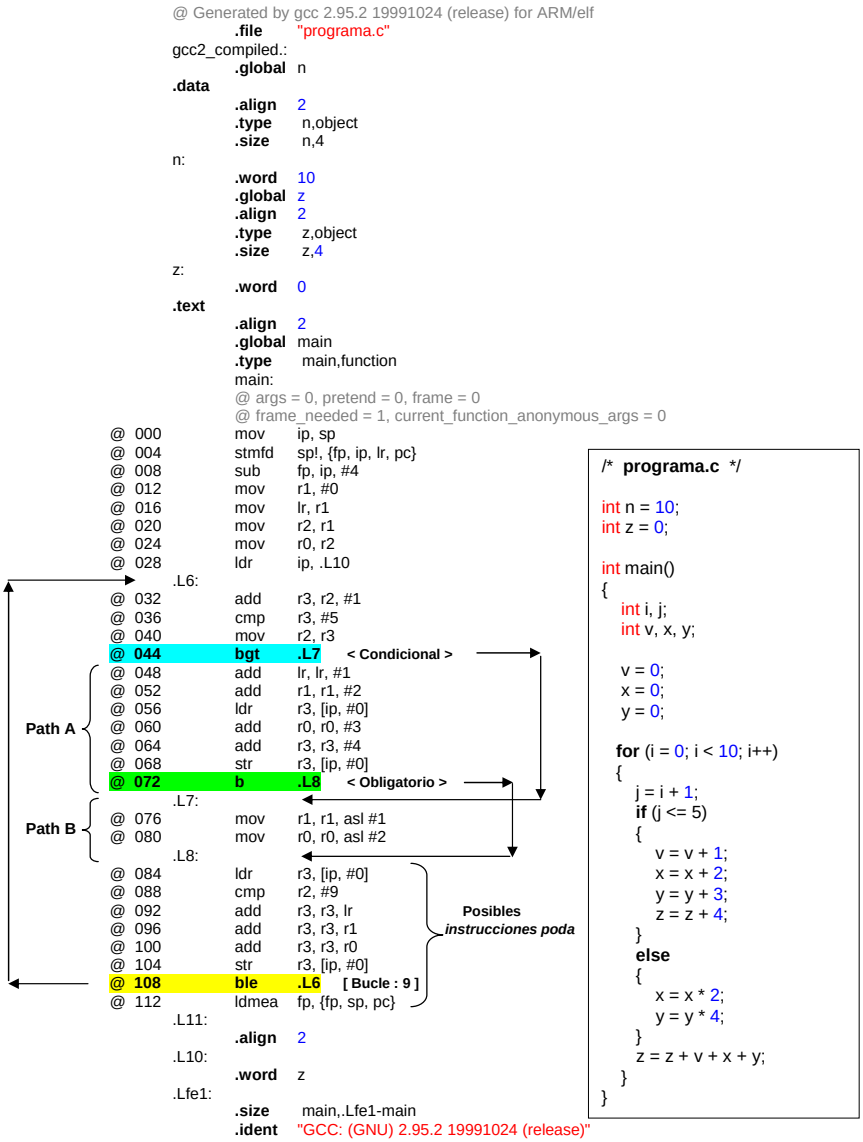


Figura 3.3: Marcado de las instrucciones más relevantes para la *poda dinámica de caminos*.

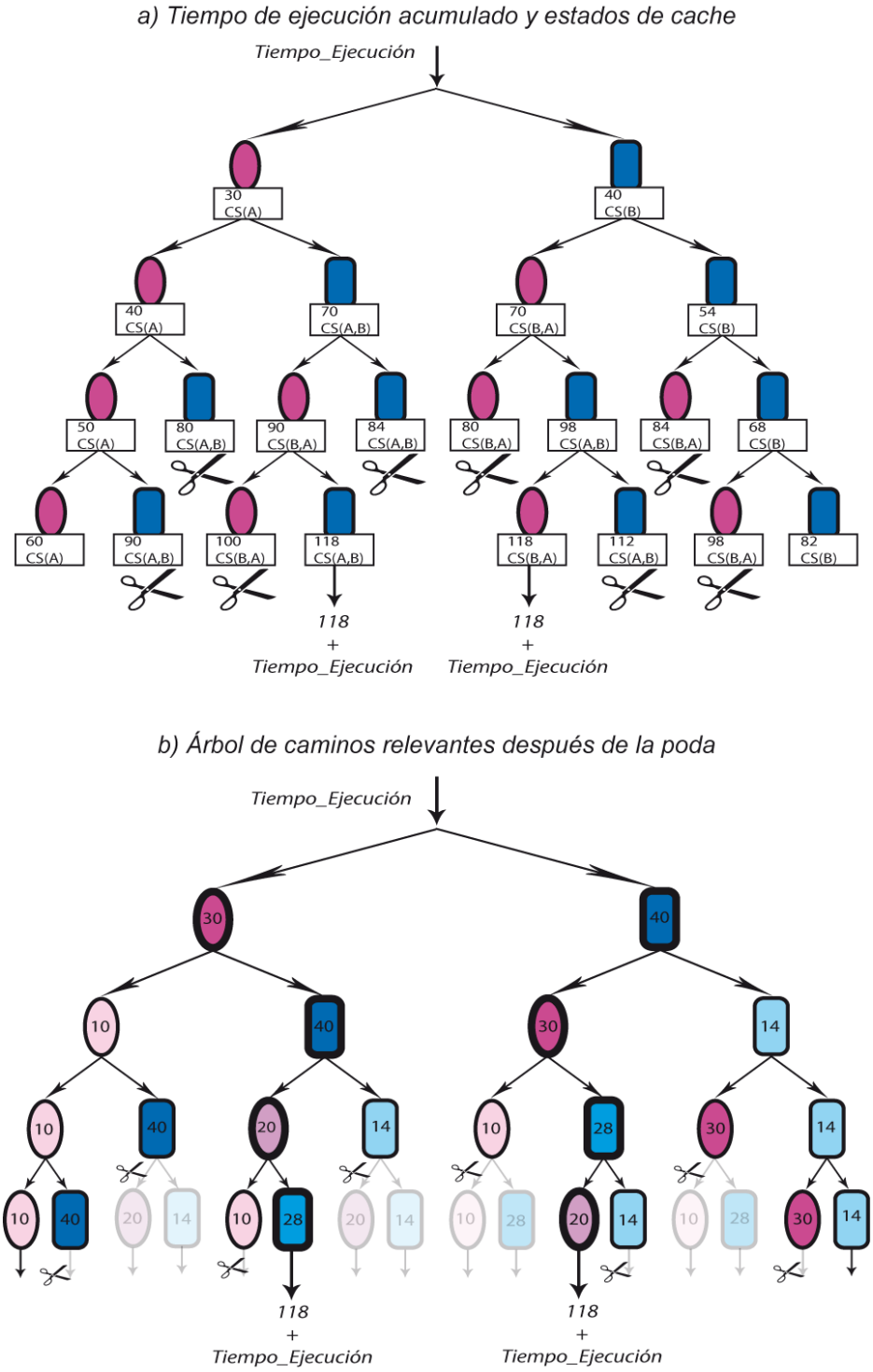


Figura 3.4: Ejemplo gráfico del cálculo preciso del WCET mediante la *poda dinámica de caminos* durante el análisis.

efectividad de la poda dependerá de la estructura del programa, de los condicionales dentro de bucles y de la organización particular de la cache de instrucciones considerada. La eficacia de la poda depende especialmente de la capacidad de la cache, del tamaño de sus líneas y de su asociatividad. Pero durante el análisis, aunque en los estados de cache sólo es necesario guardar las *tags* de los bloques de memoria que contiene la cache, si la capacidad de la cache que se analiza es grande, el tiempo dedicado a comparar los diferentes estados de cache en las *instrucciones poda* puede ser considerable. Si se dedica demasiado tiempo a comparar los estados de cache, el tiempo de análisis puede aumentar significativamente, de hecho puede suponer una de las principales deficiencias de esta técnica. Por lo tanto, el número de *instrucciones poda* no debería ser arbitrario, ya que influye decisivamente en el tiempo de análisis y cálculo del WCET.

Finalmente conviene indicar que todos aquellos *caminos relevantes* cuyo tiempo de ejecución acumulado, más el coste de cargar completamente la cache, sea inferior al tiempo acumulado por algún otro *camino relevante*, se pueden descartar del análisis, ya que en ningún caso determinan el WCET de la tarea. Esta nueva posibilidad de poda puede mejorar la eficiencia de la técnica cuando el número de iteraciones de los bucles sea muy grande. Por ejemplo, si suponemos que cargar completamente la cache en el análisis del WCET de la Figura 3.4 b) cuesta 50 unidades de tiempo, en la iteración número 3 se puede podar el camino cuyo estado de cache es CS(A) y en la iteración 4 se puede podar el camino con estado de cache CS(B) quedando reducido el análisis a la Figura 3.5. Si además aplicamos el Corolario 2, donde se indica el número mínimo de iteraciones que es necesario analizar para determinar el WCET de la tarea, se puede reducir el tiempo de análisis considerablemente.

En definitiva, el WCET obtenido mediante esta técnica de poda dinámica se puede utilizar para determinar la *planificabilidad* de un sistema de tiempo real. Sin embargo, en un sistema multitarea con expulsiones es necesario tener en cuenta las interferencias extrínsecas entre las diferentes tareas del sistema, y el coste de los cambios de contexto [14]. No obstante, el coste de las interferencias extrínsecas de cache se puede conseguir de diferentes formas, como se ha propuesto en trabajos de investigación anteriores [29, 32, 30, 100, 101, 139, 170, 171]. Por lo tanto, para una tarea $Task_i$, si se considera el WCET obtenido de forma aislada C_i, el coste de un cambio de contexto δ y el coste de las interferencias extrínsecas de cache γ, el nuevo WCET $C_i^{'}$ de la tarea queda determinado por la siguiente expresión:

$$C_i^{'} = C_i + 2\delta + \gamma$$

En la siguiente sección se verifica experimentalmente la viabilidad de la técnica de poda presentada en este apartado y se muestran los resultados más destacados obtenidos en los experimentos realizados.

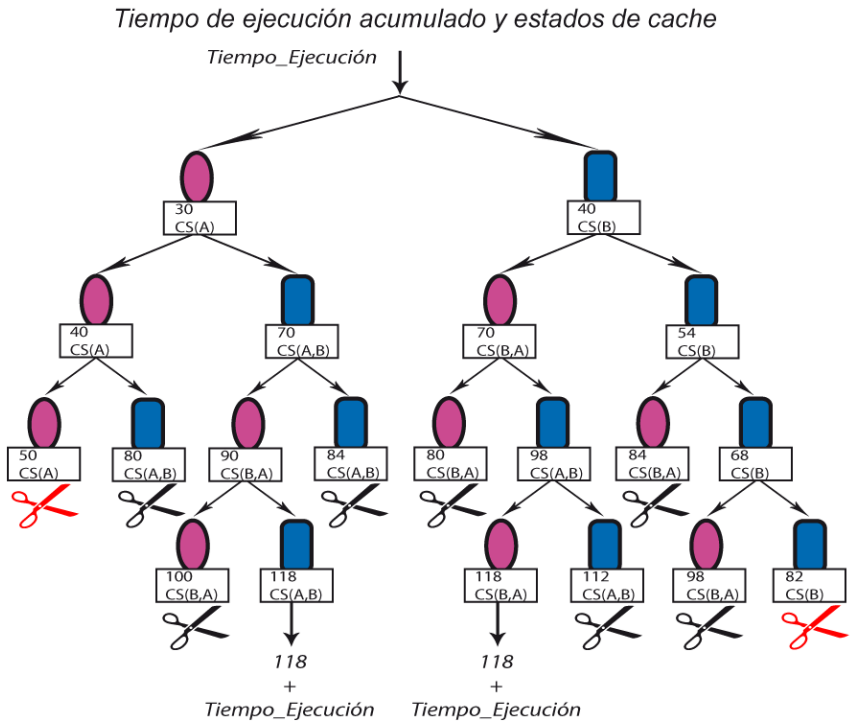


Figura 3.5: Cálculo preciso del WCET cuando el coste de cargar la cache es 50.

3.4. Resultados experimentales

Las tareas de prueba utilizadas en los experimentos se describen en la Tabla 3.2, donde también se muestra el tamaño del código y el número de instrucciones del *camino más largo*. Los códigos fuente se han compilado con GCC 2.95.2 -O2 para ARM v7. Estas tareas ya han sido utilizadas en estudios anteriores de referencia relacionados con el análisis del WCET [56, 116, 134, 146, 175].

Programa	Descripción	Tamaño	Instrucciones
array_sum	Suma elementos matriz	152 B	1 737
bs	Búsqueda binaria	112 B	56
bubble	Algoritmo bubble-sort	160 B	95 178
crc	Comp. redundancia cíclica	560 B	45 711
integral	Cálculo integral por intervalos	420 B	141 102
qurt	Raíces ecuación 2º grado	752 B	1 715

Tabla 3.2: Tareas con estructuras condicionales analizadas.

En este caso, se han seleccionado aquellas tareas que contienen sentencias

condicionales dentro de bucles. Algunas tareas utilizadas en otras propuestas como por ejemplo *jfdctint, matmult*, etc., se han descartado, puesto que sólo contienen un único camino de ejecución. Si una tarea no tiene condicionales, sólo puede seguir un camino durante la ejecución y este camino determina el WCET.

El objetivo de los experimentos realizados es determinar la contribución exacta de cada uno de los accesos a la memoria realizados en presencia de una cache de instrucciones y observar su influencia en el WCET. En este modelo, la etapa de *fetch* tiene un coste asociado de 1 ciclo cuando se produce un acierto, es decir, cuando la instrucción está en cache; pero tiene un coste asociado de 60 ciclos cuando se produce un fallo, es decir, cuando la instrucción no está en cache [166]. El coste de ejecución de una instrucción siempre es de 1 ciclo, salvo para las instrucciones de accesos a datos *load* y *store* que tienen un coste añadido de 60 ciclos, ya que se accede a la memoria principal para leer o guardar el dato.

En los experimentos realizados se ha variado el tamaño de la línea de memoria, la capacidad y la asociatividad de la cache. Se han considerado caches de instrucciones de capacidad 128, 256 y 512 bytes con tamaños de línea de 8, 16 y 32 bytes, es decir con capacidad para 2, 4 y 8 instrucciones por línea respectivamente. La pequeña capacidad de la cache está en consonancia con el reducido tamaño del conjunto de tareas analizadas. Con respecto a la asociatividad, considerando las potencias de 2 como número de vías, se han analizado todas las configuraciones de cache, desde correspondencia directa hasta totalmente asociativa.

Análisis exacto del WCET con caches

En el Tabla 3.3 se resumen los principales resultados obtenidos en los experimentos realizados. Los resultados muestran la gran diferencia entre el número de posibles caminos de ejecución de cada tarea analizada, que aparece en la tercera columna, y el número máximo de *caminos relevantes*, que se muestra en la cuarta columna. Para cada experimento realizado, aparece indicado el tiempo invertido en el análisis. Los experimentos se han efectuado en un Pentium 4 a $3,4$ GHz y, aunque se han analizado todas las iteraciones de los bucles de las tareas, el análisis ha finalizado en muy pocos segundos.

Determinar el WCET de todos los caminos posibles de ejecución es prácticamente imposible por la complejidad exponencial del problema, excepto para la tarea *bs* cuyo bucle sólo tiene 4 iteraciones. Además, si observamos el número máximo de *caminos relevantes* para las configuraciones de cache analizadas, se puede verificar que dicho número es mucho más pequeño que el límite teórico de la Proposición 3. Por lo tanto, determinar la contribución del *fetch* de instrucciones al WCET mediante la técnica de poda dinámica propuesta, para

Tareas	Itera. Bucles	Cam. Teóricos		Tam.	128 B Cache			256 B Cache			512 B Cache		
		Posib.	Relev.	Línea	Relev.		T.	Relev.		T.	Relev.		T.
					Min	Max	Análi	Min	Max	Análi	Min	Max	Análi
array_sum	100	$\approx 10^{30}$	3	8 B	2	3	0,07s	2	3	0,07s	2	3	0,09s
				16B	1	1	0,04	1	1	0,04s	1	1	0,04s
				32B	1	1	0,04s	1	1	0,04s	1	1	0,07s
bs	4	20	1	8	1	1	0,02s	1	1	0,02s	1	1	0,02s
				16B	1	1	0,02s	1	1	0,02s	1	1	0,02s
				32B	1	1	0,02s	1	1	0,02s	1	1	0,02s
bubble	5050	$\approx 10^{1\,520}$	9	8 B	3	8	0,33s	3	8	0,34s	3	8	0,35s
				16B	3	8	0,29s	3	8	0,29s	3	8	0,30s
				32B	3	6	0,22s	3	6	0,22s	3	6	0,22s
crc	2082	$\approx 10^{932}$	216	8 B	13	29	0,15s	66	114	0,25s	36	216	0,80s
				16B	6	6	0,13s	16	44	0,14s	16	81	0,51s
				32B	2	3	0,13s	4	4	0,14s	4	9	0,18s
integral	3000	$\approx 10^{2\,572}$	176	8 B	9	20	2,09s	20	58	5,55s	11	176	17,24s
				16B	6	10	1,22s	8	33	2,94s	6	83	5,29s
				32B	3	4	0,40s	4	9	0,42s	3	11	0,43s
qurt	60	$\approx 10^{44}$	553	8 B	7	16	0,05s	24	63	0,06s	79	469	0,25s
				16B	4	11	0,03s	19	24	0,04s	46	142	0,09s
				32B	4	6	0,03s	10	18	0,03s	27	84	0,07s

Tabla 3.3: Número de iteraciones, posibles caminos de ejecución y número máximo de *caminos relevantes*. Número mínimo y máximo de *caminos relevantes* obtenidos para cada configuración de cache, variando la asociatividad. Tiempo empleado en el análisis.

configuraciones particulares de cache, puede ser incluso mucho más efectivo de lo esperado. Por otro lado y de forma clara, también se observa que el número de *caminos relevantes* obtenidos durante el análisis depende, tanto de la capacidad de la propia cache, como del tamaño de las líneas de memoria. Por ejemplo, al aumentar la capacidad de la cache, el número de *caminos relevantes* crece de forma importante, obviamente hasta un cierto límite. Pero, al aumentar el tamaño de línea de cache, para una misma capacidad de cache, el número de *caminos relevantes* disminuye considerablemente. Así pues, en los experimentos realizados, el número máximo de *caminos relevantes* se obtiene para las caches con capacidad mayor y líneas de memoria más pequeñas, es decir, para caches de 512 bytes con tamaño de línea de 8 bytes.

Para medir la importancia de la asociatividad en esta técnica de poda, se ha fijado en 16 bytes el tamaño de bloque. Para cada tarea, en función de la capacidad de la cache y de la asociatividad, en las Figuras 3.6 y 3.7 se muestran el WCET y el número de *caminos relevantes*. En el eje Y de la izquierda aparece la escala asociada al WCET, y en el eje Y de la derecha se muestra la escala asociada al número de *caminos relevantes*. En general, se observa que la asociatividad no mejora claramente el WCET, pero sí aumenta considerablemente el número de *caminos relevantes*. Así pues, la asociatividad de la cache influye de forma negativa en el tiempo de análisis de esta técnica.

Finalmente conviene recordar que todos aquellos *caminos relevantes* cuyo tiempo de ejecución acumulado, más el coste de cargar completamente la cache, sea inferior al tiempo acumulado por algún otro *camino relevante*, se pueden descartar del análisis, ya que en ningún caso determinan el WCET de la tarea. Esta posibilidad de poda puede mejorar la eficiencia de la técnica cuando el número de iteraciones de los bucles sea muy grande. Además, en casos concretos, para reducir el tiempo de análisis, se pueden aplicar las conclusiones del Corolario 2 donde se indica el número mínimo de iteraciones que es necesario realizar para determinar el WCET de la tarea. Así pues, es interesante aplicar esta optimización durante el análisis de las tareas *buble, crc* e *integral*, donde el número de iteraciones de los bucles es grande. No obstante, como ya se ha comentado anteriormente, en los experimentos se han analizado todas las iteraciones de los bucles de cada tarea.

Contribución del *fetch* de instrucciones al WCET

En los experimentos realizados también se incluye una comparación con el método SCS (*Static Cache Simulation*) [134]. Esta técnica permite clasificar en el peor caso los accesos a la cache de instrucciones. No obstante, y aunque el WCET también depende del tiempo de ejecución de las instrucciones y de los accesos a datos, para que la comparación sea lo más justa posible sólo se considera el coste de acceso a las instrucciones. Además, también se analiza una cache asociativa de 2 vías, porque el método SCS obtiene las cotas del WCET

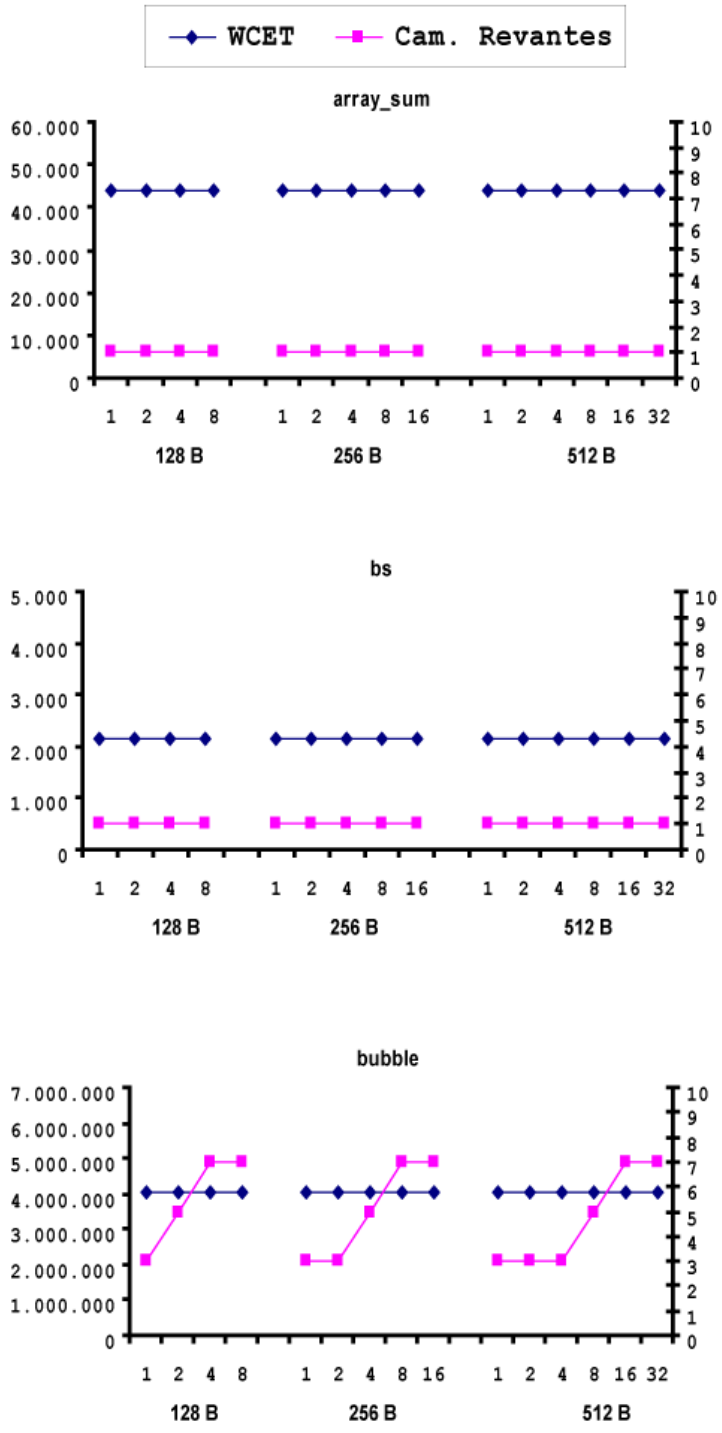


Figura 3.6: Eje Y de la izquierda: WCET. Eje Y de la derecha: *caminos relevantes*.

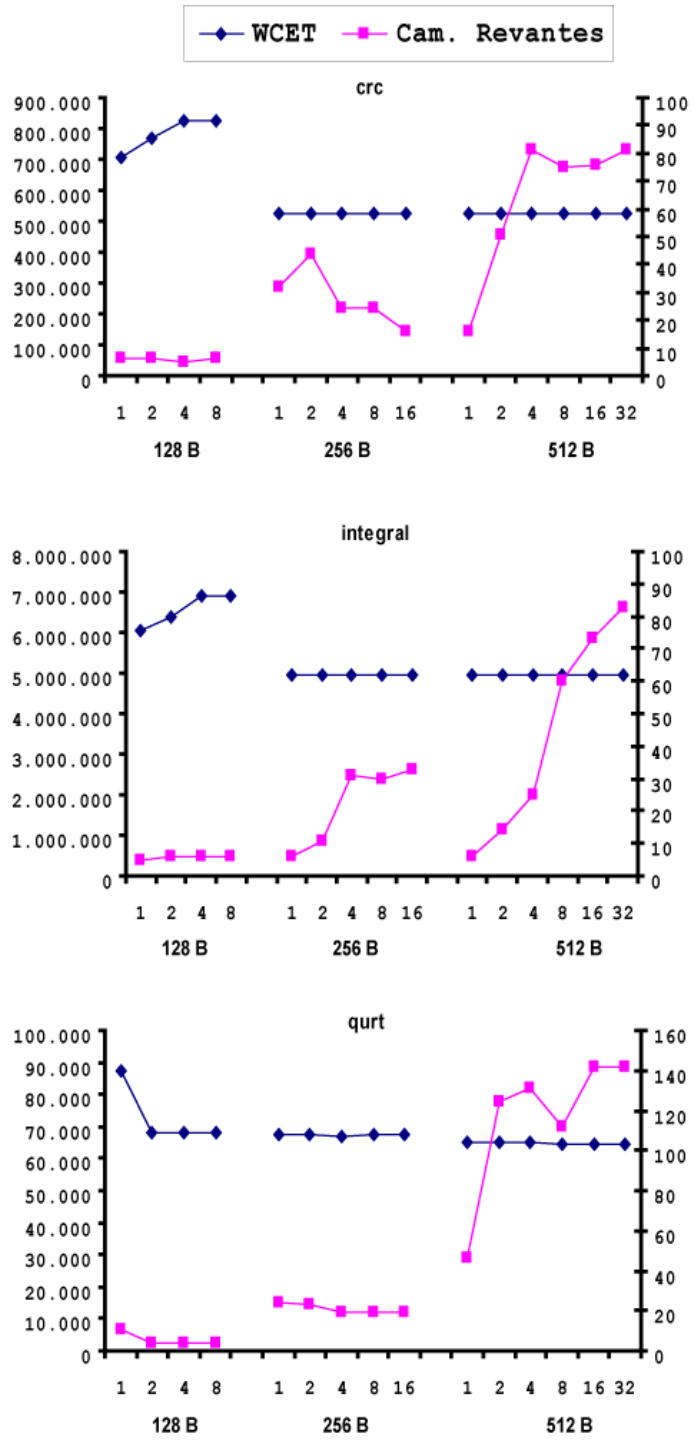


Figura 3.7: Eje Y de la izquierda: WCET. Eje Y de la derecha: *caminos relevantes*.

más precisas con esta organización de cache. De esta forma se pone en evidencia la importancia que tiene, para un sistema de tiempo real, realizar una predicción exacta del comportamiento de los accesos a las instrucciones, y también se refleja la sobrestimación que presenta la clasificación de dichos accesos realizada mediante SCS.

En la Tabla 3.4, para cada una de las tareas se resume el número de *caminos relevantes* obtenidos durante los experimentos. Además, se ha realizado una comparación de la contribución del *fetch* de instrucciones (IFC/ *Instruction Fetch Contribution*) al WCET. En las Figuras 3.8 y 3.9 se presenta el IFC obtenido mediante la técnica de poda que permite analizar todos los *caminos relevantes*, normalizado al IFC conseguido mediante la técnica SCS. También se muestra la ratio de aciertos del camino de ejecución que determina el WCET de cada tarea.

Tareas	Caminos Relevantes				
	Tamaño Línea	Capacidad de la Cache			Máximo Teórico
		128 B	256 B	512 B	
array_sum	8 B	2	2	2	3
	16 B	1	1	1	
	32 B	1	1	1	
bs	8 B	1	1	1	1
	16 B	1	1	1	
	32 B	1	1	1	
bubble	8 B	6	3	3	9
	16 B	5	3	3	
	32 B	3	3	3	
crc	8 B	14	114	126	216
	16 B	6	44	51	
	32 B	2	4	9	
integral	8 B	19	42	34	176
	16 B	10	14	14	
	32 B	4	7	5	
qurt	8 B	10	63	281	553
	16 B	4	23	124	
	32 B	4	18	73	

Tabla 3.4: *Caminos relevantes* en caches con asociatividad 2.

El IFC obtenido para cada tarea mediante la técnica de poda dinámica es exacto y siempre es inferior a la cota que proporciona el método SCS. Cuando el método SCS no puede clasificar de forma exacta un acceso a una instrucción que está en cache, se produce una sobrestimación del IFC y por consiguiente una sobrestimación en la cota del WCET calculada. Por ejemplo, si la instrucción clasificada erróneamente forma parte de un bucle, la sobrestimación generada

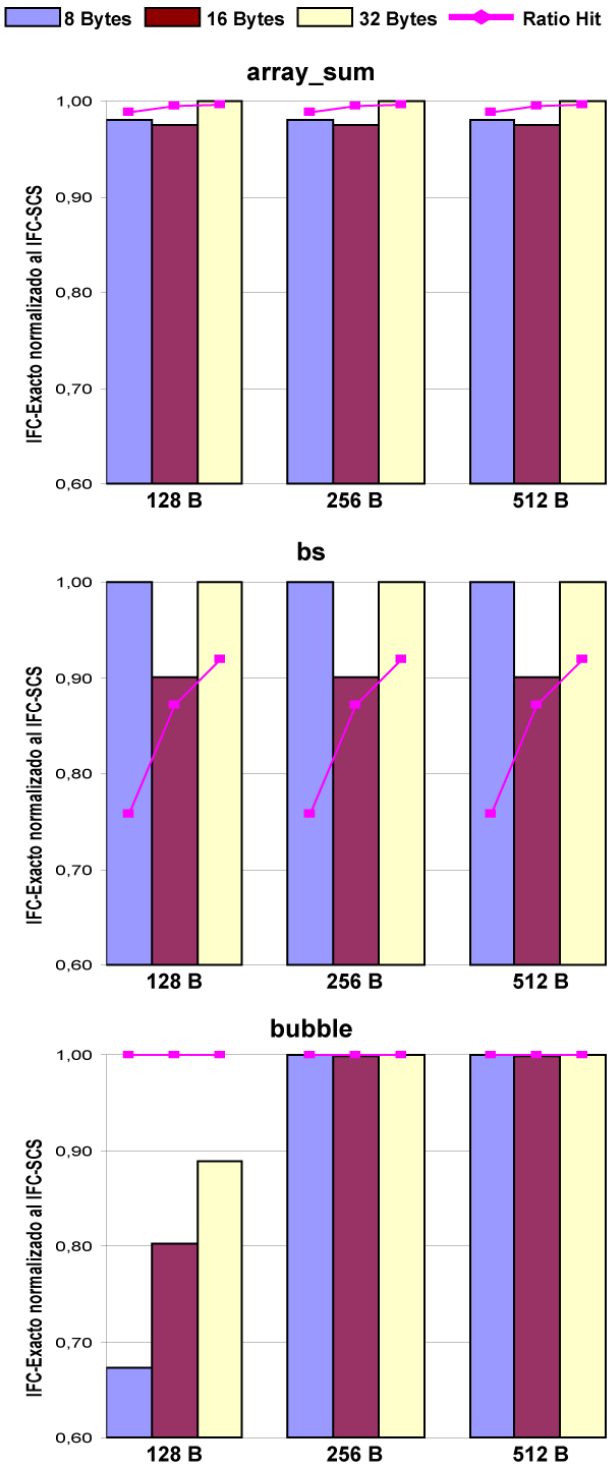


Figura 3.8: IFC exacto normalizado al IFC obtenido mediante SCS.

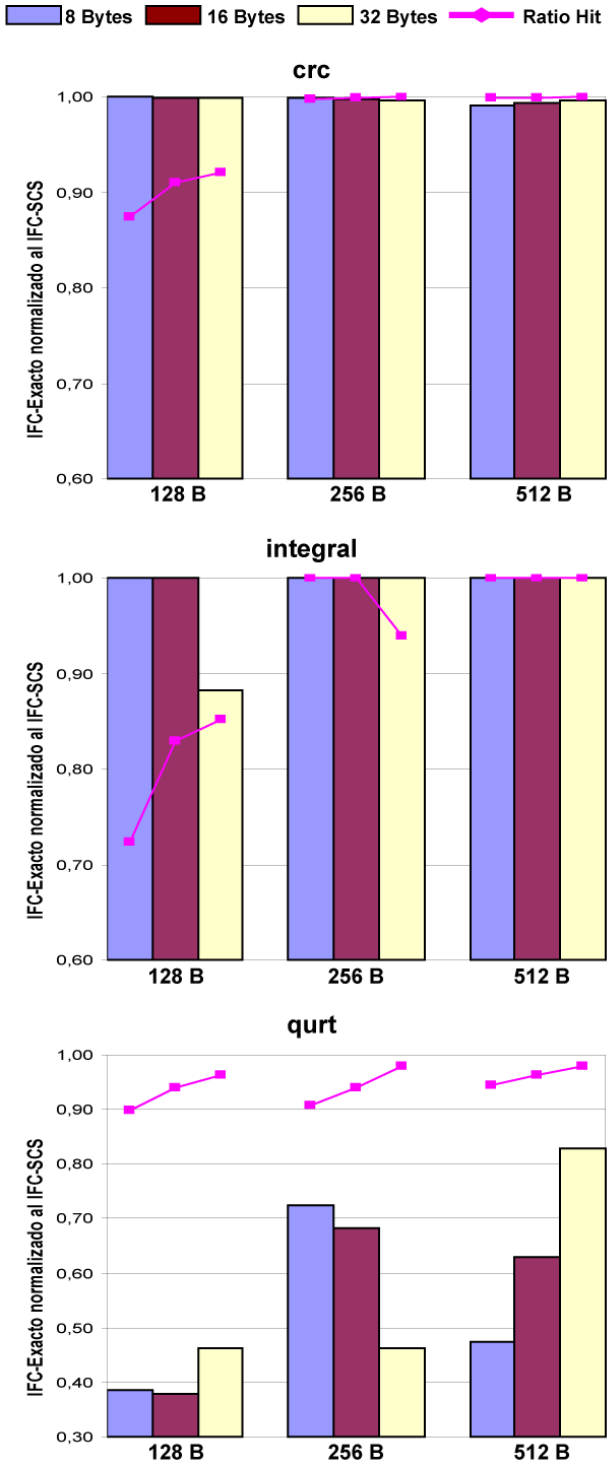


Figura 3.9: IFC exacto normalizado al IFC obtenido mediante SCS.

puede ser inadmisible. El método SCS clasifica los accesos a instrucciones de manera muy exacta cuando la tarea cabe completamente en la cache, mientras que cuando la tarea tiene muchos condicionales y no cabe en la cache, el análisis resulta excesivamente pesimista. Para evitar la explosión combinatoria de caminos, SCS construye un estado abstracto de cache asociado a todos los posibles caminos de ejecución. Pero, durante la construcción de este estado abstracto se pierde la historia de ejecución y en muchas ocasiones se deben considerar algunos accesos como fallo, cuando en realidad son aciertos.

En esta comparación entre el método de poda dinámico presentado en la sección anterior y SCS, también se ponen de manifiesto algunos resultados de interés para los diseñadores de sistemas de tiempo real. En primer lugar, como se muestra en la gráfica de la Figura 3.9, asociada a la tarea *qurt*, la exactitud del método SCS no mejora al aumentar la capacidad de la cache, cuando fijamos el tamaño de la línea de cache en 16 bytes. La exactitud del método SCS tampoco mejora al aumentar el tamaño de línea de cache, como se muestra en la gráfica de la Figura 3.8 que hace referencia a la tarea *bubble*, puesto que con una capacidad de cache de 128 bytes la exactitud mejora al incrementar el tamaño de línea, mientras que en la tarea *qurt*, con una capacidad de cache de 256 bytes, al aumentar el tamaño de línea la precisión del método SCS disminuye. La ratio de aciertos del camino que determina el WCET de cada tarea no es un buen indicador de la exactitud del método SCS. Por ejemplo, para la tarea *bubble* con una cache de capacidad de 128 bytes, la exactitud del método mejora al aumentar el tamaño de línea, pero la ratio de aciertos se mantiene constante. Finalmente, la clasificación de los accesos a memoria, obtenida mediante SCS para la tarea *qurt*, es muy pesimista, como se observa en la Figura 3.9. La tarea *qurt* calcula las raíces de una ecuación de segundo grado, pero, si se estudia la estructura del programa con más detalle, se observa que, comparada con el resto de tareas analizadas, es bastante más grande. Como la capacidad de la cache elegida para los experimentos es reducida, esta tarea no cabe en la cache para ninguno de los tamaños considerados, y además tiene una llamada a una función que contiene un estructura condicional dentro de un bucle con tres posibles caminos de ejecución. En particular, para esta tarea la técnica de poda permite reducir el IFC obtenido mediante SCS en aproximadamente un 62 %.

3.5. Conclusiones

En este capítulo se ha presentado una técnica que permite analizar, en el peor caso, una cache de instrucciones con algoritmo de reemplazo LRU. Esta nueva técnica permite podar, sin pérdida de información, los *caminos no relevantes* para el análisis y cálculo del WCET en presencia de una cache de instrucciones. En procesadores simples con caches, mediante esta técnica de poda se puede obtener el WCET exacto de una tarea, siempre y cuando el número de condicionales dentro de bucles de la tarea sea reducido. No obstante, suele coincidir que los *kernels* de tiempo real están dentro de esta categoría de programas.

Además, conviene indicar que esta técnica de poda dinámica de caminos permite cuantificar la sobrestimación en la predicción del peor caso de los accesos a la cache de instrucciones que presentan otros métodos propuestos en la literatura, como por ejemplo SCS (*Static Cache Simulation*) [134]. En particular, para la tarea *qurt*, la técnica de poda dinámica reduce aproximadamente en un 62 % el IFC obtenido mediante SCS.

También se ha demostrado que el número de caminos *relevantes* generado por estructuras condicionales dentro de bucles no depende del número de iteraciones del bucle, sino que depende del número de caminos del condicional. Por lo tanto y dado que el número de caminos alternativos de un bucle no suele ser grande, la mayoría de las tareas que forman un sistema de tiempo real se pueden analizar mediante la técnica de poda dinámica de caminos. Por lo tanto, para estas tareas es posible predecir la contribución exacta al WCET de los accesos a la cache de instrucciones.

Capítulo 4

El WCET con caches que pueden fijar su contenido

En un sistema de tiempo real multitarea, un método sencillo para resolver el problema de las interferencias de cache, tanto intrínsecas como extrínsecas, es fijar el contenido de la cache. Una forma de fijar el contenido de la cache, durante algunos periodos o durante toda la vida del sistema, es deshabilitar el algoritmo de reemplazo. Al fijar el contenido de la cache, cada acceso a memoria se puede predecir de forma totalmente exacta y segura. Además, se evita la explosión combinatoria de caminos en bucles que contienen condicionales, ya que el coste de cada acceso a memoria sólo depende de si la línea a la que se accede está bloqueada en la cache o no. Como al fijar el contenido de la cache es posible que disminuyan sus prestaciones haciendo que el WCET de las tareas aumente, es necesario que las líneas que se vayan a fijar en la cache sean las más adecuadas para que el rendimiento no disminuya. No obstante, en estudios anteriores ya se demostró que, con una selección adecuada de los contenidos a fijar, la *planificabilidad* del sistema puede mejorar considerablemente [8, 117, 118, 120, 121, 122, 124, 146, 147]. Sin embargo, en ninguno de estos estudios se garantiza que la selección de contenidos a fijar en la cache sea la óptima.

En este capítulo, para una jerarquía de memoria (ver Figura 4.1) formada por un almacén de línea de instrucciones (*Line Buffer*) y una cache que puede fijar su contenido (*Lockable iCache*), presentamos un algoritmo basado en programación lineal entera (ILP/ *Integer Linear Programming* [161, 36, 158]) para seleccionar los contenidos a fijar en la cache de instrucciones, de tal forma que cada una de las tareas del sistema pueda utilizar toda la cache y además, que la *planificabilidad* del sistema sea máxima. A este algoritmo lo hemos denominado *Lock-MS* (*Lock for Maximize Schedulability*).

El algoritmo *Lock-MS* selecciona las líneas de memoria que cada tarea del

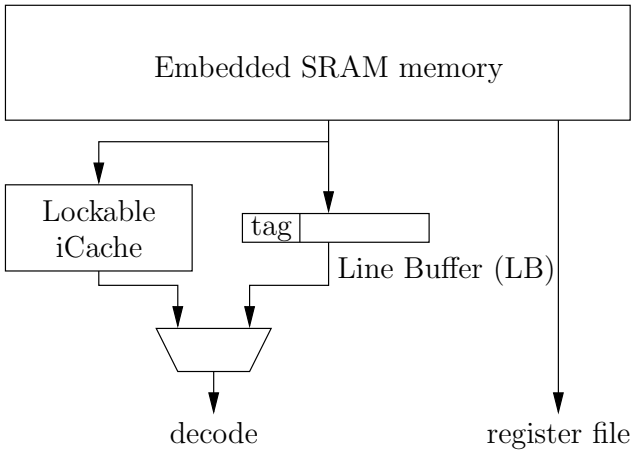


Figura 4.1: Jerarquía de memoria para instrucciones en un sistema de tiempo real.

sistema debe tener en la cache durante su ejecución. En cada cambio de contexto se actualiza el contenido de la cache con las líneas de memoria asociadas a la tarea que comienza o reanuda su ejecución. Una vez finalizada la carga de los contenidos de la cache seleccionados por el algoritmo *Lock-MS*, la cache de instrucciones quedará totalmente bloqueada. De esta forma, la tarea puede utilizar toda la cache, aunque se debe tener en cuenta el coste de cargar dichas líneas de memoria en cada cambio de contexto. Es decir, nuestra propuesta define un nuevo enfoque de las técnicas que en la literatura se han denominado *dynamic locking cache*.

4.1. Descripción de la jerarquía de memoria

En esta sección se describen en detalle los componentes y el funcionamiento de una arquitectura de memoria para instrucciones (ver Figura 4.1), que pretendemos utilizar en un sistema de tiempo real multitarea con expulsiones. Esta organización de la jerarquía de memoria ya está disponible en procesadores para sistemas empotrados. Además, también ha sido considerada en estudios anteriores, aunque sin aprovechar al máximo sus prestaciones [147, 118].

A continuación se describe el comportamiento de los dos componentes y el funcionamiento general de esta jerarquía de memoria.

- **Una cache que pueda fijar su contenido: *Lockable iCache***

El primer componente de esta jerarquía de memoria es una cache de instrucciones asociativa por conjuntos que pueda fijar o bloquear su contenido. Al fijar

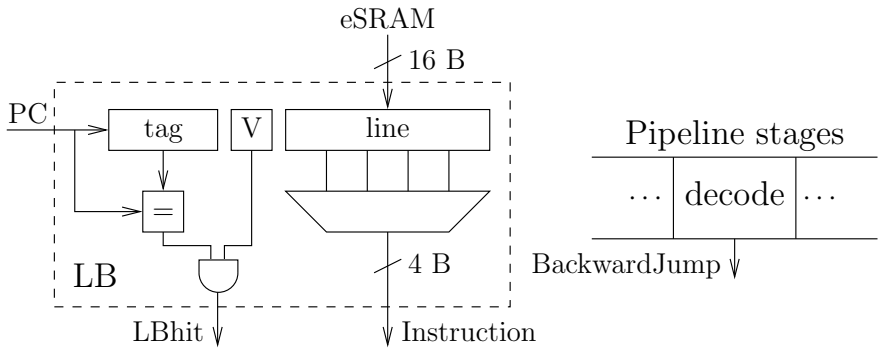


Figura 4.2: Organización y funcionamiento del LB.

el contenido de la cache, su comportamiento es totalmente predecible. Además, todas las interferencias de cache desaparecen, es decir, ya no es necesario considerar durante el análisis del peor caso, ni las interferencias intrínsecas ni las interferencias extrínsecas.

- **Un pequeño almacén de línea de instrucciones: *Line Buffer***

El segundo componente de la jerarquía de memoria es un pequeño almacén de instrucciones (LB/ *Line Buffer*) del tamaño de una línea de cache para capturar la localidad espacial. En la Figura 4.2 se muestra la organización y funcionamiento del LB propuesto.

Una *Lockable iCache* con las características descritas anteriormente sólo permite aprovechar la localidad temporal. Como se mostrará en los experimentos, este LB captura la localidad espacial muy bien, mejorando considerablemente el rendimiento de la jerarquía de memoria en el peor caso. Además, si las líneas de memoria bloqueadas en la cache son las más adecuadas para cada tarea, esta jerarquía de memoria podría superar en prestaciones incluso a una cache *convencional*.

- **Funcionamiento**

Durante la etapa de *fetch* de una instrucción se realiza una búsqueda en paralelo, en la *Lockable iCache* y en el LB. Obtendremos un acierto si encontramos

la instrucción en la cache o en el LB. En ambos casos, la instrucción se sirve en un ciclo de procesador.

Pero si se produce un fallo en ambas estructuras, la línea de memoria de la instrucción se solicita al siguiente nivel en la jerarquía de memoria, que es una *eSRAM* para sistemas empotrados, y el LB se actualizará con la línea de memoria solicitada. Por lo tanto, el comportamiento del LB es similar al de una cache con una única línea.

El funcionamiento del LB presenta un comportamiento particular, ya que su contenido se invalida automáticamente cuando se produce un acierto de cache y también cuando se produce un salto atrás, incluso si es a una instrucción dentro de la misma línea de memoria que contiene. Por lo tanto, su contenido no puede ser reutilizado y cuando una línea de memoria ya ha sido consumida, si es necesario, se debe solicitar otra vez a la memoria principal. En definitiva, este funcionamiento particular del LB evita que este componente pueda aprovechar la localidad temporal.

El procesador no dispone de otros recursos que puedan tener latencia variable, como por ejemplo un predictor de saltos o una cache de datos. El procesador no está segmentado y la ejecución de las instrucciones se realiza en orden. De esta forma, el sistema no presenta ningún tipo de *anomalía de distribución* (*timing anomalies*) [43, 115, 155, 193]. También suponemos que el número máximo de iteraciones de los bucles de cada tarea es conocido y que los caminos imposibles han sido identificados. No obstante, es importante señalar que si no se utilizan componentes con latencia variable, dependientes de la historia de ejecución, las interferencias intrínsecas de las tareas desaparecen, como ya se indicó en estudios anteriores [122].

4.2. *Lock-MS*: Selección de líneas a fijar

A continuación describimos el algoritmo *Lock-MS* (*Lock for Maximize Schedulability*) que selecciona las líneas de memoria a bloquear en la cache de cada tarea, para conseguir la máxima *planificabilidad* del sistema. En un sistema de tiempo real, este algoritmo basado en ILP hace posible obtener el máximo rendimiento de la jerarquía de memoria descrita en la Figura 4.1. Así pues, consideramos un sistema de tiempo real multitarea donde la prioridad de cada tarea es fija y se permiten las expulsiones. La planificación de las tareas del sistema se puede obtener de diferentes formas [163], en particular mediante RMA (*Rate Monotonic Analysis*).

La arquitectura de memoria que proponemos dispone de una *Lockable iCache* que puede bloquear su contenido durante algunos periodos de la ejecución del sistema, en particular durante la ejecución de cada tarea. En cada cambio

de contexto, el contenido de la cache se actualiza con las líneas de memoria de la tarea que se va a ejecutar. El comportamiento de esta cache es totalmente predecible. Cada tarea, cuando se ejecuta, puede aprovechar toda la cache de forma exclusiva, pero el coste de cargar las líneas de memoria de la tarea en cada cambio de contexto debe ser considerado. Sin embargo, las restricciones lineales para modelar una cache de instrucciones que pueda fijar su contenido son muy específicas y no son comparables con otras técnicas basadas en IPET (*Implicit Path-Enumeration Technique*) u otros métodos de modelado equivalentes [107].

La arquitectura propuesta también dispone de un LB cuyo funcionamiento se puede modelar mediante restricciones lineales, ya que su contenido sólo depende de la localización de la instrucción ejecutada previamente dentro de la línea de memoria. El funcionamiento del LB no genera ningún tipo de interferencia, puesto que cuando se produce un acierto de cache o un salto atrás, su contenido se invalida.

Con este modelo de arquitectura, en los saltos condicionales el camino de peor caso siempre seguirá la misma trayectoria, es decir, el camino de peor caso en un condicional se alcanza considerando siempre el salto como tomado o considerando siempre el salto como no tomado. En ningún caso el *camino más largo* vendrá determinado por una combinación de las dos posibilidades del condicional, ya que cada uno de los posibles caminos de ejecución es totalmente independiente de los demás.

El objetivo del algoritmo *Lock-MS* es determinar las líneas de memoria de cada tarea que se cargarán en la cache antes de su ejecución. El algoritmo tendrá en consideración el WCET de cada tarea y el coste de cargar las líneas seleccionadas en la cache en cada cambio de contexto. Para obtener la selección de las líneas a fijar, el algoritmo *Lock-MS* modela el problema a resolver mediante un conjunto de restricciones lineales basadas en ILP, cuya función objetivo será minimizar el WCET de cada tarea del sistema.

En definitiva, al minimizar el WCET de cada tarea y además tener en cuenta el coste de los cambios de contexto, el algoritmo *Lock-MS* obtiene la selección de líneas a fijar en la cache para que la *planificabilidad* del sistema sea máxima.

Modelado del problema ILP

Para modelar el problema ILP se define un sistema de tiempo real multitarea como un conjunto de tareas periódicas $Task_i$, tal que $1 \leq i \leq NTasks$, donde $NTasks$ indica el número de tareas del sistema. Así pues, una tarea $Task_i$ se puede modelar como un conjunto de caminos de principio a fin $Path_{i,j}$, con $1 \leq j \leq NPaths_i$, donde $NPaths_i$ indica el número de caminos de la tarea $Task_i$. Finalmente, cada camino $Path_{i,j}$ está formado por un conjunto

de $Nlines_{i,j}$ líneas de memoria.

En la Figura 4.3 se presenta el esquema de flujo de control de un programa sencillo. Se muestran una serie de líneas de memoria desde la L_1 hasta la L_{12}, organizadas en bloques básicos con las estructuras de control del programa y una función. Algunas líneas de memoria pueden formar parte de bloques básicos distintos, por ejemplo las líneas L_1 y L_4. Además, los bloques básicos pueden contener diferentes líneas de memoria, por ejemplo el camino de la izquierda dentro del bucle contiene las líneas L_3 y L_4. El flujo de control del *subgrafo* de la derecha corresponde a una función que se puede llamar desde dos puntos diferentes del programa que aparecen marcados con el símbolo $\mathbb{C}$.

En la Figura 4.4 se muestra una ampliación de la información del flujo de control presentado en la Figura 4.3, donde se especifica la información necesaria para el análisis y cálculo del WCET. En esta nueva representación aparecen algunos detalles clave en el modelado ILP:

i) Las líneas de memoria compartidas por bloques básicos diferentes se muestran divididas. A cada una de ellas se le asigna un identificador único, por ejemplo las líneas L_{4a} y L_{4b} forman parte de la misma línea de memoria L_4, pero pertenecen a dos bloques básicos distintos. Esto permite asociar distintos costes y diferentes números de acceso a cada una de las partes en las que se ha dividido una línea de memoria.

ii) La función se debe analizar teniendo en cuenta el punto del programa desde donde se invocó. En la Figura 4.4 se ha reflejado el análisis de las dos posibles instancias a la función. En este ejemplo se realiza una llamada a la función desde las líneas de memoria L_2 ($\mathbb{C}^1$) y L_{4b} ($\mathbb{C}^2$).

iii) Todos los bloques básicos están etiquetados con los posibles caminos a los que pertenecen. Por lo tanto, cualquier camino $Path_j$ está perfectamente identificado de principio a fin. Por ejemplo, el camino $Path_2$ recorre los bloques básicos que forman las líneas de memoria L_{1a}, L_{1b}, L_2, L_{9a}, L_{9b}, L_{11a}, L_{11b}, L_{12a}, L_{12b}, L_{8a} y L_{8b}.

iv) El coste de ejecutar cualquier línea de memoria en un camino concreto es constante, incluso aunque el camino seguido hasta esa línea sea distinto. Es decir, el coste de ejecutar una línea de memoria no depende, en ningún caso, de la línea de memoria ejecutada previamente. Por ejemplo, en el camino $Path_2$ se ejecuta la línea L_{1b} una vez, después de ejecutar la línea L_{1a}, y $bound_2^1$ veces, después de ejecutar la línea L_{8a}.

En general, cada tarea $Task_i$ se puede modelar como un conjunto de caminos $Path_{i,j}$ donde el coste de ejecución del *camino más largo* será el WCET de la tarea. Así pues, el WCET de una tarea, representado por $wcet_i$, debe ser mayor o igual que el coste de ejecución de todos y cada uno de sus caminos.

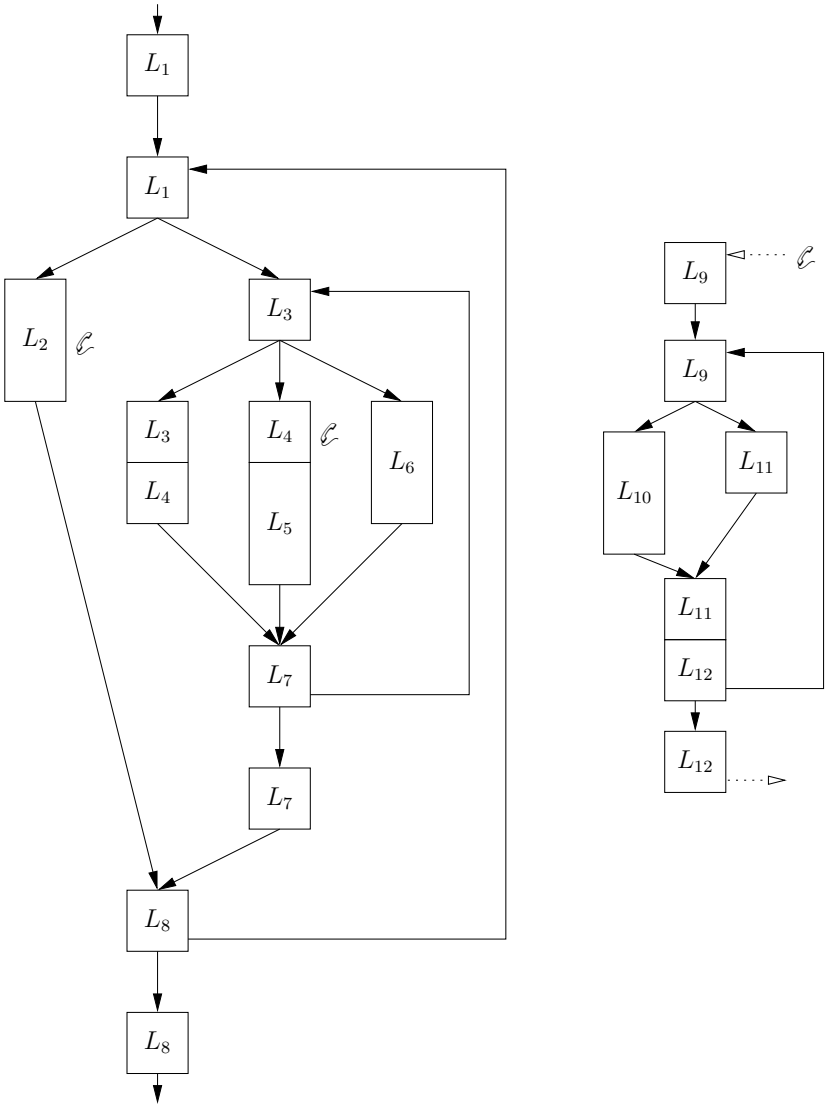


Figura 4.3: Modelado del flujo de control donde las líneas de memoria se han dividido en bloques básicos.

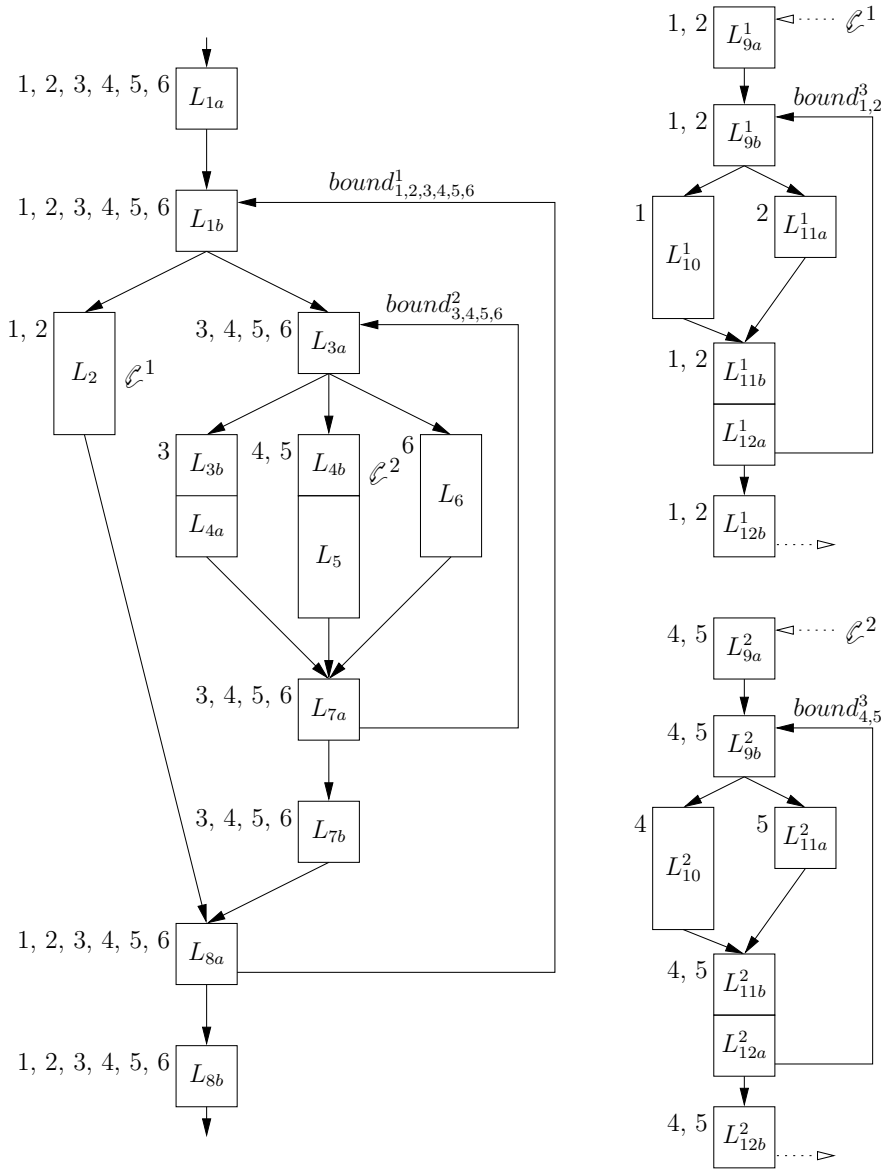


Figura 4.4: Modelado del flujo de control donde: a) las líneas de memoria divididas se han identificado de forma única, b) se han introducido las instancias a la función, c) se han anotado los caminos a los que pertenece cada bloque básico.

Si el problema ILP trata de minimizar el coste de ejecución de todos los caminos $pathCost_{i,j}$ de la tarea $Task_i$, se obtendrá la cota inferior del WCET.

Por lo tanto, en el problema ILP, el WCET de cada tarea del sistema se modela mediante el siguiente conjunto de restricciones:

$$wcet_i \quad \geq \quad pathCost_{i,j} \quad \forall\, 1 \leq j \leq NPaths_i \tag{4.1}$$

Por ejemplo, las restricciones del WCET del programa de la Figura 4.4, puesto que dicho programa puede seguir 6 caminos de ejecución, son las siguientes:

$$wcet \geq pathCost_j \quad \forall\, 1 \leq j \leq 6$$

Cada uno de los caminos $Path_{i,j}$ de la tarea $Task_i$ está formado por un conjunto de líneas de memoria, a las que accederá durante su ejecución. Por lo tanto, el coste de ejecución $pathCost_{i,j}$ de un camino $Path_{i,j}$ se puede calcular como la suma de los costes de ejecución de cada una de sus líneas de memoria $lineCost_{i,j.k}$.

Así pues, el coste de ejecución de cada camino de una tarea $Task_i$ se representa mediante la siguiente ecuación:

$$pathCost_{i,j} \quad = \quad \sum_{k=1}^{Nlines_{i,j}} lineCost_{i,j,k} \tag{4.2}$$

Por ejemplo, el coste del camino $Path_2$ en el programa de la Figura 4.4, vendrá determinado por la siguiente expresión:

$$pathCost_2 = \sum_{k=1}^{Nlines_2} lineCost_{2,k}$$

$$\begin{aligned} pathCost_2 = {} & lineCost_{2,1a} + lineCost_{2,1b} + lineCost_{2,2} + lineCost_{2,9a^1} \\ & + lineCost_{2,9b^1} + lineCost_{2,11a^1} + lineCost_{2,11b^1} + lineCost_{2,12a^1} \\ & + lineCost_{2,12b^1} + lineCost_{2,8a} + lineCost_{2,8b} \end{aligned}$$

Coste de ejecución de las líneas de memoria

Como ya se indicó en el apartado iv) anterior, cada línea de memoria $L_{i,j,k}$ con $1 \leq k \leq Nlines_{i,j}$ de un camino $Path_{i,j}$ tiene asociado un coste de ejecución constante $lineCost_{i,j,k}$. Pero este coste depende del tiempo de ejecución de las instrucciones que contiene la línea, y del número de accesos a dicha línea de memoria que son considerados aciertos y fallos de cache.

Así pues, la representación del coste de una línea de memoria se ha dividido en dos sumandos, el primero asociado al coste de un acierto de cache y el segundo asociado a un fallo de cache, que se representan con las constantes $IChitCost$ y $ICmissCost$ respectivamente. También depende del número de accesos a la línea que son aciertos, y se indica con la variable $nIChit$, y del número de accesos que son fallos, que representamos con la variable $nICmiss$.

Por lo tanto, el coste de ejecución de una línea de memoria $L_{i,j,k}$ se puede obtener mediante la siguiente expresión:

$$\begin{aligned} lineCost_{i,j,k} &= IChitCost_{i,j,k} \cdot nIChit_{i,j,k} + \\ & \quad ICmissCost_{i,j,k} \cdot nICmiss_{i,j,k} \end{aligned} \tag{4.3}$$

Además, para determinar dentro de un camino concreto $Path_{i,j}$ el valor del número de aciertos $nIChit$ y del número de fallos $nICmiss$ de cache, se debe tener en cuenta el número máximo de accesos $nfetch_{i,j,k}$ a la línea de memoria. El valor de esta constante se puede obtener de diferentes formas, como por ejemplo con el método utilizado en [147].

Por lo tanto, para cada línea de memoria, en función del número de accesos a dicha línea, se deben describir las restricciones que determinan el número de aciertos y de fallos. La variable binaria $cached_l \in [0,1]$ indica si la línea de memoria $L_{i,j,k} \in Path_{i,j}$ que comienza en la dirección física $l \times memlineSize$ está bloqueada en la cache. El algoritmo *Lock-MS* asigna el valor 1 a dicha variable ($cached_l = 1$) cuando la línea de memoria debe ser cargada y bloqueada en la cache, y el valor 0 cuando la línea no se debe cargar en la cache ($cached_l = 0$).

La selección final de las líneas de memoria a fijar en la cache de instrucciones puede incluir líneas de diferentes caminos. Esta situación se producirá cuando existan varios caminos con un coste de ejecución parecido y puedan alcanzar el WCET de la tarea, y se intente reducir el WCET de cada uno de estos caminos.

El número de aciertos y el número de fallos de cada línea de memoria $L_{i,j,k}$ se determina mediante la siguiente expresión:

$$\begin{aligned} nIChit_{i,j,k} &= nfetch_{i,j,k} \cdot cached_l \\ nICmiss_{i,j,k} &= nfetch_{i,j,k} - nIChit_{i,j,k} \end{aligned} \tag{4.4}$$

Por ejemplo, la siguiente expresión define el valor del número de accesos a la línea $L_{2,9b}$ en el camino $Path_2$ de la Figura 4.4.

$$nfetch_{2,9b} = (1 + bound_2^1) \cdot (1 + bound_2^3)$$

Se indican a continuación las restricciones asociadas a la línea de memoria L_{11} en el camino $Path_2$ de la Figura 4.4. Esta misma descripción se debe

repetir para cada una de las líneas de memoria del camino.

$$\begin{aligned} nIChit_{2,11a^1} &= nfetch_{2,11a^1} \cdot cached_{11} \\ nICmiss_{2,11a^1} &= nfetch_{2,11a^1} - nIChit_{2,11a^1} \\ nIChit_{2,11b^1} &= nfetch_{2,11b^1} \cdot cached_{11} \\ nICmiss_{2,11b^1} &= nfetch_{2,11b^1} - nIChit_{2,11b^1} \end{aligned}$$

Para finalizar la descripción de las ecuaciones que determinan el coste de ejecución de una línea de memoria $L_{i,j,k}$, sólo queda indicar cómo se calcula el valor de las constantes $IChitCost_{i,j,k}$ y $ICmissCost_{i,j,k}$.

Si la línea de memoria $L_{i,j,k}$ está en la cache, para obtener el coste de ejecución $IChitCost_{i,j,k}$ se debe sumar el coste de ejecución de las instrucciones de la línea $texec_{i,j,k}$ y el coste de cada uno de los accesos a la misma. En este caso será el coste de un acierto $thit_{CM}$ multiplicado por el número de instrucciones que contiene la línea $nIns_{i,j,k}$.

El coste de ejecución de una línea de memoria cuando se produce un acierto de cache, cuyo coste se representa por $thit_{CM}$, se calcula mediante la siguiente ecuación:

$$IChitCost_{i,j,k} = texec_{i,j,k} + thit_{CM} \cdot nIns_{i,j,k} \tag{4.5}$$

Si la línea de memoria $L_{i,j,k}$ no está en la cache, el coste de acceso dependerá de la penalización por fallo de cache. Cuando el sistema no dispone de LB, el coste de un fallo de cache $tmiss_{CM}$ será el mismo para cada una de las instrucciones de la línea. Por lo tanto, el coste de ejecución de la línea de memoria se obtiene sumando el coste de ejecución de las instrucciones que contiene $texec_{i,j,k}$ con el coste de un fallo de cache por cada una de las instrucciones que contiene la línea $nIns_{i,j,k}$.

El coste de ejecución de una línea de memoria cuando se produce un fallo de cache se representa mediante la siguiente ecuación:

$$ICmissCost_{i,j,k} = texec_{i,j,k} + tmiss_{CM} \cdot nIns_{i,j,k} \tag{4.6}$$

No obstante, como la arquitectura de memoria propuesta dispone de un pequeño LB, esta última ecuación se debe actualizar convenientemente para tener en cuenta este componente. Por lo tanto, para determinar el coste de ejecución de una línea de memoria $L_{i,j,k}$ que no está en la cache, se considera el coste de un fallo de LB $tmiss_{LB}$ para el primer acceso a la línea de memoria y se debe tener en cuenta el coste de un acierto de LB $thit_{LB}$ para el resto de los accesos $(nIns_{i,j,k} - 1)$.

En presencia de un LB, la siguiente ecuación describe el coste de ejecución de una línea de memoria cuando se produce un fallo de cache:

$$ICmissCost_{i,j,k} = texec_{i,j,k} + tmiss_{LB} + thit_{LB} \cdot (nIns_{i,j,k} - 1) \quad (4.7)$$

Finalmente, para tener en cuenta la configuración de la cache de instrucciones, es necesario modelar el número de líneas que puede contener, ya que, tanto el número de conjuntos $NSets$, como la asociatividad $NWays$, limitan el contenido de la cache. Si mediante la constante $Mlines$ se denota el número de líneas de memoria física del sistema, y las líneas de memoria que puede contener un conjunto C_s se representan mediante $L_{s+h \cdot NSets}$, con $0 \leq h \leq [Mlines - 1/NSets]$ el número de vías $NWays$ siempre debe ser mayor o igual que el número de líneas que puede contener cada conjunto C_s, con $0 \leq s \leq NSets$.

Por lo tanto, para cada conjunto C_s la configuración de la cache se modela mediante la siguiente restricción:

$$NWays \geq \sum_{h=0}^{\left[\frac{Mlines-1}{NSets}\right]} cached_{L_{s+h \cdot NSets}} \quad \forall\, 0 \leq s < NSets \quad (4.8)$$

Función objetivo y costes de los cambios de contexto

El conjunto de restricciones anteriores es suficiente para modelar el problema ILP y resolver la Ecuación 4.1. Pero en un sistema multitarea con expulsiones también se debe considerar el coste asociado a los cambios de contexto de cada tarea $Task_i$.

El coste de un cambio de contexto $tSwitch$ se puede definir como el tiempo que cuesta guardar el estado de la tarea expulsada, y el tiempo de restaurar el estado de la nueva tarea que se va a ejecutar. Además, en la jerarquía de memoria propuesta es necesario considerar el coste de cargar el contenido de la cache $ICpreloadCost_i$ de cada tarea con las líneas de memoria seleccionadas y el coste $LBpreloadCost$ de actualizar el LB, ya que su contenido será invalidado. En particular, el coste de un cambio de contexto $switchCost_i$ de una tarea $Task_i$ depende del número de líneas $numcached$ que se cargarán en la cache antes de comenzar o reanudar su ejecución.

Por lo tanto, teniendo en cuenta la constante $Mlines$ que, como se ha comentado anteriormente, representa el número de líneas de memoria física del sistema, se puede calcular el coste del cambio de contexto $switchCost_i$ de la tarea $Task_i$ mediante las siguientes ecuaciones:

$$switchCost_i = tSwitch + ICpreloadCost_i + LBpreloadCost$$

$$\begin{aligned} LBpreloadCost &= tmiss_{LB} - thit_{LB} \\ ICpreloadCost_i &= (tmiss_{CM} - thit_{CM}) \cdot numcached_i \\ numcached_i &= \sum_{l=0}^{Mlines-1} cached_l \end{aligned} \tag{4.9}$$

El número exacto de cambios de contexto $ncSwitch_i$ de cualquier tarea $Task_i$, a priori, no es conocido. Sin embargo, se puede sobrestimar de varias formas [100, 101, 139, 170, 171]. Aunque no es objetivo de esta Tesis determinar de un modo exacto el número máximo de expulsiones de una tarea, se puede cuantificar, tanto de forma analítica, como experimental. Por ejemplo, de forma analítica se puede garantizar que el número de expulsiones nunca es mayor que la suma del número máximo de activaciones de las tareas de mayor prioridad durante el periodo T_i de ejecución de la tarea $Task_i$.

$$ncSwitch_i \leq \sum_{j=1}^{i-1} \left\lceil \frac{T_i}{T_j} \right\rceil$$

No obstante, esta cota se puede mejorar cuando sea muy pesimista. Por ejemplo, una vez que el problema ILP ha sido resuelto, se puede obtener el tiempo de respuesta R_i de cada una de las tareas del sistema simulando su ejecución, y por lo tanto determinar si el sistema es *planificable*. Además, cuando el número máximo de expulsiones de cada tarea $ncSwitch_i$ es muy pesimista, se podría determinar de una forma más exacta, considerando el tiempo de respuesta de las tareas en vez de su periodo de activación, como se indica en la siguiente ecuación:

$$ncSwitch_i \leq \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil \tag{4.10}$$

Con este nuevo valor de $ncSwitch_i$, y de forma recursiva, se podrían obtener valores más precisos del WCET de cada tarea $Task_i$ del sistema.

En nuestro caso, para obtener el número de cambios de contexto, se ha simulado la planificación de un sistema multitarea mediante *Rate Monotonic*, donde todas las tareas inician su ejecución al mismo tiempo. A partir del WCET y del número de líneas de memoria de cada tarea $Task_i$ a bloquear en la cache, el simulador planifica la ejecución de las tareas y obtiene, tanto el número de cambios de contexto $ncSwitch_i$, como el tiempo de respuesta R_i de cada una de las tareas del sistema.

Así pues, para tener en cuenta el coste completo de ejecutar una tarea $Task_i$, se debe modificar la función objetivo del problema ILP de la Ecuación 4.1 añadiendo el coste completo de todos los cambios de contexto $ncSwitch_i$ sufridos por

la tarea.

Por lo tanto, para cada tarea $Task_i$ la nueva función objetivo viene determinada por la siguiente ecuación:

$$Wcost_i = wcet_i + ncSwitch_i \cdot switchCost_i \quad (4.11)$$

Cuando se minimiza la función objetivo $Wcost_i$, el algoritmo *Lock-MS* consigue la selección de líneas de cache que se cargarán y bloquearán en la cache ($cached_l = 1$) antes de empezar o de continuar la ejecución de la tarea $Task_i$. Además, también se obtendrá el coste de los cambios de contexto $switchCost_i$ y del WCET ($wcet_i$) de la tarea $Task_i$, ya que el conjunto de líneas de memoria seleccionadas por el algoritmo *Lock-MS* determina el coste de ejecución del camino de peor caso de cada tarea.

Finalmente, conviene indicar que al tener en cuenta el coste asociado a los cambios de contexto, es posible que el algoritmo *Lock-MS* no considere adecuado cargar en la cache algunas líneas de memoria aunque se acceda varias veces a ellas. Por ejemplo, si a una línea de memoria se accede 3 veces, pero la tarea puede ser expulsada hasta 10 veces, el coste de cargar esta línea en la cache es mayor que el coste de acceder a dicha línea de memoria durante la ejecución. Por lo tanto, esta línea de memoria nunca será seleccionada por el algoritmo. Así pues, al añadir los cambios de contexto en el modelo ILP, el Lock-MS decide no seleccionar este tipo de líneas, ya que no reducen el tiempo de ejecución de peor caso del sistema y podría incluso no utilizar toda la capacidad de la cache.

En definitiva, la selección de líneas de memoria a cargar y bloquear en la cache minimiza el WCET de cada tarea, teniendo en cuenta el coste de ejecución del sistema multitarea completo, por lo tanto la *planificabilidad* del sistema es máxima.

Restricciones asociadas a la información de control

El modelado ILP también hace posible añadir información adicional obtenida del análisis de flujo de control realizado en alto nivel. Por ejemplo, permite indicar si un determinado camino dentro de un bucle se ejecutará al menos una vez, o si al ejecutar un camino concreto el número de iteraciones de un bucle se reduce. Así pues, para un camino concreto, se puede modificar el número de accesos a una determinada línea de memoria. Por lo tanto, en el modelo propuesto también se puede añadir esta información mediante nuevas restricciones o modificando las ya existentes de forma muy parecida al modelo IPET [107].

En la Figura 4.5 se observa un caso típico de cómo añadir la información funcional obtenida previamente [107]. En el ejemplo se supone que cada rectángulo

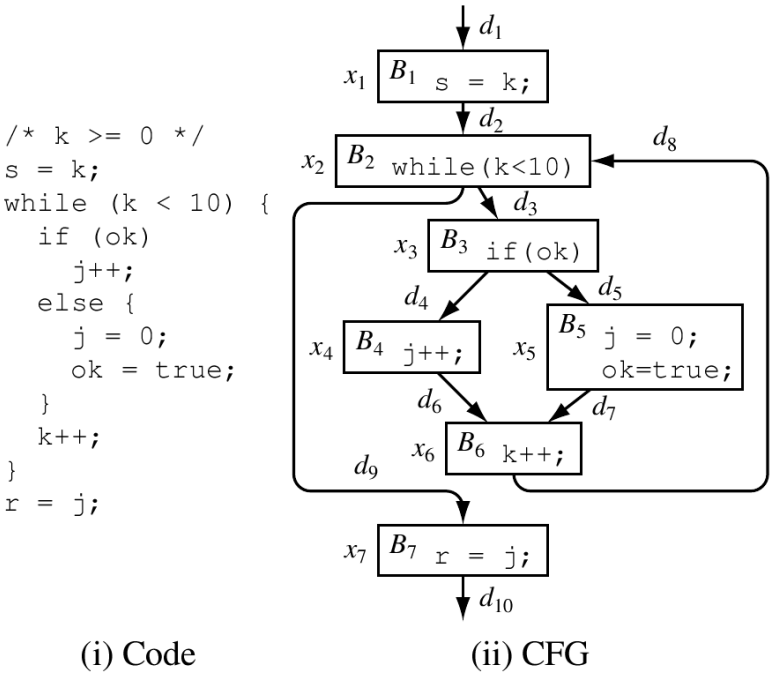


Figura 4.5: Ejemplo de programa con información funcional [107].

representa un bloque básico asociado a una única línea de memoria. También se muestran dos caminos dependientes de una estructura condicional *if-then-else* donde el número máximo de iteraciones del bucle es 10.

El interés del análisis funcional de este ejemplo reside en la línea de memoria B_5, ya que sólo se puede ejecutar una vez. Si consideramos los dos posibles caminos de ejecución dentro del bucle, el camino P_1 siempre ejecutará el caso *then* y el camino P_2 ejecutará sólo una vez el caso *else*. El número máximo de accesos a las líneas de memoria B_4 y B_5 para cada uno de los caminos P_1 y P_2 se puede modelar añadiendo al problema ILP las siguientes restricciones:

$$\begin{aligned} nfetch_{i,P1,B_4} &= 10 \\ nfetch_{i,P1,B_5} &= 0 \\ nfetch_{i,P2,B_4} &= 9 \\ nfetch_{i,P2,B_5} &= 1 \end{aligned}$$

Si se sabe que cuando se ejecuta la línea B_5 el número de iteraciones del bucle es de exactamente 5 iteraciones, se tiene otro ejemplo distinto. En este caso, el número máximo de accesos a las líneas de memoria B_4 y B_5, para cada uno de

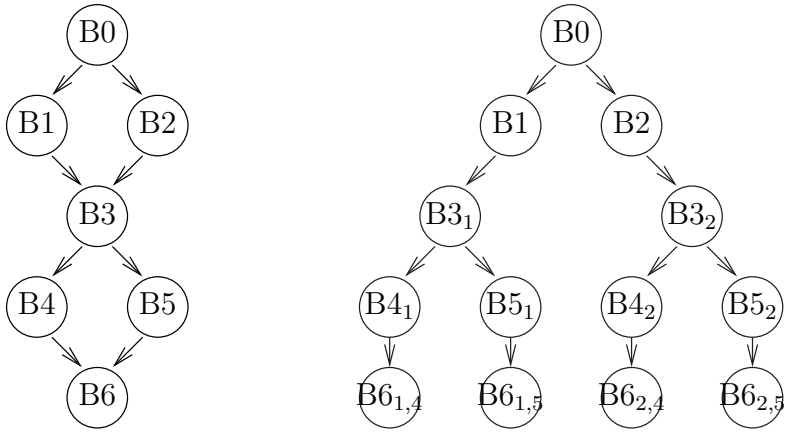


Figura 4.6: Grafo de flujo de control y explosión de los caminos de ejecución.

los caminos P_1 y P_2, se puede modelar mediante las siguientes restricciones:

$$\begin{aligned} nfetch_{i,P1,B_4} &= 10 \\ nfetch_{i,P1,B_5} &= 0 \\ nfetch_{i,P2,B_4} &= 4 \\ nfetch_{i,P2,B_5} &= 1 \end{aligned}$$

4.3. Un *modelo compacto* para reducir las restricciones

La base del algoritmo *Lock-MS* es la descripción de todos y cada uno de los posibles caminos de ejecución. Pero, si el número de caminos de las tareas es grande, el conjunto de restricciones del problema ILP puede crecer considerablemente haciendo incluso inviable su descripción. En esta sección definimos una transformación para simplificar la descripción de los posibles caminos de ejecución en el modelo ILP.

Con el fin de ilustrar esta transformación se considera el ejemplo de la Figura 4.6 donde se muestra un sencillo grafo de flujo de control y la explosión de los posibles caminos de ejecución. El grafo de flujo de control representa la ejecución de 4 caminos y sus bloques básicos asociados.

En este ejemplo, mediante las siguientes descripciones de los caminos, se calcula el WCET como el máximo tiempo de ejecución asociado a cada uno de

los 4 caminos:

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

$$WCET = \text{máx}(P1, P2, P3, P4) \tag{4.12}$$

Para calcular el WCET de la tarea se deben seleccionar, cargar y bloquear las líneas de memoria L_k, de tal forma que el WCET de la tarea sea mínimo. Por lo tanto, la Ecuación 4.12 se transforma, en función del conjunto de líneas de memoria Set_L asociadas a dicha tarea, como se indica a continuación:

$$WCET = \min_{L_k \in Set_L} (\text{ máx}(P1_{L_k}, P2_{L_k}, P3_{L_k}, P4_{L_k}))$$

No obstante, el modelo ILP presentado puede resolver esta ecuación eligiendo de forma exacta y concreta las líneas $L_k \in Set_L$. Por claridad en la notación y sin pérdida de generalidad, se eliminan los subíndices, y la ecuación anterior se escribe de este modo:

$$WCET = \text{mín}(\text{ máx}(P1, P2, P3, P4)) \tag{4.13}$$

Además, como ya se ha comentado anteriormente, al cargar y fijar en la cache los contenidos seleccionados por el algoritmo Lock-MS, el coste de ejecución de cualquier bloque básico es independiente del bloque ejecutado anteriormente. De forma análoga, esta idea puede extenderse al LB, ya que su comportamiento sólo depende de la instrucción anterior, que además pertenece al camino analizado. Por lo tanto, en la notación utilizada para representar los bloques básicos propios de cada camino también se suprimen los subíndices, ya que el coste de ejecución de todos los bloques se calcula de forma aislada. Con esta nueva notación se actualiza la Ecuación 4.13 anterior describiendo el coste de ejecución de cada bloque básico.

$$\begin{aligned} WCET = \text{mín}(& \text{ máx}(B0 + B1 + B3 + B4 + B6, \\ & B0 + B1 + B3 + B5 + B6, B0 + B2 + B3 + B4 + B6, \\ & B0 + B2 + B3 + B5 + B6)) \end{aligned}$$

Finalmente, si se agrupan los bloques comunes a todos los caminos en varios sumandos constantes, se tiene una nueva ecuación que determina el WCET de forma compacta:

$$\begin{aligned} WCET = \text{mín}(& B0 + B3 + B6 \\ & + \text{máx}(B1 + B4, B1 + B5, B2 + B4, B2 + B5)) \end{aligned} \tag{4.14}$$

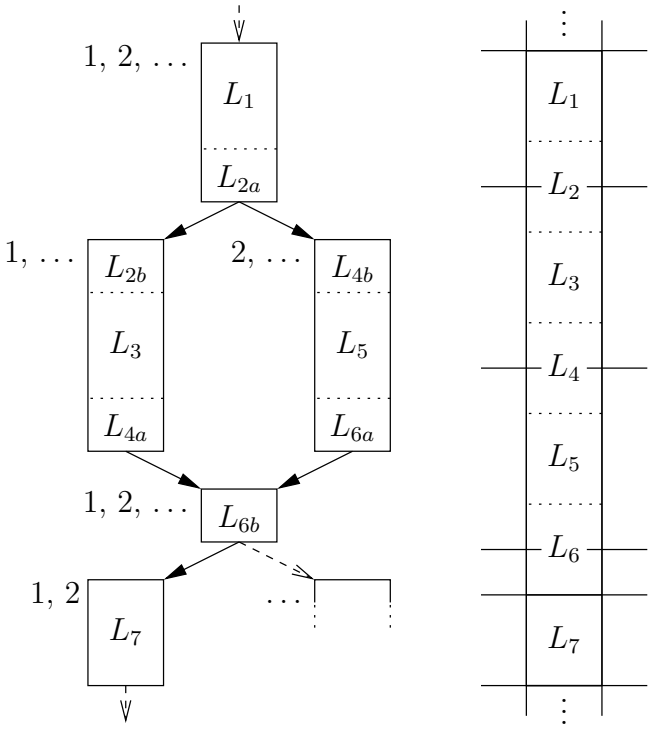


Figura 4.7: Descripción gráfica del *modelo explícito* y del *compacto*.

Para detallar la simplificación anterior, se presenta el ejemplo de la Figura 4.7 donde se muestran dos caminos $Path_1$ y $Path_2$ formados por una serie de líneas de memoria L_x y sus bloques básicos asociados. Si una línea de memoria es seleccionada y bloqueada en la cache, el coste de acceso a las instrucciones de esta línea siempre es el coste de un acierto de cache. Si la línea de memoria no está en cache, el coste de un acceso a la línea depende del comportamiento del LB. El primer acceso a la línea de memoria siempre tiene un coste de fallo de LB $tmiss_{LB}$, mientras que el resto de los accesos tienen un coste de acierto de LB $thit_{LB}$. Además, como ya se ha comentado anteriormente, y se observa en la figura, se debe tener en cuenta la posible división entre las líneas de memoria y los bloques básicos.

En el Tabla 4.1 se resume el coste del primer acceso al LB cuando ninguna de las líneas de los caminos $Path_1$ y $Path_2$ de la Figura 4.7 han sido bloqueadas en la cache. Toda esta información se puede obtener estáticamente analizando el código. Mediante el *modelo explícito* de caminos se observan las líneas que pertenecen a cada camino y el coste del primer acceso a cada una de ellas. Si la línea pertenece a los dos caminos, tendría dos costes asociados y si sólo pertenece a uno de los caminos sólo tendrá un coste asociado.

Líneas	Modelo explícito		Modelo compacto
	$Path_1$	$Path_2$	
L_1	$tmiss_{LB}$	$tmiss_{LB}$	$tmiss_{LB}$
L_{2a}	$tmiss_{LB}$	$tmiss_{LB}$	$tmiss_{LB}$
L_{2b}	0	-	0
L_3	$tmiss_{LB}$	-	$tmiss_{LB}$
L_{4a}	$tmiss_{LB}$	-	$tmiss_{LB}$
L_{4b}	-	$tmiss_{LB}$	$tmiss_{LB}$
L_5	-	$tmiss_{LB}$	$tmiss_{LB}$
L_{6a}	-	$tmiss_{LB}$	0
L_{6b}	$tmiss_{LB}$	0	$tmiss_{LB}$
L_7	$tmiss_{LB}$	$tmiss_{LB}$	$tmiss_{LB}$

Tabla 4.1: Coste del primer acceso a las líneas de memoria en la Figura 4.7 cuando no están en cache.

En la columna etiquetada como *Modelo compacto* de la Tabla 4.1 se muestra el coste del primer acceso a cada una de las líneas de memoria del ejemplo, cuando se aplica el método compacto propuesto. En la mayor parte de los casos se puede trasladar directamente el coste del primer acceso a una línea de memoria del *modelo explícito*, al *modelo compacto*, definiendo una serie de casos básicos que se describen a continuación:

- *Caso 1:* Si una línea de memoria pertenece a dos o más caminos, el coste del primer acceso a dicha línea se traslada directamente al *modelo compacto*.
- *Caso 2:* Si una línea de memoria pertenece a un solo camino, el coste del primer acceso a dicha línea también se traslada directamente al *modelo compacto*.
- *Caso 3:* Si una línea de memoria tiene una parte común que pertenece a dos o más caminos, y una parte que sólo pertenece a uno de los caminos, entonces el coste del primer acceso a dicha línea se asocia y se traslada directamente a la parte común en el *modelo compacto*, y se asigna un coste 0 a cada una de las partes particulares de cada camino.

Como se puede observar en la Tabla 4.1, las líneas L_1 y L_7 son comunes a los caminos $Path_1$ y $Path_2$, y representan un ejemplo del *Caso 1*. Por lo tanto, en el *modelo compacto* se asigna directamente el coste del primer acceso a la línea de memoria. Las líneas L_3 y L_{4a} pertenecen sólo al camino $Path_1$, mientras que las líneas L_{4b} y L_5 pertenecen sólo al camino $Path_2$. Ambos casos, representan un ejemplo del *Caso 2*. Por lo tanto, en el *modelo compacto* se asigna directamente el coste del primer acceso a cada una de las líneas de memoria. Como ejemplo del *Caso 3*, se observa, por un lado, la línea L_2 que tiene una parte común L_{2a} a los dos caminos y una parte particular L_{2b} que

pertenece únicamente al camino $Path_1$, y por otro lado, la línea L_6 que tiene una parte particular L_{6a} que pertenece únicamente al camino $Path_2$, y una parte común L_{6b} que pertenece a los dos caminos. Por lo tanto, el coste en el *modelo compacto* se asigna a la parte común de las líneas, que en este caso son L_{2a} y L_{6b}, y se asocia un coste 0 a las líneas L_{2b} y L_{6a} que son particulares de cada camino.

Aplicando los casos anteriormente descritos, al *modelo explícito*, el coste de cada camino es igual a la suma de los costes de sus líneas de memoria en el *modelo compacto*. Por ejemplo, la suma de las columnas asociadas al *modelo explícito* de los caminos $Path_1$ y $Path_2$ de la Tabla 4.1 es igual a la suma de las líneas de memoria de dichos caminos en el *modelo compacto*. Así pues, para cualquier línea de memoria tenemos una representación equivalente al coste de su primer acceso, independientemente del número de caminos que contengan dicha línea, sin añadir ningún tipo de sobrestimación.

Siguiendo con el ejemplo de la Figura 4.6, y considerando la Ecuación 4.14, se puede obtener una expresión equivalente de dicha ecuación teniendo en cuenta que si $B1 \leq B2$, obviamente $B1 + B4 \leq B2 + B4$, y por lo tanto $B1 + B4$ se podría eliminar de la ecuación, mientras que $B2 + B4$ debería permanecer. Así pues, utilizando primero el operador máximo entre $B1$ y $B2$ y después entre $B4$ y $B5$, la ecuación del WCET en el *modelo compacto* sería la siguiente:

$$\begin{aligned} WCET = \text{mín}(\ & B0 + B3 + B6 \\ & + \ \text{máx}(B1 + B4, B1 + B5,\ B2 + B4, B2 + B5)\ ) \end{aligned}$$

$$\begin{aligned} WCET = \text{mín}(\ & B0 + B3 + B6 \\ & + \ \text{máx}(\ \text{máx}(B1, B2) + B4,\ \text{máx}(B1, B2) + B5\ )\ ) \end{aligned}$$

$$WCET = \text{mín}(\ B0 + B3 + B6 + \text{máx}(B1, B2) + \text{máx}(B4, B5)\ )$$

Finalmente, si se agrupan los costes de los bloques básicos comunes a los dos caminos, es decir, $B0$, $B3$ y $B6$, la ecuación final que se obtiene todavía es mucho más reducida:

$$\begin{aligned} CmnCost &= B0 + B3 + B6 \\ WCET &= \text{mín}(\ CmnCost + \text{máx}(B1, B2) + \text{máx}(B4, B5)\ ) \end{aligned}$$

Para ilustrar esta idea con un ejemplo más complejo, se considera el flujo de control de la Figura 4.4. Las restricciones para obtener el WCET, en vez de tener en cuenta los caminos explícitos, se construyen a partir de los costes comunes a los 6 caminos indicados en la Figura 4.4. En la Tabla 4.2 se describen las líneas que recorre cada uno de los caminos de la Figura 4.4. Esta Tabla 4.2

$Path1$	L_{1a}	L_{1b}	L_2									L_{8a}	L_{8b}	L^1_{9a}	L^1_{9b}	L^1_{10}		L^1_{11b}	L^1_{12a}	L^1_{12b}
$Path2$	L_{1a}	L_{1b}	L_2									L_{8a}	L_{8b}	L^1_{9a}	L^1_{9b}		L^1_{11a}	L^1_{11b}	L^1_{12a}	L^1_{12b}
$Path3$	L_{1a}	L_{1b}		L_{3a}	L_{3b}	L_{4a}				L_{7a}	L_{7b}	L_{8a}	L_{8b}							
$Path4$	L_{1a}	L_{1b}		L_{3a}			L_{4b}	L_5		L_{7a}	L_{7b}	L_{8a}	L_{8b}	L^2_{9a}	L^2_{9b}	L^2_{10}		L^2_{11b}	L^2_{12a}	L^2_{12b}
$Path5$	L_{1a}	L_{1b}		L_{3a}			L_{4b}	L_5		L_{7a}	L_{7b}	L_{8a}	L_{8b}	L^2_{9a}	L^2_{9b}		L^2_{11a}	L^2_{11b}	L^2_{12a}	L^2_{12b}
$Path6$	L_{1a}	L_{1b}		L_{3a}					L_6	L_{7a}	L_{7b}	L_{8a}	L_{8b}							
Cmn_{All}	L_{1a}	L_{1b}										L_{8a}	L_{8b}							
$Cmn_{\{1,2\}}$			L_2											L^1_{9a}	L^1_{9b}			L^1_{11b}	L^1_{12a}	L^1_{12b}
$Cmn_{\{3,4,5,6\}}$				L_{3a}						L_{7a}	L_{7b}									
$Cmn_{\{4,5\}}$							L_{4b}	L_5						L^2_{9a}	L^2_{9b}			L^2_{11b}	L^2_{12a}	L^2_{12b}

Tabla 4.2: Clasificación de las líneas de memoria comunes y particulares de los caminos de la Figura 4.4.

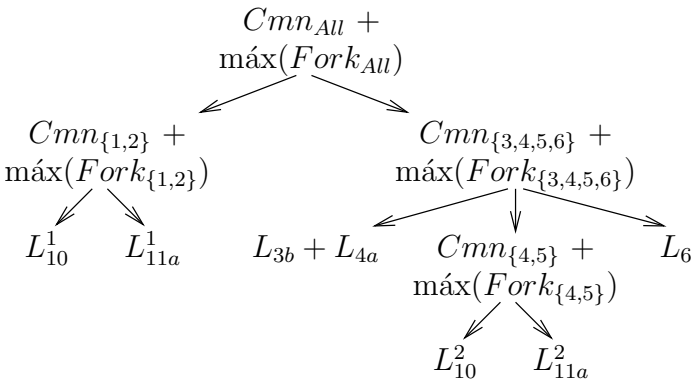


Figura 4.8: Grafo compacto de restricciones de la Figura 4.4.

se puede trasladar a un árbol como el de la Figura 4.8 que se interpreta como un AST (*Abstract Syntax Tree*) o como un CFG (*Control Flow Graph*). Las líneas de memoria aparecen sólo una vez, es decir, cada nodo es un conjunto de líneas comunes a varios caminos y cada rama representa un camino alternativo.

El conjunto de restricciones asociadas al nuevo problema ILP permite calcular el WCET ($wcet_i$) de una tarea como en la Ecuación 4.1, pero utilizando directamente el valor de los costes de ejecución de las líneas de memoria $lineCost_{i,j,k}$ sin necesidad de emplear las restricciones asociadas a cada uno de los caminos explícitos $pathCost_i$.

Por lo tanto, estas nuevas restricciones pueden sustituir a las restricciones del *modelo explícito* como se indica a continuación, evitando así tener que definir todos y cada uno de los posibles caminos de ejecución:

$$
\begin{aligned}
wcet_i &= Cmn_{All} + Fork_{All} \\
Cmn_{All} &= lineCost_{1a} + lineCost_{1b} + lineCost_{8a} + lineCost_{8b} \\
Fork_{All} &\geq Cmn_{\{1,2\}} + Fork_{\{1,2\}} \\
Fork_{All} &\geq Cmn_{\{3,4,5,6\}} + Fork_{\{3,4,5,6\}} \\
Cmn_{\{1,2\}} &= lineCost_2 + lineCost^1_{9a} + lineCost^1_{9b} + lineCost^1_{11b} + \\
&\quad lineCost^1_{12a} + lineCost^1_{12b} \\
Fork_{\{1,2\}} &\geq lineCost^1_{10} \\
Fork_{\{1,2\}} &\geq lineCost^1_{11a} \\
Cmn_{\{3,4,5,6\}} &= lineCost_{3a} + lineCost_{7a} + lineCost_{7b} \\
Fork_{\{3,4,5,6\}} &\geq lineCost_{3b} + lineCost_{4a} \\
Fork_{\{3,4,5,6\}} &\geq Cmn_{\{4,5\}} + Fork_{\{4,5\}} \\
Fork_{\{3,4,5,6\}} &\geq lineCost_6
\end{aligned}
$$

$$
\begin{aligned}
Cmn_{\{4,5\}} &= lineCost_{4b} + lineCost_5 + lineCost^2_{9a} + lineCost^2_{9b} + \\
&\quad lineCost^2_{11b} + lineCost^2_{12a} + lineCost^2_{12b} \\
Fork_{\{4,5\}} &\geq lineCost^2_{10} \\
Fork_{\{4,5\}} &\geq lineC^2_{11a} \qquad (4.15)
\end{aligned}
$$

4.4. Evaluación del algoritmo *Lock-MS*

En esta sección se evalúan las prestaciones del algoritmo *Lock-MS* para un sistema multitarea, formado por un conjunto de tareas de prioridad fija con una planificación basada en *Rate Monotonic*.

Las tareas analizadas en los experimentos realizados son las mismas que se han utilizado en trabajos anteriores [147]. Los programas considerados son los siguientes:

jfdctint: transformada discreta del coseno.

crc: comprobación de redundancia cíclica.

matmult: multiplicación de matrices.

integral: integral por intervalos.

minver: inversión de una matriz.

qurt: cálculo de las raíces de una ecuación de segundo grado.

fft: transformada rápida de Fourier.

En la Tabla 4.3 se muestran las tareas analizadas dividas en dos conjuntos denominados *small* y *medium*.

Conjunto	Tarea	WCET con-LB	Periodo	Tamaño
small	jfdctint	10108	23248	1072 B
	crc	109696	329088	536 B
	matmul	542229	2440031	208 B
	integral	716633	3583165	400 B
medium	minver	8522	19601	1360 B
	qurt	10117	30351	752 B
	jfdctint	10108	44475	1072 B
	fft	2886680	15010736	1016 B

Tabla 4.3: Conjunto de tareas: *small* y *medium*.

En un sistema cuya jerarquía de memoria está formada por un único LB, los periodos de cada tarea se han elegido para conseguir una utilización de 1,2 en los dos conjuntos de tareas *small* y *medium*. Además, el WCET y los periodos de cada conjunto de tareas siguen patrones diferentes. Por ejemplo, en el conjunto *small*, el WCET de cada tarea va creciendo de forma uniforme, al igual que sus periodos. En cambio, en el conjunto *medium*, 3 tareas tienen un WCET pequeño y sus periodos de ejecución también son pequeños, mientras que en la cuarta tarea, tanto su WCET, como su periodo son relativamente grandes. En este caso, esta tarea será expulsada muchas veces durante la ejecución del sistema. En concreto, el conjunto de tareas *medium* tiene muchos más cambios de contexto que el conjunto *small*. Esto puede servir para observar en detalle cómo influyen los cambios de contexto en la selección de líneas a bloquear en la cache por parte del algoritmo *Lock-MS*.

En los experimentos, la arquitectura considerada está formada por un procesador ARM v7 con instrucciones de 4 bytes y una jerarquía de memoria como la que se describe en la Figura 4.1. Suponemos que el procesador elegido se ha construido bajo la tecnología de 32 nm, con una velocidad de ciclo equivalente a 36 FO4[1] que podría estar alrededor de los 2.4 GHz. Este procesador representa perfectamente las características de un procesador actual de altas prestaciones para sistemas empotrados [1]. El tamaño del LB y de cada una de las líneas de la *Lockable iCache* es de 16 bytes, es decir de 4 instrucciones. En los experimentos se varía la capacidad de la cache de instrucciones desde 128 bytes a 4 KB, mientras que el tamaño de la *eSRAM* se mantiene constante en 256 KB.

Para determinar la latencia de memoria mínima, de la arquitectura propuesta, se ha utilizado Cacti V.6.0 [138]. Si la implementación se realiza con transistores de bajo consumo en reserva, se ha verificado que el tiempo de acceso a una *eSRAM* de 256 KB estará en torno a unos 7 ciclos. El coste de *fetch* de una instrucción cuando se produce un acierto, es decir cuando la instrucción está en la cache o en el LB, será de 1 ciclo. Pero si se produce un fallo, es decir si se accede a la *eSRAM*, el coste de *fetch* será de 7 ciclos. Con esta latencia de memoria se consigue estresar el funcionamiento de la *Lockable iCache*, por lo tanto los resultados obtenidos representan una cota mínima del rendimiento que se puede obtener con esta jerarquía de memoria. El coste de ejecutar una instrucción, si no se accede a memoria, será de 2 ciclos. No obstante, el coste asociado a una instrucción predicada que no se ejecuta será de 1 ciclo. Las instrucciones predicadas son instrucciones generales que sólo se ejecutan si se cumple una determinada condición. El coste de ejecución de una instrucción de acceso a memoria, instrucciones *load* y *store*, tendrá un incremento adicional de 7 ciclos, ya que los accesos a datos se sirven directamente desde la *eSRAM*.

En los experimentos realizados se calcula el WCET de cada una de las ta-

[1]Un FO4 (*A fan-out-of-4*) representa el retardo de propagación de un inversor cuando la carga de trabajo es 4 veces la suya propia.

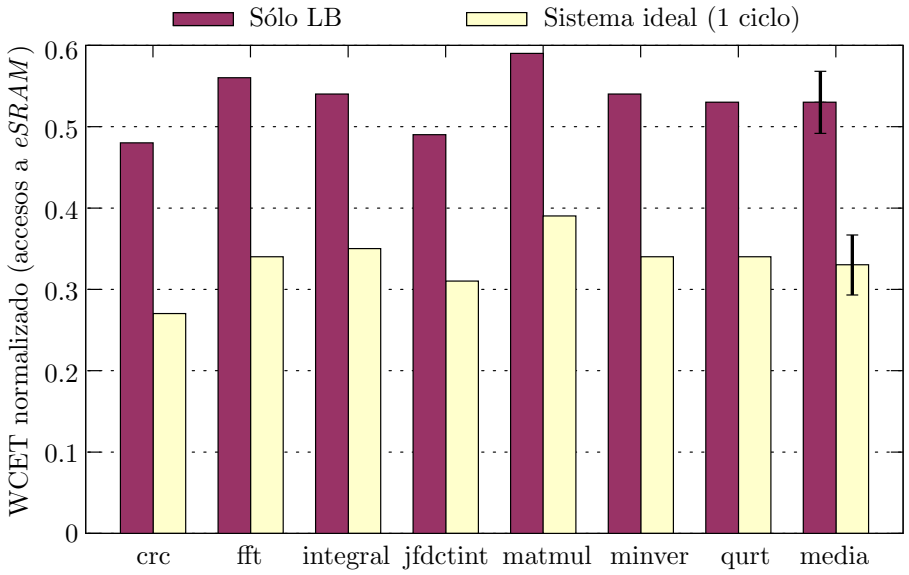


Figura 4.9: WCET de cada tarea normalizado al WCET calculado cuando se accede directamente a la *eSRAM*

reas de la Tabla 4.3, en tres configuraciones distintas de la jerarquía de memoria propuesta. En primer lugar se supone que todos los accesos, a instrucciones y a datos, se realizan directamente desde la *eSRAM*. El WCET obtenido representa el límite superior, ya que se consideran todos los accesos como fallos. En segundo lugar, se considera únicamente el LB. Esta configuración de la jerarquía de memoria permite determinar el impacto de la localidad espacial en el WCET. Finalmente, se supone que todos los accesos a instrucciones tienen un coste de un solo ciclo, es el caso ideal, es decir el WCET obtenido representa el límite inferior, ya que se considera que todos los accesos son aciertos.

En la Figura 4.9 se muestra el WCET de cada tarea, obtenido en los experimentos propuestos. El WCET de la Figura se ha normalizado al WCET calculado cuando todos los accesos se realizan directamente a la *eSRAM*.

Localidad espacial

Si se analiza el WCET de cada tarea obtenido en la jerarquía de memoria que dispone únicamente de un LB, en la Figura 4.9 se observa que el impacto de la localidad espacial es bastante significativo.

De media, en un sistema con un simple LB, el WCET se reduce en un 47 % con respecto al WCET obtenido en un sistema cuando todos los accesos se realizan a la *eSRAM*. Esta reducción en el WCET de la tarea representa un

incremento de aproximadamente 1,8 veces en su velocidad de ejecución (*speed-up*). En un sistema ideal, donde se consideran todos los accesos a la memoria de instrucciones como aciertos de cache, se llega a conseguir de media una reducción del 67 % en el WCET de las tareas. En este caso, la reducción en el WCET de la tarea representa un incremento de unas 3 veces en su *speed-up*. Así pues, con respecto a un sistema con LB, el máximo incremento en el *speed-up* de una tarea podría llegar a ser de 1,6 veces cuando se disponga de una *Lockable iCache*.

Para verificar los efectos de la localidad espacial en el WCET, también determinamos la utilización de la CPU en un sistema multitarea. Es decir, se calcula la fracción de tiempo que el procesador estará ocupado ejecutando, en el peor caso, el conjunto de tareas que forman el sistema.

$$U = \sum_{i=1}^{NTasks} \frac{Wcost_i}{T_i}$$

Como ya se ha indicado anteriormente, los periodos de cada tarea se han propuesto con el fin de conseguir una utilización de 1,2, para los dos conjuntos de tareas *small* y *medium*, en un sistema cuya jerarquía de memoria está formada por un único LB. Pero, si todos los accesos a memoria se realizan directamente desde la *eSRAM*, la utilización del procesador para los conjuntos *small* y *medium* sobrepasa los valores 2,33 y 2,24 respectivamente. Mientras, en el caso ideal en el que todos los accesos a las instrucciones son aciertos, la utilización del procesador es inferior a 0,75 para el conjunto *small* e inferior a 0,72 para el conjunto *medium*.

Por lo tanto, se puede afirmar que un simple LB optimiza las prestaciones de un sistema de tiempo real, a un coste muy bajo, ya que mejora, tanto el WCET de cada tarea, como la utilización del procesador y la *planificabilidad* del sistema. Además, la mejora puede ser incluso mayor, si se considera la *Lockable iCache* propuesta en la jerarquía de memoria de la Figura 4.1 para capturar también la localidad temporal.

Selección de líneas: *Lock-MU* vs. *Lock-MS*

En la siguiente sección se compara el rendimiento del algoritmo de baja complejidad *Lock-MU* (*Algorithm for Minimize Utilization*) [147], que selecciona las líneas de memoria a fijar en la cache durante toda la vida del sistema, con el rendimiento del algoritmo *Lock-MS* propuesto en la Sección 4.2 anterior. Ambos algoritmos se han diseñado para un sistema de tiempo real multitarea y para una jerarquía de memoria formada por un LB y una *Lockable iCache* como la que se describe en la Figura 4.1.

Los conjuntos de tareas utilizados en la comparativa ya se presentaron en la Tabla 4.3. En esta comparativa se supone, sin pérdida de generalidad, que el código de todas las tareas comienza en la misma dirección de memoria, de tal forma que dicha dirección de memoria se corresponda con el conjunto 0 de la cache de instrucciones.

El algoritmo *Lock-MU* selecciona el conjunto de líneas de memoria de todas las tareas del sistema a las que mayor número de veces se accederá durante la ejecución. Este conjunto de líneas de memoria se carga en la cache al iniciar la ejecución del sistema y permanece bloqueado durante toda la ejecución del mismo. Este algoritmo es un ejemplo de las técnicas denominadas en la literatura como *static locking cache*. Por lo tanto, el sistema no sufre ningún tipo de penalización en los cambios de contexto, pero las tareas no pueden utilizar toda la cache.

El algoritmo *Lock-MS* selecciona un conjunto de líneas de memoria de cada tarea del sistema. En cada cambio de contexto, la cache se carga con las líneas de la tarea que se va a ejecutar, por lo tanto se debe añadir una penalización por la carga de estas líneas en la cache. No obstante, las tareas pueden utilizar toda la cache. Este algoritmo es un ejemplo de las técnicas denominadas en la literatura como *dynamic locking cache*.

La comparación realizada no sería correcta si sólo se comparase el WCET de las tareas analizadas, ya que los periodos combinados con el WCET de cada una de ellas determinan un número diferente de cambios de contexto para cada uno de los algoritmos estudiados. Si el WCET de una tarea es pequeño, ésta sufre menos expulsiones, y por lo tanto el tiempo de respuesta disminuye. Además, los cambios de contexto influyen de forma diferente en los algoritmos *Lock-MS* y *Lock-MU*, por lo tanto es necesario verificar cómo afecta este parámetro al funcionamiento del sistema.

En los experimentos realizados, se compara la *planificabilidad* del sistema para los dos conjuntos de tareas en función del tamaño y asociatividad de la cache utilizada. También se compara el tiempo de respuesta de la tarea con la prioridad más baja, ya que su WCET depende en gran medida de la ejecución del resto de las tareas del sistema. En la Figura 4.10 se muestran los resultados experimentales obtenidos para cada una de las diferentes configuraciones de cache estudiadas. Representamos como *speed-up* del tiempo de respuesta el cociente entre el periodo de la tarea de menor prioridad y su tiempo de respuesta. Esta métrica proporciona el tiempo excedido por la tarea cuando el sistema no es *planificable*. Como se observa, se varía tanto la capacidad como la asociatividad de la cache. Por ejemplo, la capacidad de cache varía desde 128 bytes a 2 KB para el conjunto de tareas *small*, y desde 256 bytes a 4 KB para el conjunto de tareas *medium*. La asociatividad analizada es de correspondencia directa, o bien puede tener dos o cuatro vías, o bien puede ser totalmente asociativa.

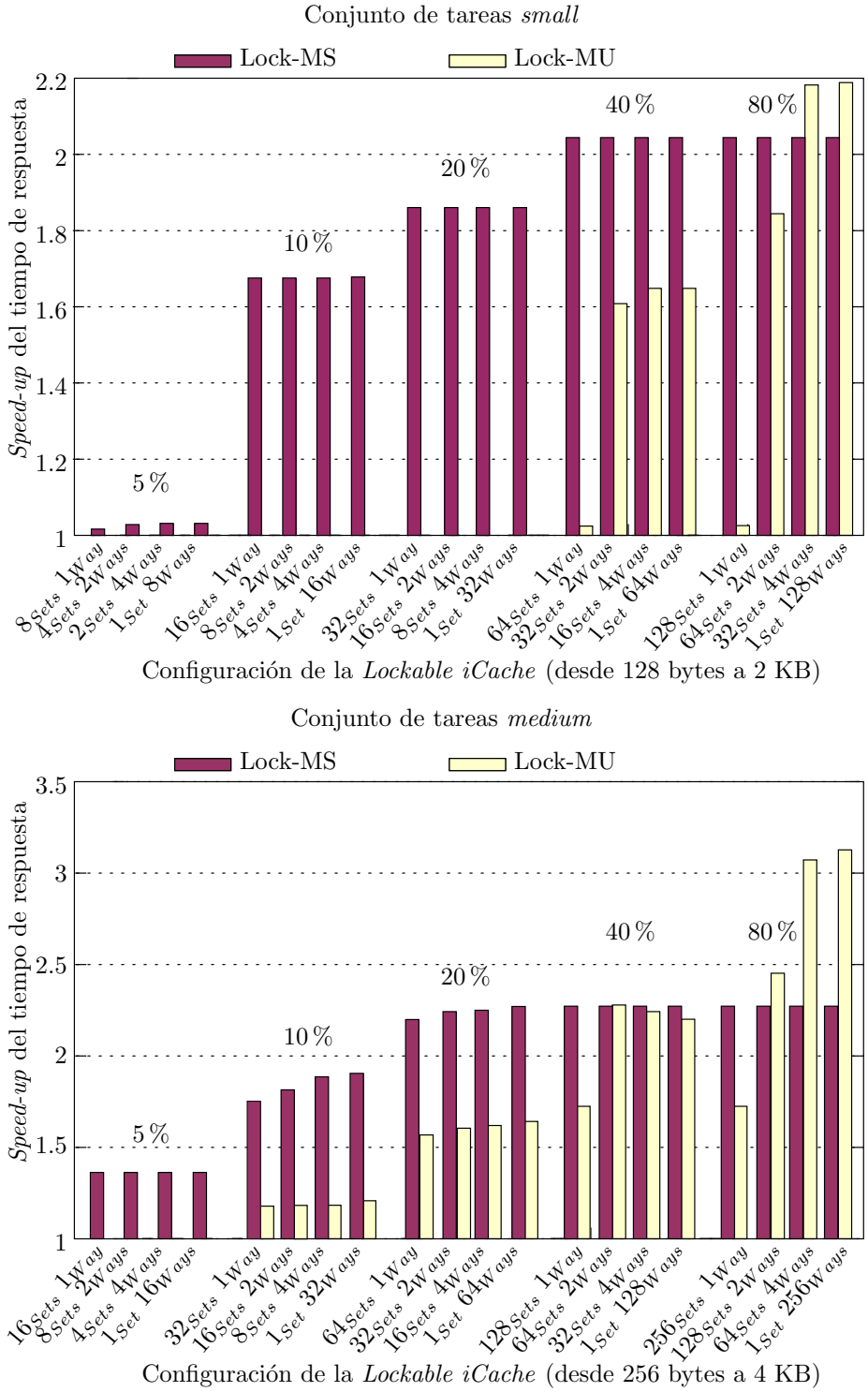


Figura 4.10: *Speed-up* del tiempo de respuesta para los conjuntos de tareas *small* y *medium*.

En primer lugar, se debe indicar que el algoritmo *Lock-MS* consigue que el sistema sea *planificable* con una capacidad de cache de aproximadamente el 5 % del tamaño en instrucciones del conjunto de tareas analizado. Mientras que el algoritmo *Lock-MU*, para conseguir que el sistema sea *planificable*, necesita que el tamaño de la cache sea de más de un 40 % del tamaño del conjunto de tareas *small* y de un 10 % del tamaño del conjunto de tareas *medium*. El porcentaje de la capacidad de la cache con respecto al tamaño del conjunto de tareas se muestra en la parte superior de cada grupo de configuraciones de cache estudiado. Además, el algoritmo *Lock-MS* obtiene mejor tiempo de respuesta que el algoritmo *Lock-MU*, para ambos conjuntos de tareas *small* y *médium* y para cualquier capacidad de la cache, excepto las más grandes, que alcanzan el 80 % del tamaño del código, es decir, cuando ya cabe prácticamente todo el código del programa en la cache. Así pues, como se observa en la Figura 4.10, el *speed-up* del tiempo de respuesta obtenido con el algoritmo *Lock-MS* casi siempre es mejor que el conseguido con *Lock-MU*. Por lo tanto, con este hardware sencillo el algoritmo *Lock-MS* obtiene mejores prestaciones que el algoritmo *Lock-MU*.

Obviamente, el número de cambios de contexto influye considerablemente en las prestaciones de un sistema que utiliza el algoritmo *Lock-MS*, ya que en cada cambio de contexto las líneas seleccionadas de la tarea que reanuda su ejecución deben ser cargadas y bloqueadas en la cache. Mientras que los cambios de contexto no afectan a las prestaciones de un sistema que utiliza el algoritmo *Lock-MU*. El número de cambios de contexto por el tiempo de respuesta del sistema se ha calculado teniendo en cuenta el número de cambios de contexto que se producen cuando se lanzan a ejecución todas las tareas del sistema a la vez, hasta que la CPU queda ociosa, y se ha dividido por el tiempo transcurrido hasta ese momento. De media, el número de cambios de contexto obtenido es $2,7$ veces más grande para el conjunto de tareas *medium* que para el conjunto *small*. Por este motivo, el algoritmo *Lock-MU* proporciona mejor rendimiento para el conjunto de tareas *medium* que para el conjunto de tareas *small*.

Por ejemplo, sin tener en cuenta los cambios de contexto, en el sistema ideal donde todos los accesos tienen un coste de un ciclo, el *speed-up* del tiempo de respuesta obtenido es de $2,28$ y $3,14$ para los conjuntos de tareas *small* y *medium* respectivamente. Por lo tanto, en una comparación justa con un sistema que utiliza el algoritmo *Lock-MS*, se debe considerar el coste de cargar todas las líneas de memoria en la cache, ya que esto permite obtener siempre un acierto en todos los accesos a la cache de instrucciones. Así pues, en el sistema ideal, cuando se consideran los cambios de contexto, el *speed-up* del tiempo de respuesta pasa a ser de $2,19$ y $2,55$ para los conjuntos de tareas *small* y *medium* respectivamente. Como conclusión, se puede afirmar que la penalización por los cambios de contexto puede variar entre el 4 % y 19 % del *speed-up* total, para los conjuntos *small* y *medium* respectivamente.

Otra conclusión interesante que se pone de manifiesto en la mayor parte de los casos, es que el algoritmo *Lock-MS* no es sensible al grado de asociatividad de

la cache, ya que *Lock-MS* sólo consigue aprovechar la asociatividad con tamaños de cache grandes, de 512 bytes y 1 KB y para el conjunto de tareas *medium*. Por lo tanto, este algoritmo se puede utilizar en sistemas con cache de correspondencia directa, sin perder prestaciones. Al contrario, el algoritmo *Lock-MU* es especialmente sensible a esta característica de la cache. Por ejemplo, se puede observar que el *speed-up* del tiempo de respuesta aumenta en función de la asociatividad, para ambos conjuntos de tareas. *Lock-MU* aprovecha la asociatividad de la cache, principalmente para poder cargar en el mismo conjunto líneas de memoria de diferentes tareas. Aunque al aumentar la asociatividad se puede incrementar el tiempo de acceso y el consumo de energía, este problema se podría evitar situando el código de cada tarea en las posiciones de memoria más adecuadas.

Finalmente, conviene señalar que con caches de tamaño grande, por ejemplo para una cache con una capacidad del 80 % del tamaño del conjunto de tareas, los resultados de la Figura 4.10 muestran que el algoritmo *Lock-MU* supera en prestaciones al algoritmo *Lock-MS*. Esto es debido, principalmente, a que las penalizaciones por la carga de la cache crecen significativamente en los cambios de contexto. Es decir, cuando todas las líneas de memoria de las tareas del conjunto estudiado caben en la cache, es mejor fijar el contenido de la misma durante toda la vida del sistema, en vez de cargar los contenidos de cada tarea en cada cambio de contexto.

Caches *convencionales* vs. *Lock-MS*

Los métodos que analizan el comportamiento en el peor caso de las caches *convencionales* determinan el WCET de una tarea aislada, por lo tanto, para que los resultados obtenidos sean seguros, es necesario añadir posteriormente el coste de las interferencias extrínsecas de cache. Sólo añadiendo esta información se puede determinar, de forma segura, si un sistema con una cache *convencional* es *planificable*. Pero el coste computacional de todos estos métodos de análisis suele ser tan grande que su aplicación se hace prácticamente imposible [7, 107, 197].

En esta sección se reflejan los resultados obtenidos con el algoritmo *Lock-MS* en una jerarquía de memoria como la propuesta en la Figura 4.1 y los resultados obtenidos mediante el método de *poda dinámica de caminos* [7], descrito con todo detalle en el Capítulo 3, en una cache *convencional*. No obstante, conviene indicar que el funcionamiento de estas dos jerarquías de memoria es muy diferente. Ambos métodos de análisis proporcionan resultados seguros y exactos del WCET en sus respectivas jerarquías de memoria. Pero, mientras el método de *poda dinámica de caminos* presenta limitaciones en códigos con muchos condicionales dentro de un bucle, el algoritmo *Lock-MS* en su versión *modelo compacto* puede analizar cualquier tipo de código. Por otro lado, el algoritmo *Lock-MS* permite analizar un sistema multitarea completo considerando los

posibles cambios de contexto, mientras que el método de poda sólo permite analizar una tarea de forma aislada.

El conjunto de tareas utilizado en los experimentos ya se ha presentado en la Tabla 4.3. El método de poda puede analizar este conjunto de tareas y obtener el WCET exacto en poco tiempo. En los experimentos realizados sólo hemos analizado una cache de correspondencia directa. La asociatividad no influye de forma significativa, ni en los resultados que proporciona el método de *poda dinámica de caminos*, ni tampoco en los resultados obtenidos mediante el algoritmo *Lock-MS*. Por lo tanto, un aumento de la asociatividad de la cache no aporta una mejora relevante en los resultados logrados por ambos métodos.

En primer lugar, hemos analizado el WCET de las tareas de los conjuntos *small* y *medium*. Los resultados muestran que el WCET de las tareas obtenido con el método de poda, en presencia de una cache de instrucciones *convencional*, es equivalente al WCET conseguido mediante el algoritmo *Lock-MS* en una jerarquía de memoria formada por un LB y una *Lockable iCache*. Las diferencias del WCET de cada tarea calculado mediante estas dos técnicas varían poco, entre un $-3{,}8\,\%$ y un $7{,}4\,\%$.

También hemos analizado el *speed-up* del tiempo de respuesta de los conjuntos de tareas *small* y *medium*. Con el fin de determinar el coste de las interferencias extrínsecas de la cache, para cada tarea se ha tenido en cuenta el peor caso de expulsión, es decir se ha considerado el número máximo de líneas de memoria que puede tener cada tarea en la cache. En la Figura 4.11 se muestra el *speed-up* del tiempo de respuesta de la tarea de menor prioridad del conjunto de tareas *small*. Como se observa, el comportamiento de ambos métodos de análisis es equivalente. Además, al aumentar el tamaño de la cache, entre el $20\,\%$ y el $40\,\%$ del tamaño del código del conjunto de tareas analizado, no se obtiene una mejora significativa en ninguno de los dos métodos.

En particular, con caches pequeñas el algoritmo *Lock-MS* funciona mejor que el método de *poda dinámica*. Por ejemplo, su rendimiento es mejor si la capacidad de la cache varía entre el 5 y el $10\,\%$ del tamaño del código del conjunto de tareas considerado. Esto es debido a que las interferencias intrínsecas de cache son mayores en caches *convencionales* de pequeño tamaño, mientras que estas interferencias desaparecen al fijar el contenido de la cache. Cuando la capacidad de la cache está entre el 20 y el $80\,\%$ del tamaño del código del conjunto de tareas considerado, el rendimiento es similar con ambas técnicas de análisis, ya que las diferencias entre ambos métodos son inferiores al $5\,\%$. Esta tendencia también se mantiene cuando se analiza el conjunto de tareas *medium*. No obstante, en este caso y debido al mayor número de cambios de contexto, el sistema sólo es *planificable* si la capacidad de la cache *convencional* está entre el 40 y el $80\,\%$ del tamaño del código del conjunto de tareas considerado, y el *speed-up* del tiempo de respuesta es inferior a $1,1$, por eso no se ha presentado la gráfica de resultados.

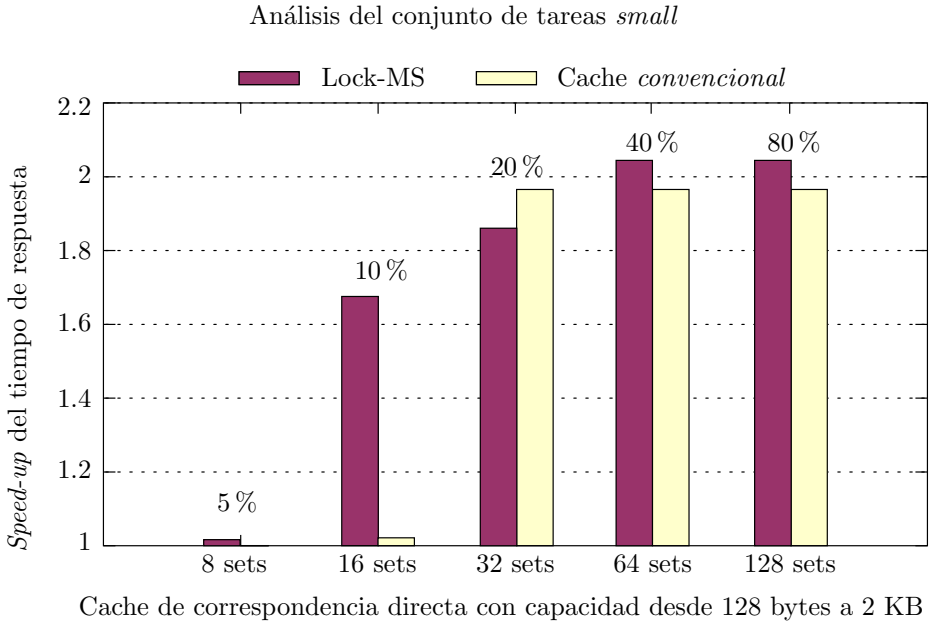


Figura 4.11: Comportamiento de *Lock-MS* vs. caches *convencionales*.

En definitiva, estos dos métodos de análisis y cálculo del WCET, tanto el método de *poda dinámica de caminos*, como el algoritmo *Lock-MS*, en las jerarquías de memoria particulares para las que se han diseñado cada uno de ellos, proporcionan un WCET equivalente en las tareas analizadas.

Coste computacional de *Lock-MS*

El coste computacional del análisis de los conjuntos de tareas *small* y *medium* de la Tabla 4.3 considerados en los experimentos no es relevante, ya que la solución del problema ILP se obtiene en unos pocos milisegundos. No obstante, el coste computacional del algoritmo *Lock-MS* depende de la estructura y del tamaño de las tareas analizadas, y del *solver* que se utiliza para solucionar el problema ILP.

Para observar de una forma más exacta el coste computacional del algoritmo *Lock-MS*, se ha creado una colección de tareas sintéticas en las que el análisis es más complejo. Se trata de un conjunto de tareas, cuyo tamaño varía entre 16 KB y 96 KB, que se han diseñado con diversas estructuras condicionales *if-then-else* consecutivas para conseguir aproximadamente unos 2^{216} posibles caminos de ejecución. Todas estas tareas sintéticas se han analizado en la jerarquía de memoria que se describe en la Figura 4.1. En la Tabla 4.4 se resume

el conjunto de experimentos realizados para los que se han considerado tres tamaños de cache. En total se han realizado 33 experimentos en un Intel Xeon de 64-bits a 2 GHz. El *solver* utilizado ha sido *lp_solve* versión 5.5.0.14 con las opciones por defecto.

Tareas		iCache (64 conjuntos)	
Caminos	Tamaño(KB)	Vías	Capacidad(KB)
2^{36}	16	$4, 8, 12$	$4, 8, 12$
2^{72}	32	$8, 16, 24$	$8, 16, 24$
2^{108}	48	$12, 24, 36$	$12, 24, 36$
2^{144}	64	$16, 32, 48$	$16, 32, 48$
$2^9, 2^{18}, 2^{30}, 2^{54}, 2^{108}, 2^{162}, 2^{216}$	96	$24, 48, 72$	$24, 48, 72$

Tabla 4.4: Espacio experimental para los programas sintéticos.

En los experimentos efectuados se considera la función a minimizar *Wcost* de la Ecuación 4.11 y para el cálculo del WCET de cada tarea el conjunto de restricciones de la Ecuación 4.15. El *solver* obtiene la solución real muy rápidamente, ya que el espacio de soluciones es continuo. Ésta es la solución óptima del problema, pero puede que en algunas ocasiones no sea válida porque no es entera. Si la solución real no es válida, el *solver* obtiene una solución entera en muy poco tiempo. Esta solución no suele ser la óptima, pero suele aproximarse bastante. No obstante, el *solver* sigue verificando otras soluciones hasta que encuentra la óptima o se da por finalizada la resolución del problema al sobrepasar el límite de tiempo indicado. Si la diferencia entre una solución entera y la solución real es pequeña, será muy difícil que la solución entera se pueda mejorar, ya que podría ser ya la óptima. Por lo tanto, una solución entera del problema sólo se puede mejorar si la diferencia entre la solución entera y la solución real es grande.

En la Figura 4.12 se muestra la distribución acumulada de las diferencias entre la primera solución entera y la solución real del problema asociada a los experimentos realizados. En el eje X se representan las diferencias entre ambas soluciones y en el eje Y se representa el número de ocurrencias. Como se observa, en el 28 % de los casos la diferencia es 0, es decir la primera solución entera y la solución real coinciden. Para el resto, la diferencia está por debajo del 0,45 %. Es decir, en el 72 % de los casos, la sobrestimación en el WCET que se produciría utilizando la primera solución entera, sería de unos 5 ciclos de procesador por cada 1000 utilizados. Además, puesto que la solución real del problema no es válida, también es posible que la primera solución entera encontrada por el *solver* sea la óptima.

En la Figura 4.13 se muestra una comparativa entre el tiempo de análisis de ambas soluciones, la primera solución entera y la solución real. En el eje X de la gráfica se representa el tamaño de cada tarea analizada, mientras que en el eje Y se indica el tiempo de análisis. Los tamaños base de la *iCache* con-

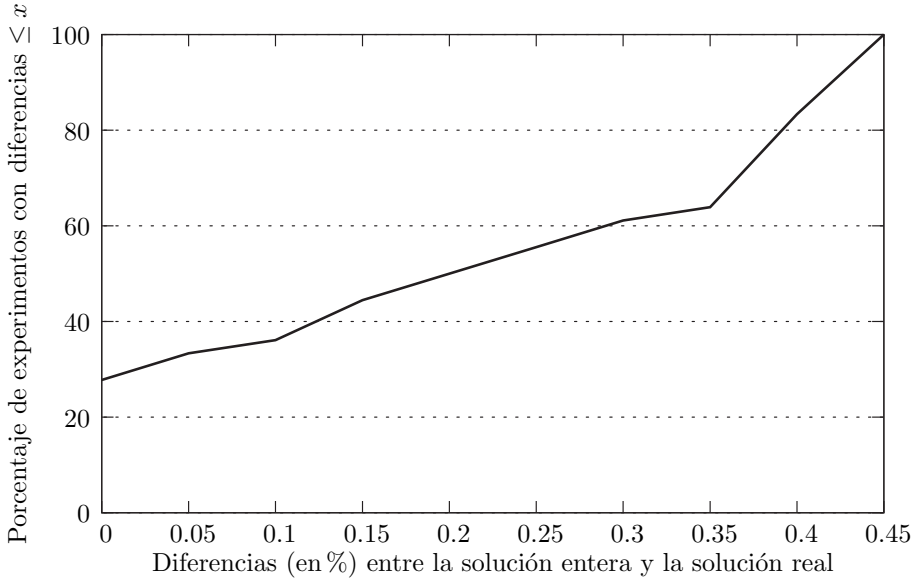


Figura 4.12: Distribución de diferencias entre las soluciones enteras y las reales.

siderados en los experimentos son 4, 8 y 12 KB. Así pues, para cada tarea se han analizado tres tamaños de cache, y el tiempo de análisis de cada solución se ha representado con la misma marca. En la Figura 4.13 también se muestra la curva de tendencia que sigue el tiempo de análisis en función del tamaño de las tareas. Se observa de forma clara que el tiempo de análisis crece en función de la complejidad del problema. En la parte superior izquierda también aparecen representadas las ecuaciones de las curvas de tendencia del tiempo de análisis de ambas soluciones. Por lo tanto, el tiempo de análisis crece de forma cuadrática en función de la complejidad, es decir, en función del tamaño del código, del número de caminos de la tarea y del tamaño de cache considerado.

Para finalizar este análisis sobre el coste computacional del algoritmo *Lock-MS*, en la Figura 4.14 se muestra un estudio equivalente para observar la influencia del número de caminos en el tiempo de análisis del problema. Para realizar este experimento se ha analizado la tarea sintética de mayor tamaño (96 KB) con un número de caminos que varía desde 2^9 hasta 2^{216}. La cache considerada es de 64 conjuntos con 24, 48 y 72 vías para tener un tamaño total de cache de 24, 48 y 72 KB respectivamente. La gráfica de la Figura 4.14 indica que el tiempo de análisis no presenta una clara tendencia ascendente con respecto al número de caminos de la tarea, esto significa que el número de caminos no aumenta el tiempo de análisis. Pero además, el tiempo de análisis de programas más grandes y con mayor número de caminos también es pequeño. Por ejemplo, analizar algunas tareas sintéticas de más de 96 KB de tamaño, con un número de caminos mayor de 10^{65} en una cache asociativa de 72 vías, ha costado menos de

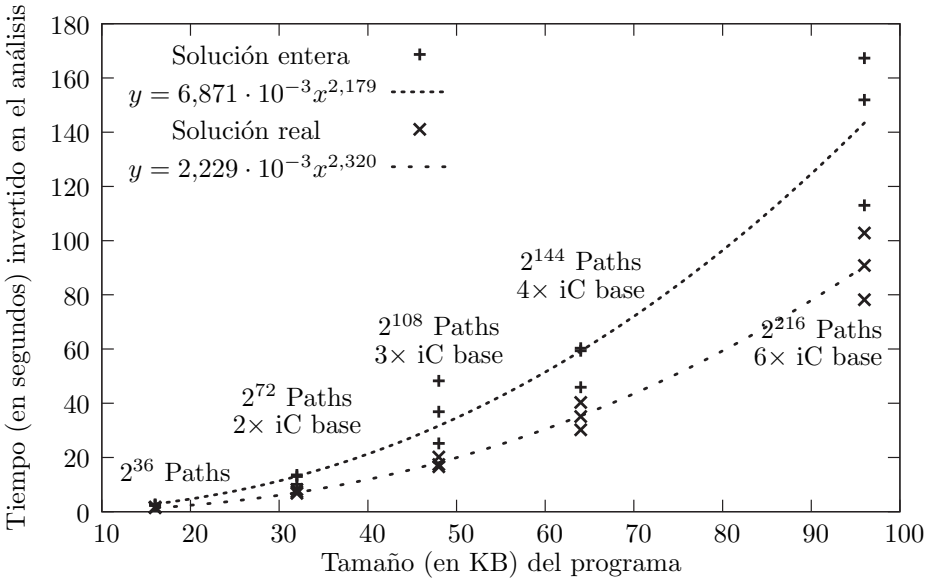


Figura 4.13: Tiempo de análisis en función de la *Lockable iCache* y del tamaño del programa.

3 minutos. En particular, este tiempo es relativamente pequeño en comparación con el de otros modelos de análisis y cálculo del WCET basados en ILP, que han sido criticados por tener un coste computacional demasiado grande [107, 197].

4.5. Conclusiones

En este capítulo se analiza el comportamiento en el peor caso de una jerarquía de memoria formada por un LB y una *Lockable iCache* (ver Figura 4.1). Para obtener el mejor rendimiento de esta jerarquía de memoria, en un sistema de tiempo real multitarea, se ha propuesto el algoritmo *Lock-MS*, que obtiene las líneas de memoria de cada tarea del sistema que se cargarán y fijarán durante su ejecución en la *Lockable iCache*.

El algoritmo *Lock-MS* está basado en ILP y su objetivo es obtener la máxima *planificabilidad* del sistema en esta jerarquía de memoria, teniendo en cuenta además, tanto el WCET de cada tarea, como el coste de los cambios de contexto del sistema.

El algoritmo *Lock-MS* no es especialmente sensible a la asociatividad, por lo tanto este algoritmo puede obtener una buena *planificabilidad* del sistema con caches de correspondencia directa. Además, con caches de capacidad pequeña, comprendidas entre el 5 % y 10 % del código del sistema, el algoritmo *Lock-MS* consigue que el sistema sea *planificable*. En sistemas multitarea con expulsiones,

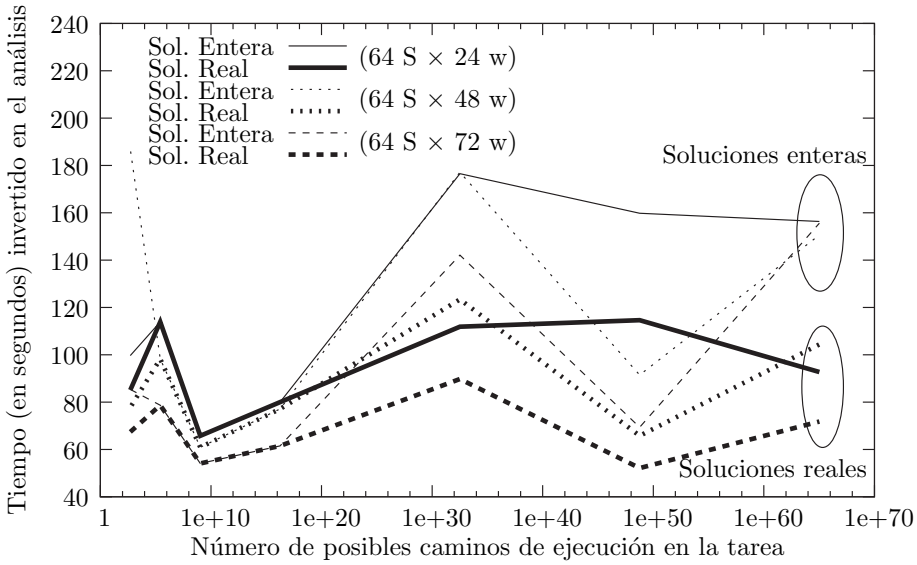


Figura 4.14: Tiempo de análisis en función del número de condicionales.

el coste de los cambios de contexto es determinante para lograr buenas prestaciones, pero debido a este coste, cuando la capacidad de la cache es grande, de aproximadamente un 80 % del código del sistema, algoritmos que bloquean la cache durante toda la vida del sistema como *Lock-MU* [147] pueden superar en rendimiento al algoritmo *Lock-MS* presentado.

Los resultados obtenidos muestran que el redimiendo, en el peor caso, de una jerarquía de memoria formada por un LB y una *Lockable iCache* puede ser incluso mejor que el rendimiento de una cache de instrucciones *convencional*. Por lo tanto, la utilización de esta jerarquía de memoria está totalmente justificada en sistemas de tiempo real. También conviene recordar que la *Lockable iCache* es totalmente predecible y evita la complejidad exponencial de los condicionales dentro de bucles. Por otra parte, el LB captura muy bien la localidad espacial mejorando considerablemente el rendimiento de la jerarquía de memoria.

El método *Lock-MS* tiene un coste computacional relativamente bajo, pero, cuando el número de caminos del programa es grande, representar todos estos caminos mediante restricciones lineales no es posible. No obstante, también hemos propuesto un *modelo compacto* de *Lock-MS* que permite reducir el número de caminos del problema ILP, sin perder precisión en el WCET obtenido. El tiempo de resolución del problema ILP para el *modelo compacto* crece aproximadamente de forma cuadrática con respecto a la complejidad del problema, principalmente en función del tamaño de las tareas analizadas. Esto permite analizar códigos grandes en un tiempo relativamente pequeño.

Capítulo 5

Una jerarquía de memoria para sistemas de tiempo real

En la actualidad, uno de los mayores costes en la ejecución de una instrucción sigue siendo su búsqueda en la memoria. Las técnicas de prebúsqueda tratan de llevar a la CPU un nuevo bloque de la memoria antes de que sea referenciado. La prebúsqueda intenta predecir los futuros accesos a memoria, para ocultar la latencia en dichos accesos. Así pues, durante la ejecución se solicita, de forma especulativa, un nuevo bloque de memoria al siguiente nivel de la jerarquía. El hardware de prebúsqueda es sencillo y no necesita el soporte del software, ni para decidir la solicitud de un bloque de memoria, ni para indicar el instante en el que se debe traer dicho bloque al procesador para mejorar su rendimiento [113, 156, 179].

Con objeto de elegir los bloques de memoria que se llevarán al procesador, se utilizan algoritmos basados en algún tipo de correlación asociada a cierta información recogida durante la ejecución. Así por ejemplo, los fallos de cache durante los accesos a memoria ayudan a resolver si un determinado bloque de memoria se solicitará de forma especulativa [86, 167]. En estos casos la prebúsqueda necesita guardar en tablas esta información para decidir si un determinado bloque se solicita a memoria o no.

Otra forma de prebúsqueda más sencilla es simplemente solicitar el siguiente bloque de memoria. Esta técnica conocida como prebúsqueda secuencial se basa en algunas de las siguientes políticas:

- Ordenar la lectura de la línea de memoria $memLine_{i+1}$ siempre que se realiza un acceso a la línea $memLine_i$ (*next-line always*) [85].
- Si se produce un fallo en el acceso a una línea de memoria, se ordena

también la lectura de la línea siguiente (*next-line on miss*) [143].

- Cada línea de memoria tiene asignado un bit de estado que durante la prebúsqueda está a cero. Cuando se accede por primera vez, y se produce un acierto, el bit de estado cambia y se ordena la lectura de la siguiente línea de memoria (*next-line tagged*) [164].

Todos estos esquemas se pueden extender en grado, es decir, se puede aumentar el número de líneas que se solicitarán así como la distancia entre las líneas a prebuscar [164, 143].

En la literatura es habitual encontrar técnicas de análisis de la cache de instrucciones en el peor caso. Algunas de estas técnicas incluyen un sencillo LB (*Line Buffer*) para mejorar el rendimiento de la jerarquía de memoria, ya que permite explotar la localidad temporal a un coste reducido, y su análisis no presenta dificultades relevantes. Pero en sistemas de tiempo real, el hardware de prebúsqueda no se ha utilizado porque es difícil modelar estáticamente su comportamiento. Además, el hardware de prebúsqueda poluciona la cache aumentando la dificultad de predecir su funcionamiento.

En este capítulo se introduce una importante mejora en la arquitectura de memoria descrita en el Capítulo 4 (ver Figura 4.1). Se trata de incorporar un sencillo almacén de prebúsqueda que se actualiza con la siguiente línea de memoria del programa a la que se accederá. El resultado es una nueva jerarquía de memoria, cuyo comportamiento temporal en el peor caso se puede modelar de una forma muy precisa, y cuyas prestaciones son ideales para un sistema de tiempo real. Al considerar una cache que bloquea su contenido durante la ejecución, desaparece el problema de la polución, ya que la prebúsqueda no puede modificar el contenido de la cache. Además, al combinar en la jerarquía de memoria un LB (*Line Buffer*) y un PB (*Prefetch Buffer*) se reduce considerablemente el tamaño de la cache de instrucciones y también aumenta la *planificabilidad* del sistema.

Para conseguir el máximo rendimiento de esta nueva jerarquía de memoria que proponemos, es necesario actualizar el algoritmo *Lock-MS* (Lock for Maximize Schedulability) que selecciona las líneas a fijar en la cache, para que la *planificabilidad* del sistema sea máxima.

En la siguiente sección se describe la nueva jerarquía de memoria que proponemos para sistemas de tiempo real y se explica su funcionamiento.

5.1. Jerarquía de memoria con prebúsqueda

Como se observa en la Figura 5.1, la nueva jerarquía de memoria está formada por tres componentes: una *Lockable iCache*, un LB (*Line Buffer*) y un PB (*Pre-*

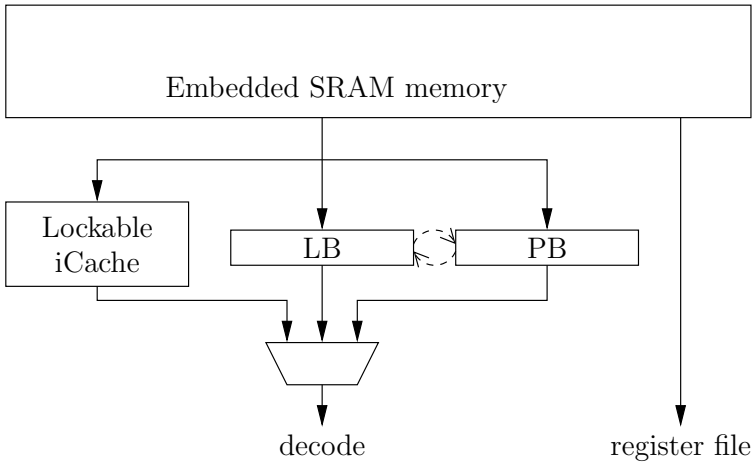


Figura 5.1: Jerarquía de memoria para sistemas de tiempo real con prebúsqueda.

fetch Buffer). Su funcionamiento es totalmente predecible y su análisis temporal se puede formular como un problema ILP equivalente al problema del Capítulo 4.

A continuación describimos cada uno de los componentes de esta nueva jerarquía de memoria y explicamos su funcionamiento general.

- **Una *Lockable iCache***

La *Lockable iCache* permite aprovechar la localidad temporal, como ya se ha descrito en el Capítulo 4. Se trata de una cache de instrucciones que puede fijar su contenido. Por lo tanto su comportamiento es totalmente predecible, ya que todo tipo de interferencias de cache desaparecen. La cache quedará bloqueada durante la ejecución de cada tarea y se podrá actualizar su contenido en cada cambio de contexto.

- **Un *Line Buffer***

Un LB captura la localidad espacial. Se trata de un pequeño almacén de línea de instrucciones que mejora considerablemente el rendimiento de la jerarquía de memoria, con un coste mínimo. El LB es un componente habitual en procesadores empotrados, y su funcionamiento se podría decir que es equivalente al funcionamiento de una cache con una única línea.

- **Un almacén de prebúsqueda: *Prefetch Buffer***

Dispone de un almacén de prebúsqueda (PB/ *Prefetch Buffer*) para mejorar todavía más la localidad espacial. El PB captura de forma especulativa la siguiente línea física de memoria. Se trata de una implementación particular de

prebúsqueda secuencial basada en la política *next-line tagged*. La prebúsqueda generará un acierto si se accede a la siguiente línea de memoria en secuencia; pero si se realiza un salto en la secuencia habitual de ejecución, se producirá un fallo. Además, el hardware de prebúsqueda no poluciona la cache en ningún instante, ya que su contenido está bloqueado durante la ejecución de cada tarea.

- **Funcionamiento**

Durante la etapa de *fetch* de una instrucción se realiza una búsqueda en paralelo en los tres componentes, es decir, en la *Lockable iCache*, en el LB y en el PB. Si se produce un acierto en alguna de estas tres estructuras, las instrucciones se sirven en un ciclo de procesador. Pero si se produce un fallo, es necesario solicitar la línea de memoria al siguiente nivel de la jerarquía de memoria, que en este caso consiste en una *eSRAM* para sistemas empotrados de altas prestaciones. Posteriormente el LB se actualizará con la línea de memoria solicitada. En la Figura 5.2 se muestra un esquema de las operaciones de la prebúsqueda en la etapa de *fetch* de una instrucción cuando la *Lockable iCache* dispone de un puerto dual.

Para respaldar la prebúsqueda secuencial que se propone, tanto la *Lockable iCache*, como el LB y el PB disponen de un bit para informar al controlador de prebúsqueda del primer acceso a su contenido y poder comenzar una nueva prebúsqueda. Es decir, cuando se produce el primer acierto en alguna de estas tres estructuras, el controlador de prebúsqueda solicita la siguiente línea de memoria a la que supuestamente se accederá. Para poder realizar esta operación, suponemos que también existe un puerto dedicado para que, antes de solicitar la línea al siguiente nivel de la jerarquía de memoria, el controlador verifique que la línea no está en la *iCache*. Sólo en el caso de que el acceso a esta línea vaya a generar un fallo de cache, es cuando se pide realmente la línea al siguiente nivel de la jerarquía de memoria. Posteriormente, el PB se actualizará con la línea de memoria solicitada.

La jerarquía de memoria propuesta presenta dos comportamientos particulares:

a) Cuando todas las instrucciones del LB ya han sido procesadas por la CPU, tanto el LB como el PB intercambian sus funciones.

b) Cuando se produce un acierto en la *Lockable iCache*, tanto el LB como el PB invalidan sus contenidos. Esto elimina cualquier potencial dependencia del camino previamente seguido por el programa durante su ejecución y hace más predecible su comportamiento.

Finalmente, conviene indicar que el sistema no dispone de otros recursos con latencia variable, como por ejemplo un predictor de saltos o una cache de datos.

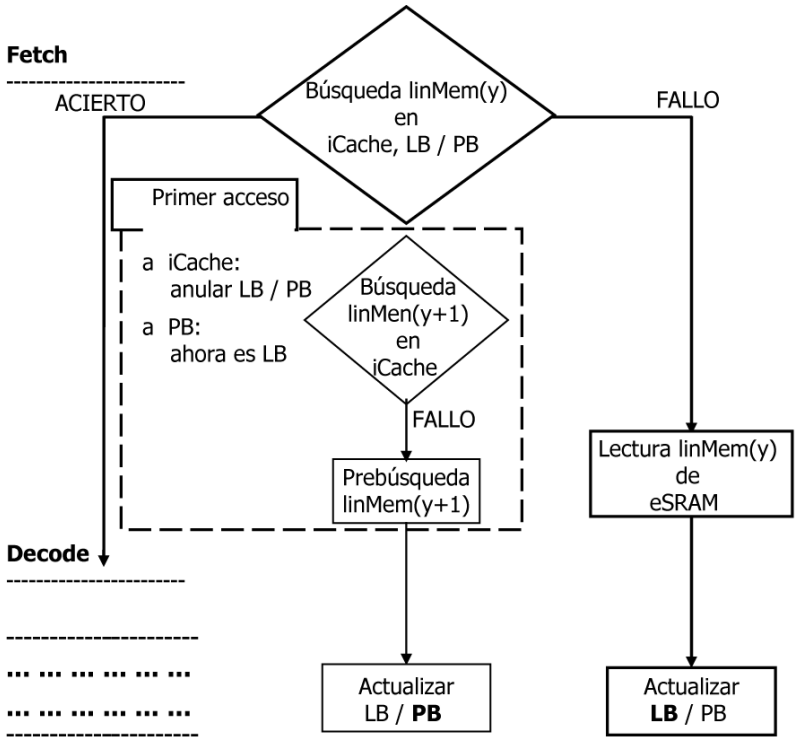


Figura 5.2: Operaciones de la prebúsqueda en la etapa de *fetch* cuando la *iCache* dispone de un puerto dual.

El procesador no está segmentado y la ejecución de las instrucciones se realiza en orden. Por lo tanto el sistema no presenta ningún tipo de *anomalía de distribución* (*timing anomalies*) [115, 155, 193, 43]. Además, suponemos conocido el número máximo de iteraciones de los bucles de cada tarea del sistema y los caminos imposibles se han eliminado.

5.2. Extensión de *Lock-MS* con prebúsqueda

En esta sección se extiende el algoritmo *Lock-MS* (*Lock for Maximize Schedulability*) para optimizar la nueva jerarquía de memoria con prebúsqueda (ver Figura 5.1). Este algoritmo basado en ILP selecciona las líneas de memoria de cada tarea que se bloquean en la cache para que la *planificabilidad* del sistema sea máxima. Se considera un sistema de tiempo real multitarea donde la prioridad de cada tarea es fija y se permiten las expulsiones. La planificación de las tareas se obtiene mediante RMA (*Rate Monotonic Analysis*).

La *Lockable iCache* bloquea su contenido durante algunos periodos de la ejecución del sistema, en particular durante la ejecución de cada tarea. En cada cambio de contexto, el contenido de la cache se actualiza con las líneas de memoria de la tarea que se va a ejecutar. El comportamiento de esta cache es totalmente predecible, y además la prebúsqueda en ningún caso poluciona la cache. Cada tarea, cuando se ejecuta, puede aprovechar toda la cache de forma exclusiva, pero es necesario tener en cuenta el coste de cargar las líneas de memoria de la tarea en cada cambio de contexto.

En esta nueva jerarquía de memoria, el coste de ejecución de cada bloque básico en un camino concreto $Path_i$ sigue siendo constante durante toda la vida del sistema, y su valor se calcula teniendo en cuenta el tiempo de ejecución de cada una de las instrucciones que contiene y el coste de cada uno de los accesos a dicho bloque. Esto permite definir explícitamente el tiempo de ejecución de todos los caminos. Además, como consecuencia de disponer de una cache que fija su contenido durante la ejecución de las tareas, cada uno de los posibles caminos de ejecución es totalmente independiente de los demás. Así pues, el *camino más largo* en ningún caso vendrá determinado por una combinación de diferentes ramas en un mismo condicional.

El objetivo de la extensión del algoritmo *Lock-MS* sigue siendo determinar las líneas de memoria de cada tarea, que se cargarán y bloquearán en la cache antes de su ejecución. El algoritmo tiene que tener en cuenta el WCET de cada tarea y el coste de cargar las líneas seleccionadas en la cache durante cada cambio de contexto. Para obtener la selección de las líneas a fijar, el algoritmo *Lock-MS* modela el problema mediante un conjunto de restricciones lineales basadas en ILP, cuya función objetivo será minimizar el WCET de cada tarea del sistema.

Problemas de contención debido a la prebúsqueda

Al considerar el funcionamiento del PB en la jerarquía de memoria propuesta (ver Figura 5.1), aparece un problema de contención durante los accesos a memoria, ya que todos los accesos a datos, siempre que haya una prebúsqueda en curso, deben esperar su finalización. Mientras se realiza una prebúsqueda, y hasta que la línea solicitada no se guarde en el PB, no es posible solicitar un nuevo dato. Por lo tanto, las instrucciones *load* y *store* pueden sufrir algún tipo de contención o retardo adicional.

Para evitar esta contención durante el acceso a datos, se propone utilizar una arquitectura tipo *Harvard* que separa la memoria de instrucciones y la memoria de datos. También se puede utilizar una *eSRAM* con puerto dual, pero el consumo de energía de cada acceso realizado es mayor. En la Figura 5.3 se muestra esta nueva organización de la jerarquía de memoria con algunos detalles más específicos relacionados con la prebúsqueda.

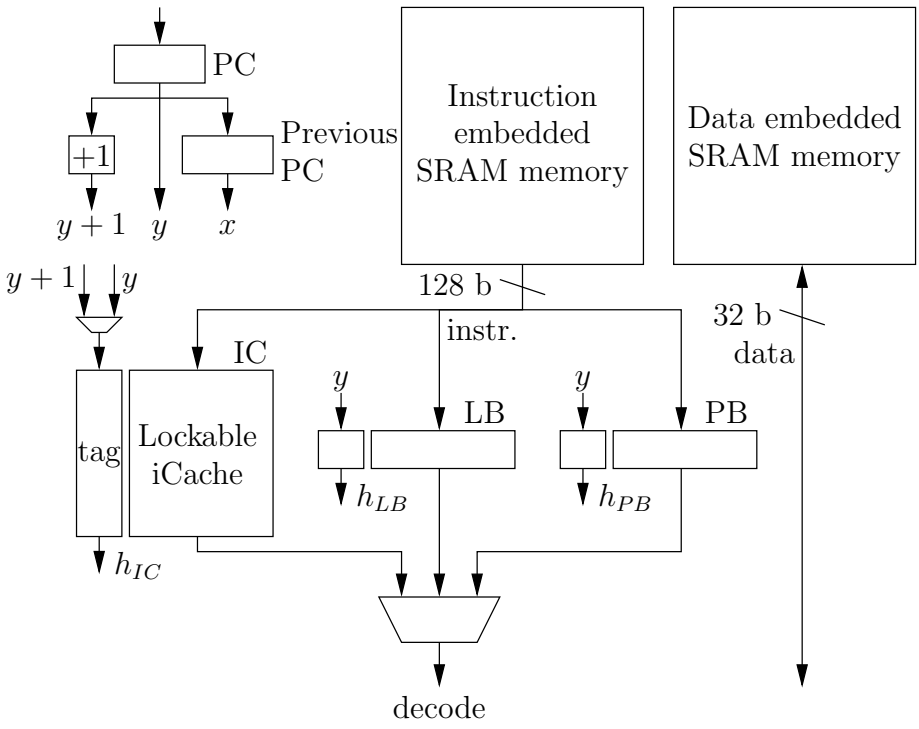


Figura 5.3: Detalle de una jerarquía de memoria para sistemas de tiempo real con prebúsqueda.

Como ya se ha comentado anteriormente, después del primer acierto en cualquiera de las tres estructuras que forman la jerarquía de memoria, *Lockable iCache*, LB o PB, se lanza una nueva prebúsqueda a la *IeSRAM*. Para evitar la prebúsqueda de líneas de memoria que ya están bloqueadas en la cache, tan pronto como sea posible se realiza una simple búsqueda de la siguiente línea de memoria en la *Lockable iCache*. Sólo en el caso de que se vaya a producir un fallo, se lanza la prebúsqueda y se solicita la línea a la *IeSRAM*. Posteriormente esta línea de memoria se guardará en el PB.

En la Figura 5.4 se describe la etapa de *fetch* y *decode* de una instrucción cuando la cache no dispone de un puerto dual. Si se produce un acierto de cache, obviamente durante la etapa de *fetch* el controlador de prebúsqueda no puede usar el puerto de la *Lockable iCache*, ya que se está sirviendo la instrucción para su decodificación. No obstante, cuanto antes se inicie la prebúsqueda, mayor rendimiento se consigue. Así pues, el controlador iniciará la prebúsqueda en el ciclo siguiente.

Cuando se produce un fallo de cache, y en el primer acceso a una línea de

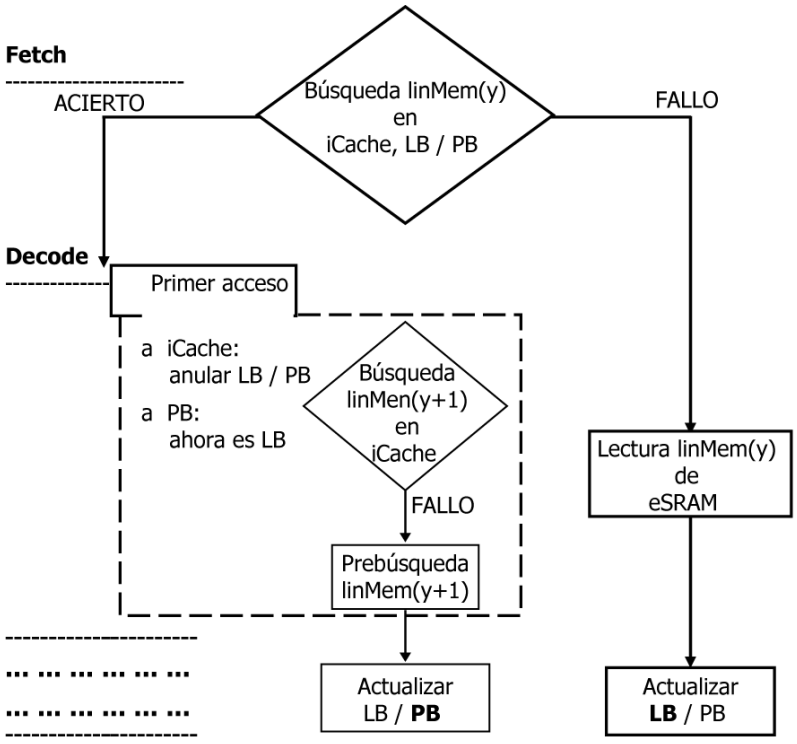


Figura 5.4: Operaciones de la prebúsqueda en las etapas de *fetch* y *decode* cuando la *iCache* sólo tiene un puerto.

memoria $memLine_y$ se consigue un acierto de LB, se realizará el *fetch* de la instrucción de la línea $memLine_y$ para su ejecución. En el ciclo siguiente se buscará la línea $menLine_{y+1}$ en la cache, con el fin de ver si es necesario efectuar una prebúsqueda. Si la línea $menLine_{y+1}$ no está en la cache, se lanza la prebúsqueda que posteriormente actualiza el PB con dicha línea. Como el LB y el PB intercambian sus funciones, si se produce un acierto de PB, pasará a comportarse como el LB y el proceso de prebúsqueda será totalmente equivalente al descrito. Finalmente, si se da un fallo en las tres estructuras de la jerarquía de memoria, es decir, tanto en la *Lockable iCache*, como en el LB y también en el PB, se solicita la línea $menLine_y$ directamente a la *IeSRAM* y posteriormente se guarda dicha línea en el LB.

Así pues, con la jerarquía de memoria propuesta en la Figura 5.3, el único caso de contención aparece si durante la prebúsqueda se ejecuta un salto tomado o un salto obligatorio. En ese instante, si el hardware reconoce que la línea solicitada no será utilizada, se puede iniciar la solicitud de la nueva línea requerida. Pero para considerar el peor caso, suponemos que esta posibilidad no existe, es decir, la memoria no puede abortar las solicitudes de prebúsqueda pendientes y

debe esperar a que éstas finalicen, por lo tanto, en estos casos se producirá una penalización que se debe tener en cuenta.

La penalización por contención, asociada a una línea de memoria $L_{i,j,k}$ que es destino de salto (obligatorio o tomado), se obtiene de una forma equivalente al coste de prebúsqueda, cuando en caso de acierto hay que esperar a que el PB se actualice con la línea de memoria solicitada. Es decir, en todos los casos es necesario esperar a que finalice la solicitud de prebúsqueda pendiente. Por lo tanto, para determinar la penalización por contención, se calculará el tiempo que falta para que finalice la prebúsqueda en curso y se pueda solicitar la nueva línea de memoria $L_{i,j,k}$ a la que se accederá después de la ejecución del salto. En concreto, la posible penalización por contención *contentionPenalty* depende del coste de un fallo de PB $tmissPB$ y del tiempo transcurrido *contentionCost* desde que comenzó la prebúsqueda hasta ese instante de la ejecución.

Por lo tanto, la penalización debido a la contención que se produce en el primer acceso a la línea $L_{i,j,k}$ cuando se está ejecutando la línea de memoria $L_{i,j,m}$, que ha solicitado la prebúsqueda de la línea $L_{i,j,m+1}$ de forma errónea, se representa mediante estas dos nuevas restricciones ILP:

$$\begin{aligned} contentionPenalty_{i,j,k} &\geq tmissPB - contentionCost_{i,j,m} \\ contentionPenalty_{i,j,k} &\geq 0 \end{aligned} \tag{5.1}$$

Esta penalización de contención, causada por una prebúsqueda que no se utiliza durante la ejecución de una línea $L_{i,j,m}$ que contiene un salto obligatorio o tomado, se debe extender a cualquier línea de memoria $L_{i,j,k}$ que contenga una instrucción destino de salto. No obstante, cuando cualquiera de las líneas de memoria implicadas, $L_{i,j,k}$ o $L_{i,j,m+1}$, están bloqueadas en la cache, la penalización es 0, ya que en estos casos no es necesario comenzar la prebúsqueda de la línea $L_{i,j,m+1}$. Sin embargo, cuando la línea de memoria $L_{i,j,m+1}$ está en cache pero la línea que es destino de salto $L_{i,j,k}$ no está en cache, se produce un fallo de PB, que ya está contemplado en las ecuaciones que modelan el problema ILP.

Formulación del problema

La función objetivo para esta nueva jerarquía de memoria, sigue siendo la Ecuación 4.11. Para cada tarea $Task_i$ del sistema se debe tener en cuenta el coste completo de ejecutar la tarea, junto con el coste de todos los cambios de contexto que puede sufrir durante su ejecución.

$$Wcost_i = wcet_i + ncSwitch_i \cdot switchCost_i$$

Pero al considerar la prebúsqueda, algunas ecuaciones se deben actualizar para tener en cuenta el comportamiento del PB. Por ejemplo, el coste de los

cambios de contexto de una tarea $Task_i$, además de considerar el coste de cargar el contenido de la cache $ICpreloadCost_i$ de cada tarea con las líneas de memoria seleccionadas y el coste $LBpreloadCost$ de actualizar el LB, también debe tener en cuenta el coste $PBpreloadCost$ de actualizar el PB, ya que su contenido será invalidado durante el cambio de contexto.

Así pues, teniendo en cuenta la constante $Mlines$ que representa el número de líneas de memoria física del sistema, se puede calcular el coste del cambio de contexto $switchCost_i$ de la tarea $Task_i$ mediante las siguientes ecuaciones:

$$\begin{aligned} switchCost_i &= tSwitch + ICpreloadCost_i + \\ &\quad LBpreloadCost + PBpreloadCost \end{aligned}$$

$$\begin{aligned} PBpreloadCost &= tmiss_{PB} - thit_{PB} \\ LBpreloadCost &= tmiss_{LB} - thit_{LB} \\ ICpreloadCost_i &= (tmiss_{CM} - thit_{CM}) \cdot numcached_i \\ numcached_i &= \sum_{l=0}^{Mlines-1} cached_l \end{aligned} \tag{5.2}$$

Cuando se añade el PB también es necesario actualizar las ecuaciones que describen el coste de ejecución de una línea de memoria, para tener en cuenta el efecto de la prebúsqueda. Así pues, la siguiente ecuación sustituye a la Ecuación 4.7 que describía el coste de ejecución de una línea de memoria cuando se produce un fallo de cache:

$$\begin{aligned} ICmissCost_{i,j,k} &= PBCost_{i,j,k} \cdot nICmissPBhit_{i,j,k} + \\ &\quad + LBCost_{i,j,k} \cdot nICmissPBmiss_{i,j,k} \end{aligned} \tag{5.3}$$

El primer sumando hace referencia al coste de un acierto de prebúsqueda y se ha representado con la variable $PBCost$. Para cada línea de memoria, la ecuación que permite calcular esta variable es la siguiente:

$$\begin{aligned} PBCost_{i,j,k} &= texec_{i,j,k} + PBPenalty_{i,j,k} + \\ &\quad thit_{LB} \cdot (nIns_{i,j,k} - 1) \end{aligned} \tag{5.4}$$

El segundo sumando hace referencia al coste de un fallo de prebúsqueda. En este caso, la restricción del coste de ejecución es la misma que cuando se produce un fallo de cache $ICmissCost$ y la arquitectura de memoria no dispone de prebúsqueda. Es decir, el primer acceso a la línea de memoria será un fallo de LB y el resto serán aciertos de LB, ya que se habrá actualizado con la línea de memoria solicitada al nivel superior de la jerarquía de memoria. No obstante,

para diferenciar ambos casos, se ha definido la constante $LBCost$.

Por lo tanto, cuando se produce un fallo de prebúsqueda, se calcula el coste de ejecución de la línea de memoria mediante la siguiente ecuación:

$$LBCost_{i,j,k} = texec_{i,j,k} + tmiss_{LB} + thit_{LB} \cdot (nIns_{i,j,k} - 1) \quad (5.5)$$

De forma análoga, cuando se produce un fallo de cache, el número de accesos a cada línea de memoria se divide en dos casos, los accesos realizados en secuencia, que son aciertos de prebúsqueda, y los accesos que se producen como destino de salto, que son fallos de prebúsqueda. Para diferenciar estos dos tipos de accesos, se utilizan las constantes $nfetchInSequence$ y $nfetchAfterJump$ respectivamente. Obviamente, estas constantes dependen del número máximo de accesos a cada línea de memoria $nfetch$ durante la ejecución de un camino particular.

La siguiente ecuación muestra la relación entre cada uno de los tipos de accesos posibles:

$$nfetch_{i,j,k} = nfetchInSequence_{i,j,k} + nfetchAfterJump_{i,j,k} \quad (5.6)$$

Pero como ya se ha comentado anteriormente, incluso si se produce un acierto en la prebúsqueda, puede aparecer cierta penalización que se ha de tener en cuenta. Esta nueva penalización $PBPenalty_k$ se da cuando el coste de ejecución de la línea $L_{i,j,k}$, considerando todos los accesos a las instrucciones de la línea como aciertos $hitCost_{i,j,k}$, es menor que el tiempo que cuesta actualizar el PB $tmiss_{PB}$ con la línea de memoria prebuscada. Esta penalización será 0 cuando se pueda ocultar totalmente el tiempo de prebúsqueda, es decir, la ejecución de la línea termina después de que el PB se haya actualizado con la línea solicitada. La penalización máxima que se puede producir será el tiempo de traer una línea del nivel superior en la jerarquía de memoria, en este caso será el tiempo de acceso a la *IeSRAM* menos el coste de ejecutar la línea de memoria a la que se está accediendo en ese instante.

Así pues, la penalización que se puede sufrir cuando se produce un acierto en la prebúsqueda es la siguiente:

$$PBPenalty_{i,j,k} = \text{máx}\,(0,\ miss_{PB} - hitCost_{i,j,k-1})$$

Esta penalización da lugar a dos nuevas restriccciones en el problema ILP:

$$\begin{aligned} PBPenalty_{i,j,k} &\geq tmiss_{PB} - hitCost_{i,j,k-1} \\ PBPenalty_{i,j,k} &\geq 0 \end{aligned} \quad (5.7)$$

En cualquier caso, una vez que la línea de memoria ya se ha almacenado en el PB, cuando el procesador solicita por primera vez una instrucción de dicha línea, el PB pasa a realizar las funciones del LB, y el LB se convierte en el nuevo PB. Por lo tanto, todos los accesos a dicha línea de memoria tienen un coste de $thitLB$.

Para finalizar la descripción del modelado ILP del problema, suponemos que un acierto de cache $thit_{CM}$ tiene el mismo coste que un acierto de LB $thit_{LB}$. Esta suposición es habitual y simplifica la notación en las restricciones que describen el problema, ya que así no es necesario especificar si se está accediendo a una línea de memoria en la cache o en el LB, aunque para establecer esta distinción sólo se tendría que añadir una nueva restricción al problema.

A continuación se describen el conjunto de restricciones asociadas a una línea de memoria $L_{i,j,k}$ que el algoritmo *Lock-MS* necesita para decidir si dicha línea de memoria se bloqueará en la cache[1]:

$$\begin{aligned}
lineCost_{i,j,k} &= IChitCost_{i,j,k} \cdot nIChit_{i,j,k} + \\
&\quad \mathbf{PBCost_{i,j,k}} \cdot \mathbf{nICmissPBhit_{i,j,k}} + \\
&\quad \mathbf{LBCost_{i,j,k}} \cdot \mathbf{nICmissPBmiss_{i,j,k}} \\
\\
IChitCost_{i,j,k} &= texec_{i,j,k} + thit_{CM} \cdot nIns_{i,j,k} \\
\mathbf{PBCost_{i,j,k}} &= \mathbf{texec_{i,j,k}} + \mathbf{PBPenalty_{i,j,k}} + \\
&\quad \mathbf{thit_{LB}} \cdot (\mathbf{nIns_{i,j,k}} - \mathbf{1}) \\
\\
\mathbf{PBPenalty_{i,j,k}} &\geq \mathbf{tmissPB} - \mathbf{IChitCost_{i,j,k-1}} \\
\mathbf{PBPenalty_{i,j,k}} &\geq \mathbf{0} \\
\\
nIChit_{i,j,k} &= nfetch_{i,j,k} \cdot cached_l \\
\mathbf{nICmissPBhit_{i,j,k}} &= \mathbf{nfetchInSequence_{i,j,k}} \cdot (\mathbf{1} - \mathbf{cached_l}) \\
\mathbf{nICmissPBmiss_{i,j,k}} &= \mathbf{nfetchAfterJump_{i,j,k}} \cdot (\mathbf{1} - \mathbf{cached_l}) \\
\mathbf{nfetch_{i,j,k}} &= \mathbf{nfetchInSequence_{i,j,k}} + \mathbf{nfetchAfterJump_{i,j,k}}
\end{aligned}$$

Además, en nuestro caso, donde la arquitectura de memoria no está exenta de las posibles contenciones causadas por las prebúsquedas erróneas, tanto en los saltos obligatorios, como en los saltos tomados, la ecuación para obtener el valor de la variable $LBCost_{i,j,k}$ se reemplaza por las siguientes restricciones:

$$\begin{aligned}
\mathbf{LBCost_{i,j,k}} &= \mathbf{texec_{i,j,k}} + \mathbf{contentionPenalty_{i,j,k}} + \\
&\quad \mathbf{tmiss_{LB}} + \mathbf{thit_{LB}} \cdot (\mathbf{nIns_{i,j,k}} - \mathbf{1})
\end{aligned}$$

[1]Las nuevas restricciones del problema ILP para tener en cuenta el comportamiento del PB se han marcado en negrita.

$$\begin{aligned} \mathbf{contentionPenalty_{i,j,k}} &\geq \mathbf{tmissPB - contentionCost_{i,j,m}} \\ \mathbf{contentionPenalty_{i,j,k}} &\geq \mathbf{0} \end{aligned}$$

Restricciones asociadas a la información de control

Al igual que en el capítulo anterior, el modelado ILP permite añadir otras restricciones, como por ejemplo las limitaciones que se consiguen durante el análisis de flujo de control. Si un bucle contiene varios caminos alternativos, se pueden añadir nuevas restricciones para indicar el número exacto de veces que se ejecutará cada uno de los caminos. La forma de modelar estas nuevas restricciones es totalmente equivalente a las ya utilizadas en otros trabajos [6, 107].

Prebúsqueda: *Modelo explícito* vs. *Modelo compacto*

La base del algoritmo *Lock-MS* es la descripción de todos y cada uno de los posibles caminos de ejecución, como ya se indicó en el Capítulo 4. No obstante, para reducir la complejidad del problema, también se describió un *modelo compacto* que, en principio, no se puede aplicar directamente a una jerarquía de memoria con prebúsqueda, ya que depende de la historia de ejecución.

En este apartado, para ilustrar cómo se determinan en general los costes de prebúsqueda en el *modelo compacto*, se considera el ejemplo de la Figura 5.5. Este ejemplo, ya utilizado en el Capítulo 4, es bastante representativo, puesto que cada bloque básico y sus líneas de memoria abarcan todos los casos de acierto de prebúsqueda que pueden aparecer en un determinado código. Obviamente, suponemos que ninguna de las líneas de memoria del ejemplo está bloqueada en la cache, ya que en estos casos cada acceso se considera como un acierto de cache.

Si la jerarquía de memoria no dispone de prebúsqueda, el coste del primer acceso a cada una de las líneas de memoria será $miss_{LB}$, es decir, se debe considerar un fallo de LB. Pero si la jerarquía de memoria está formada por un LB y un PB, como en la Figura 5.1, cuando se produce un acceso a la siguiente línea de memoria en secuencia, el coste del primer acceso a dicha línea de memoria dependerá de la efectividad de la prebúsqueda, no obstante, se debe tener en cuenta la posible penalización de prebúsqueda $PBPenalty$. En este caso, el primer acceso se debe tratar como un acierto de PB. Pero si se produce un acceso después de un salto obligatorio o de un salto tomado, el coste del primer acceso a la línea de memoria será $miss_{LB}$, es decir, se debe tratar como un fallo de prebúsqueda.

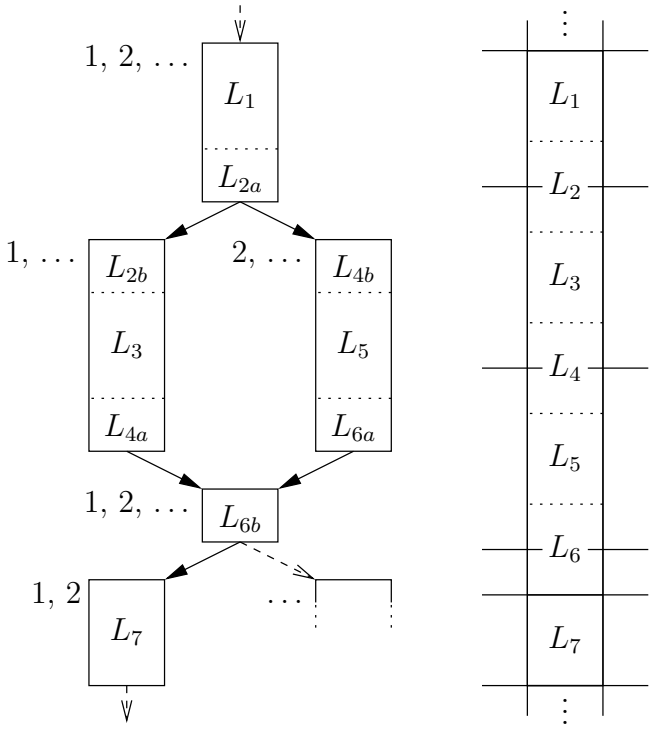


Figura 5.5: Descripción gráfica del *modelo explícito* y del *compacto*.

En la columna etiquetada como *Modelo explícito* de la Tabla 5.1 se resume el coste del primer acceso a cada una de las líneas de memoria asociadas a los caminos $Path_1$ y $Path_2$ de la Figura 5.5. Por claridad en la notación, con superíndices aparece indicada la línea anterior cuyo coste de ejecución determina la posible penalización por la prebúsqueda de dicha línea, representada por la variable $PBPenalty$. Desafortunadamente la transformación al *modelo compacto* no es directa, llegando en algún caso a tener que sobrestimar el coste del primer acceso a la línea de memoria. En la columna etiquetada como *Modelo compacto* de la Tabla 5.1 se resume el coste del primer acceso a cada una de las líneas de memoria asociadas a los caminos $Path_1$ y $Path_2$ de la Figura 5.5.

El coste del primer acceso a la línea L_{2a}, que es secuencial, dependerá del coste de ejecución de la línea anterior, es decir, de la línea L_1. En este caso concreto el coste del primer acceso viene determinado por la constante $PBPenalty^{L_1} = \text{máx}\,(0,\ miss_{PB} - hitCost_{L_1})$, donde la constante $hitCost_{L_1}$ representa el coste de ejecución de la línea de memoria L_1, considerando todos los accesos como aciertos. El coste del primer acceso a las líneas L_3, L_{4a}, L_5 y L_{6a} se calcula de forma equivalente. En todos estos casos, no se produce sobrestimación al asociar directamente estos costes a cada una de las líneas en el *modelo compacto*.

Líneas	Modelo explícito		Modelo compacto
	Path 1	*Path* 2	
L_1	$miss_{LB}$	$miss_{LB}$	$miss_{LB}$
L_{2a}	$PBPenalty^{L_1}$	$PBPenalty^{L_1}$	$PBPenalty^{L_1}$
L_{2b}	0	-	0
L_3	$PBPenalty^{L_2}$	-	$PBPenalty^{L_2}$
L_{4a}	$PBPenalty^{L_3}$	-	$PBPenalty^{L_3}$
L_{4b}	-	$miss_{LB}$	$miss_{LB}$
L_5	-	$PBPenalty^{L_{4b}}$	$PBPenalty^{L_{4b}}$
L_{6a}	-	$PBPenalty^{L_5}$	$PBPenalty^{L_5} - miss_{LB}$
L_{6b}	$miss_{LB}$	0	$miss_{LB}$
L_7	$PBPenalty^{L_{6b}}$	$PBPenalty^{L_6}$	$PBPenalty^{L_{6b}}$

Tabla 5.1: Coste del primer acceso a las líneas de memoria de la Figura 5.5 en la jerarquía de memoria propuesta.

El coste del primer acceso a la línea L_{4b} se ha de tratar como un fallo de PB, ya que esta línea pertenece en exclusividad al camino $Path_2$, y sólo se puede acceder a ella después de un salto condicional cuando es tomado. En este caso la prebúsqueda fallará y el coste del primer acceso será $miss_{LB}$. No obstante, en este caso tampoco se produce sobrestimación al asignar directamente el coste de la línea L_{4b} al *modelo compacto*.

Pero el primer acceso a la línea L_{6b} representa un caso en el que se produce una sobrestimación al trasladar dicho coste al *modelo compacto*. En este caso, se puede acceder a esta línea de dos formas distintas. Si se recorre el camino $Path_1$ es destino de salto obligatorio, por lo tanto, al igual que en el ejemplo anterior, la prebúsqueda falla y el coste del primer acceso es $miss_{LB}$. Pero si se recorre el camino $Path_2$, la línea de memoria L_{6b} ya está en el PB, por lo tanto el coste del primer acceso es 0. Si en el *modelo compacto* asignamos el valor $miss_{LB}$ al coste del primer acceso a la línea L_{6b}, cuando se calcule el coste del camino $Path_2$ se producirá una sobrestimación, puesto que dicha línea ya está en el LB. Obviamente, esta sobrestimación sólo afecta al camino $Path_2$ y para evitarla se puede realizar un ajuste actualizando el coste del primer acceso a la línea L_{6a}, que pertenece en exclusividad al camino $Path_2$. El ajuste a realizar resta dicho valor al coste del primer acceso de la línea L_{6a}, como ya se ha indicado en la Tabla 5.1. Es decir, para evitar la sobrestimación en el primer acceso a la línea L_{6b} en el *modelo compacto*, se realiza un ajuste en el coste del primer acceso a la línea L_{6a} que viene condicionado por la siguiente expresión: $PBPenalty^{L_5} - miss_{PB}$.

Desafortunadamente, no se puede realizar un ajuste para evitar la sobrestimación que se produce en el primer acceso a la línea L_7. Por un lado, en el camino $Path_1$ el coste de ejecución de la línea anterior se corresponde con

$hitCost_{L_{6b}}$, mientras que por otro, en el camino $Path_2$ se debe considerar el coste de ejecución de la línea L_{6a}, y el de la L_{6b}, es decir, se debe tener en cuenta el coste de ejecución de la línea L_6 completa. En este caso, en el *modelo compacto* no se puede evitar la sobrestimación en el primer acceso a la línea L_7, ya que forma parte de ambos caminos.

Con la jerarquía de memoria propuesta en este capítulo, también podría ser interesante adoptar una solución intermedia o híbrida entre el *modelo explicito* y el *modelo compacto*. Es decir, se pueden tratar algunos trozos del programa de forma explícita para obtener más detalle, tanto de la información específica del flujo de control, como de la historia de ejecución. De esta forma, los resultados conseguidos en algunos trozos del programa son más precisos, y en el resto se analizaría con el *modelo compacto*. Por ejemplo, los trozos de código en los que la prebúsqueda tiene asociada una gran sobrestimación, se podrían analizar utilizando el *modelo explícito*. El resultado parcial se añadiría al resto del tiempo de ejecución del programa determinado con el *modelo compacto*.

5.3. Evaluación del rendimiento

En la siguiente sección evaluamos el rendimiento de la jerarquía de memoria propuesta (ver Figura 5.3), en un sistema de tiempo real multitarea donde se permiten las expulsiones y las tareas tienen prioridad fija. En los experimentos se estudian las tareas que se muestran en la Tabla 5.2, son las mismas que las utilizadas en el Capítulo 4 y, en este caso, también se han planificado mediante *Rate Monotonic*.

Conjunto	Tarea	WCET con-LB	Periodo	Tamaño
small	jfdctint	10108	23248	1072 B
	crc	109696	329088	536 B
	matmul	542229	2440031	208 B
	integral	716633	3583165	400 B
medium	minver	8522	19601	1360 B
	qurt	10117	30351	752 B
	jfdctint	10108	44475	1072 B
	fft	2886680	15010736	1016 B

Tabla 5.2: Conjunto de tareas: *small* y *medium*.

Los códigos fuente se han compilado con GCC 2.95.2 -O2 y suponemos que el código de cada tarea se cargará en una dirección de memoria física que se corresponda con el conjunto 0 de la cache. El WCET, que aparece indicado en la Tabla 5.2, hace referencia al tiempo de ejecución de peor caso considerando

un sistema con LB, es decir, sin cache ni prebúsqueda, y sin tener en cuenta el coste de los posibles cambios de contexto. Los periodos de cada tarea se han elegido para que la utilización de la CPU en el sistema base sea de $1,2$. El conjunto de tareas *small* y *medium*, y la relación entre los periodos de cada tarea, ya se han considerado en trabajos anteriores [6, 147].

Al igual que en el Capítulo 4, la arquitectura que analizamos en los experimentos es un ARM v7 con instrucciones de 4 bytes. El tamaño de una línea de memoria es de 16 bytes, tanto en la *Lockable iCache*, como en el LB y en el PB; por lo tanto, cada línea puede contener hasta 4 instrucciones. En los experimentos realizados, la capacidad de la cache varía desde 128 bytes hasta 4 KB. La memoria principal que tiene una organización tipo *Harvard* es de tamaño fijo. En este caso, tanto la memoria de datos *DeSRAM*, como la memoria de instrucciones *IeSRAM* tienen 256 KB.

Para determinar el coste de los accesos a memoria se ha utilizado Cacti V.6.0, una herramienta que permite modelar los circuitos de una jerarquía de memoria [138]. También suponemos que esta jerarquía de memoria formará parte de un procesador de alto rendimiento para sistemas empotrados, construido bajo la tecnología 32 nm, cuyo ciclo de procesador podría ser equivalente a 36 FO4. Un procesador de 36 FO4 en la tecnología 32 nm tendría aproximadamente una frecuencia de reloj de unos $2,4$ GHz, que está en consonancia con la tendencia del mercado actual [1]. También se ha verificado que todas las caches analizadas, excepto las totalmente asociativas, pueden proporcionar una instrucción en un ciclo de procesador. El tiempo mínimo de acceso a una *eSRAM* de 256 KB, fabricada con transistores de bajo consumo en reserva, podría ser de unos 7 ciclos.

Así pues, en los experimentos realizados hemos supuesto que el coste de un acierto de cache o de LB es de 1 ciclo, mientras que el coste de un fallo es de 7 ciclos; con esta latencia de memoria se consigue estresar el funcionamiento de la *Lockable iCache*. Por otro lado, en caso de producirse un acierto de prebúsqueda, su coste estará comprendido entre 1 y 7 ciclos, en función del instante en el que se realizó la solicitud de prebúsqueda y del tiempo transcurrido. También conviene indicar que todos los accesos a datos se efectúan directamente a la *DeSRAM*. Finalmente, los costes de ejecución se han modelado suponiendo que una instrucción predicada, que no se ejecuta debido a la condición, consume 1 ciclo; y que las instrucciones que se ejecutan, y no son accesos a memoria, lo hacen en 2 ciclos. Todas las instrucciones de acceso a memoria (*load* y *store*) consumen 1+7 ciclos.

Localidad espacial con prebúsqueda

En este apartado se analiza la influencia de la localidad espacial en el WCET de cada tarea cuando se ejecuta de forma aislada. Los componentes de la jerarquía de memoria que permiten explotar la localidad espacial son el LB y el PB.

No obstante, se analiza el WCET de cada tarea en 4 escenarios distintos:

- En el primer escenario todos los accesos a memoria son fallos, es decir, tanto los accesos a las instrucciones, como a los datos, se realizan directamente a la *IeSRAM* y a la *DeSRAM* respectivamente. Éste es el escenario más pesimista, ya que se asume que ninguno de los componentes de la jerarquía de memoria propuesta está en funcionamiento. Por lo tanto, representa el límite superior en cuanto al coste de los accesos a memoria.
- En el segundo escenario la jerarquía de memoria dispone de LB. Aunque este escenario ya se analizó con todo detalle en el Capítulo 4, conviene volver a presentar los resultados logrados, para realizar una comparación con mayor facilidad.
- En el tercer escenario se tiene en cuenta la prebúsqueda y se añade el PB a la jerarquía de memoria analizada.
- Finalmente, en el cuarto escenario se considera un sistema ideal donde todos los accesos a instrucciones son aciertos. Este caso es el más optimista. Todos los accesos se realizan directamente desde la cache y ni siquiera se tiene en cuenta la penalización por cargar las líneas de memoria en la cache. Este escenario representa el límite inferior en cuanto al coste de los accesos a memoria.

Es importante destacar que ninguno de los escenarios propuestos incluye el estudio de la *Lockable iCache*, ya que estamos analizando la localidad espacial en la jerarquía de memoria propuesta (ver Figura 5.3). Así pues, utilizamos la *Lockable iCache* para capturar la localidad temporal.

En la Figura 5.6 se muestra una comparativa de los cuatro escenarios analizados. Los resultados se han normalizado con respecto al primer escenario, en el que todos los accesos se consideran fallos ya que se realizan directamente a la *eSRAM*. Como se observa en la Figura 5.6, el WCET obtenido, tanto en la jerarquía de memoria que dispone de LB, como en la que dispone de LB y PB, presenta una reducción muy significativa. Esto es debido a que, tanto el LB, como el PB consiguen explotar muy bien la localidad espacial. Por ejemplo, con un simple LB, el WCET se reduce de media en aproximadamente un 47 %. En un sistema con LB y PB, el WCET se reduce de media en un 60 %. Finalmente, en el sistema ideal se reduce el WCET de las tareas analizadas hasta en un 67 %, pero, como ya se ha comentado anteriormente, en este escenario hemos supuesto que todas las instrucciones de la tarea están en la cache, sin tener en cuenta el coste de cargar dichas instrucciones en cache. Por lo tanto, la máxima reducción del WCET que se podría obtener añadiendo, a un sistema que ya dispone de un LB y un PB, una cache de instrucciones para explotar la localidad temporal, sería de aproximadamente un 7 %, asumiendo que el caso ideal se puede conseguir. También conviene indicar que los accesos a datos podrían proporcionar una mejora de aproximadamente el 33 %.

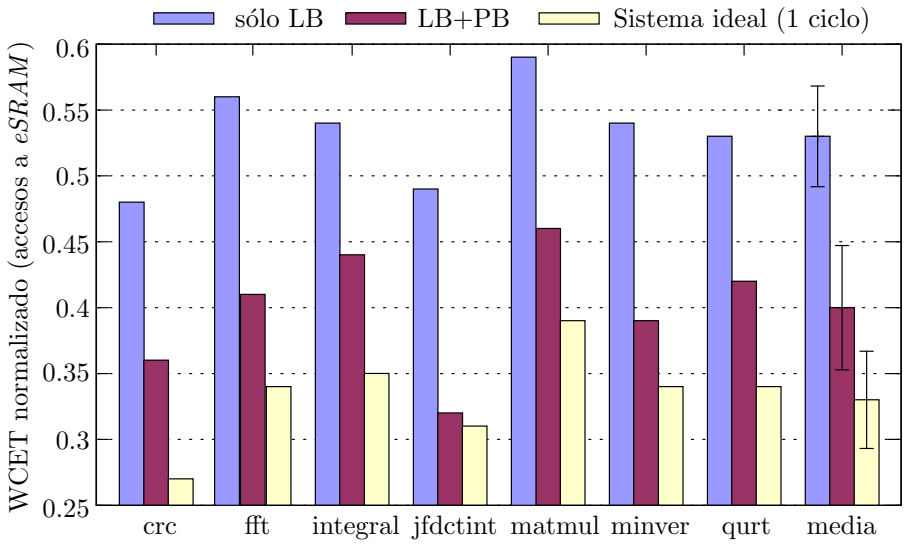


Figura 5.6: WCET de cada tarea en las 4 configuraciones consideradas.

En definitiva, teniendo en cuenta que en el sistema ideal el incremento en el rendimiento alcanza de media un 67 %, las mejoras de rendimiento que se pueden obtener en la jerarquía de memoria propuesta (ver Figura 5.3) son las siguientes:

- Si la jerarquía de memoria sólo dispone de un LB, la mejora en el rendimiento que se alcanza es de un 47 % que representa un incremento relativo del 70 %. En este caso una *Lockable iCache* podría aumentar el rendimiento hasta un 30 % más.
- Si la jerarquía de memoria dispone de un LB y de un PB, la mejora en el rendimiento que se puede alcanzar es de un 60 % que representa un incremento relativo del 90 %. En este caso una *Lockable iCache* podría mejorar el rendimiento sólo hasta un 10 % más.

Utilización de la CPU con prebúsqueda

En un sistema multitarea también se puede calcular la utilización del procesador para medir los efectos de la localidad espacial. La utilización del procesador se define como la fracción de tiempo de la CPU en la que está ocupada ejecutando las tareas del sistema en el peor caso.

$$U = \sum_{i=1}^{NTask} \frac{Wcost_i}{T_i}$$

Una utilización del procesador superior a 1 indica que el sistema no es *planificable*. Pero aún cuando la utilización del procesador es inferior a 1, no está garantizado que el sistema sea *planificable*, ya que esta condición es necesaria, pero no suficiente. No obstante, cuando la utilización del procesador es inferior a 1, la *planificabilidad* del sistema se puede verificar mediante el análisis del tiempo de respuesta (RTA/ *Response Time Analysis*). Mediante RTA se verifica si el tiempo de respuesta R_i de cada tarea $Task_i$ del sistema es menor que su plazo de finalización D_i. Es decir, el sistema es *planificable* si $R_i \leq D_i$ con $1 \leq i \leq NTask$.

Como ya se ha comentado anteriormente, los periodos indicados en la Tabla 5.2 se han ajustado para que en ambos conjuntos de tareas, *small* y *medium*, la utilización del procesador sea de $1,2$. Es decir, para que el sistema no sea *planificable*. Los resultados obtenidos ya indican que en una jerarquía de memoria sin cache de instrucciones, pero que tenga un LB y un PB, la utilización del sistema es inferior a $0,9$, por lo tanto el sistema ya podría ser *planificable*.

Para los conjuntos de tareas *small* y *medium*, en la Figura 5.7 se muestra la utilización del procesador en un sistema donde se accede directamente a la *eSRAM*; en este caso el sistema no es *planificable* en absoluto. Se indica la utilización del procesador en una jerarquía de memoria formada únicamente por un LB donde la utilización alcanza el $1,2$. También se muestra la utilización del procesador en una jerarquía de memoria formada por un LB y un PB; en este caso el sistema ya podría ser *planificable*. Y finalmente, se expone la utilización del procesador en un sistema ideal, donde todos los accesos a memoria son aciertos.

Con respecto al sistema en el que se accede directamente a la *eSRAM*, la reducción de la utilización del procesador es cada vez mayor. Por ejemplo, con un simple LB, la utilización se reduce en casi un 49 %. Cuando se considera un LB y un PB, la utilización del procesador se reduce hasta un 61 %. En el caso ideal la reducción aproximadamente alcanza hasta un 68 %. Como consecuencia de estos resultados, conviene indicar que con una cache de instrucciones la reducción máxima en la utilización del procesador que se podría alcanzar, sería de aproximadamente un 7 %. Por lo tanto, en una jerarquía de memoria formada por un LB junto con un PB, la localidad espacial se captura perfectamente. Así pues, los resultados experimentales confirman que el rendimiento de un sistema de tiempo real aumenta de forma espectacular, con una jerarquía de memoria formada únicamente por un LB y un PB. Además, ambos componentes, y en particular el PB, reducen la influencia de la cache de instrucciones en el WCET de las tareas.

Análisis del tiempo de respuesta con prebúsqueda

En el siguiente apartado se compara el tiempo de respuesta obtenido mediante los algoritmos *Lock-MU* [147] y *Lock-MS* para los conjuntos de tareas *small*

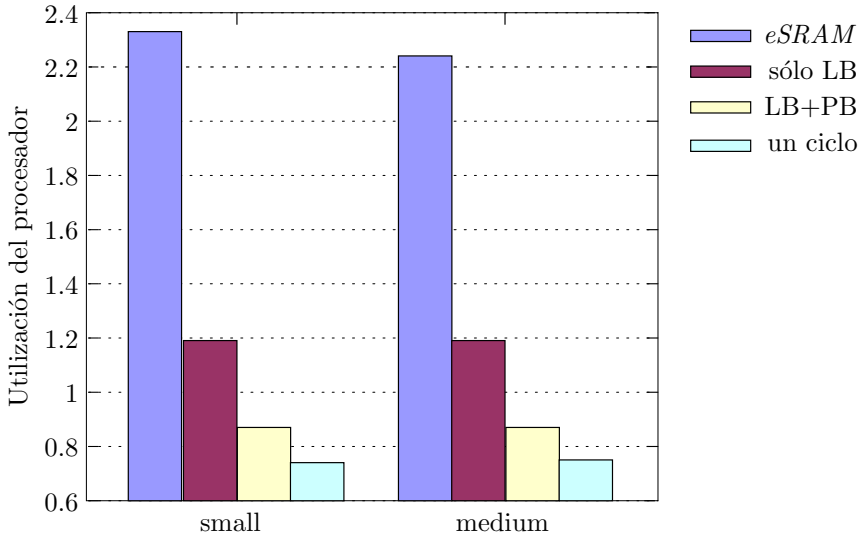


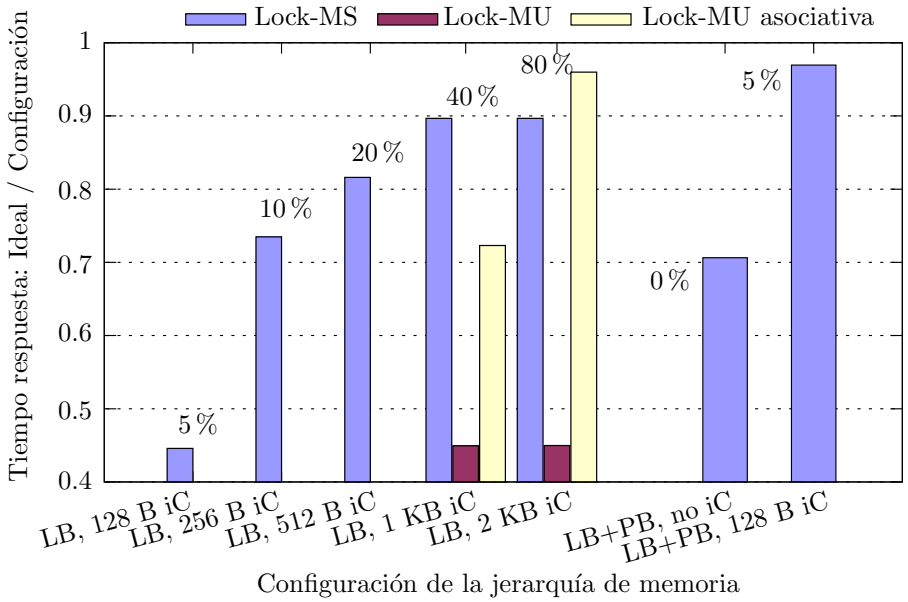
Figura 5.7: Utilización del procesador para los conjuntos de tareas *small* y *medium*.

y *medium*. En concreto comparamos el tiempo de respuesta de la tarea de prioridad más baja conseguido mediante *Lock-MU* y *Lock-MS* en una jerarquía de memoria formada por un LB y una *Lockable iCache*. En la Figura 5.8 se muestra el análisis realizado para una cache de correspondencia directa, no obstante, para el algoritmo *Lock-MU* también se indican los resultados conseguidos en una cache totalmente asociativa, ya que este algoritmo logra el mejor rendimiento para esta configuración de la cache. También se muestran los resultados obtenidos mediante *Lock-MS* en una jerarquía de memoria con prebúsqueda, sin cache, y con una cache de 128 bytes.

El tiempo de respuesta de la tarea de prioridad más baja en los conjuntos *small* y *medium* se ha calculado teniendo en cuenta la selección de líneas de memoria, que los algoritmos *Lock-MU* y *Lock-MS* deciden fijar en la cache. En la parte izquierda de la Figura 5.8 se analiza una jerarquía de memoria formada por un LB y una *Lockable iCache* cuya capacidad varía desde 128 bytes hasta 2 KB para el conjunto *small* y desde 256 bytes hasta 4 KB para el conjunto *medium*. Los resultados que se presentan aparecen normalizados al tiempo de respuesta conseguido en el sistema ideal, donde todos los accesos a las instrucciones se consideran aciertos. En la gráfica junto a cada barra aparece indicado el porcentaje que representa la capacidad de la cache con respecto al tamaño del conjunto de tareas analizado.

Como se observa en la parte izquierda de la Figura 5.8, el algoritmo *Lock-MS*

a) Conjunto de tareas *small*



b) Conjunto de tareas *medium*

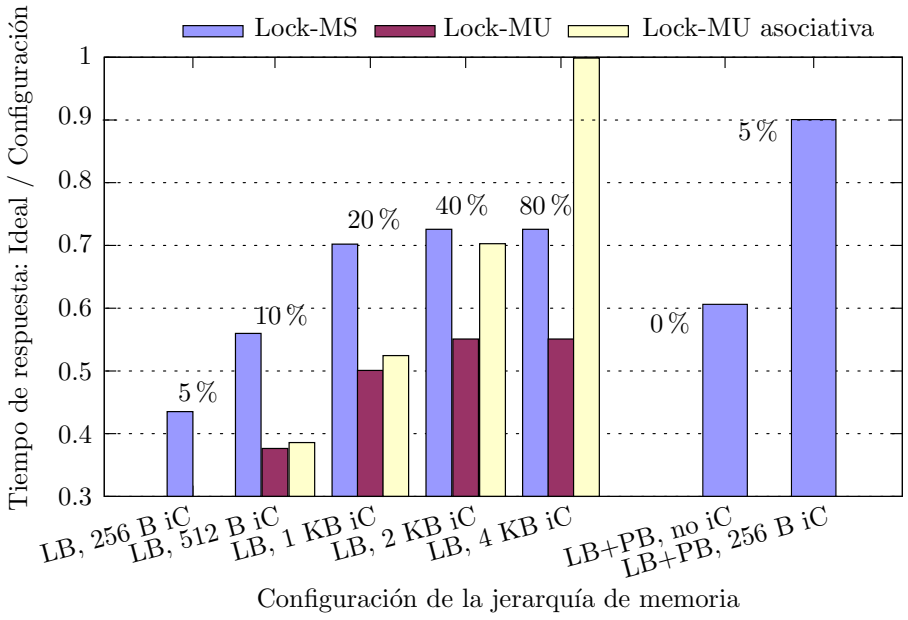


Figura 5.8: Tiempo de respuesta de la tarea de prioridad más baja, normalizado al caso ideal.

consigue mejor rendimiento con caches pequeñas que el algoritmo *Lock-MU*. El algoritmo *Lock-MU* sólo supera en prestaciones al algoritmo *Lock-MS*, cuando la capacidad de la cache es superior al 80 % del tamaño del conjunto de tareas analizado. También se muestra claramente que, con caches grandes totalmente asociativas, el algoritmo *Lock-MU* se aproxima al caso ideal. Finalmente conviene indicar que el algoritmo *Lock-MU* consigue mejores resultados que el algoritmo *Lock-MS* en el conjunto de tareas *medium*, debido al mayor número de cambios de contexto que presenta este conjunto de tareas. En este caso, los cambios de contexto influyen negativamente en el rendimiento del algoritmo *Lock-MS*.

En la parte derecha de la Figura 5.8 se indica el tiempo de respuesta obtenido en una jerarquía de memoria con prebúsqueda. La prebúsqueda aumenta significativamente el rendimiento del sistema. Con una *Lockable iCache* pequeña, de 128 bytes, se supera en prestaciones a un sistema sin prebúsqueda con una cache grande. Por ejemplo, tanto en el conjunto de tareas *small* como en el *medium*, con una cache de capacidad reducida, de tan sólo un 5 % del tamaño del conjunto de tareas analizado, se alcanza un rendimiento equivalente al caso ideal, en el que no se ha tenido en cuenta el coste de cargar los contenidos de la cache. En general, esto es debido a que sólo es necesario cargar en la cache las líneas de memoria que son destino de salto, tanto los obligatorios, como los tomados. En estos casos es cuando la prebúsqueda falla. Obviamente hay otros detalles a tener en cuenta, pero será el *solver* el que decida si una determinada línea de memoria se debe bloquear en la cache o no. Así pues, en presencia de un sencillo hardware de prebúsqueda, el coste de cargar y bloquear las instrucciones en la cache, en cada cambio de contexto, también se reduce considerablemente.

El hardware de prebúsqueda aumenta de forma significativa el rendimiento del sistema en el peor caso y reduce la capacidad de la cache necesaria para alcanzar un rendimiento equivalente al caso ideal. Por lo tanto, un sistema con caches grandes y totalmente asociativas, basado en técnicas de *static locking cache*, puede que no supere en prestaciones a un sistema con caches pequeñas y correspondencia directa basado en técnicas de *dynamic locking cache*.

En definitiva, en un sistema de tiempo real, una jerarquía de memoria con prebúsqueda como la propuesta (ver Figura 5.1) consigue mejor rendimiento con una cache pequeña que una jerarquía de memoria sin prebúsqueda, aunque ésta disponga de una cache grande totalmente asociativa.

Efecto de la penalización en la prebúsqueda

Un hardware de prebúsqueda sencillo como el de la jerarquía de memoria propuesta (ver Figura 5.1) incrementa el rendimiento en el peor caso. Aunque es habitual que, en la mayor parte de los sistemas que disponen de hardware de prebúsqueda, la penalización quede totalmente oculta, en este apartado se

estudia la influencia de una posible penalización en la prebúsqueda en el WCET de una tarea.

En general, en los experimentos realizados, hemos supuesto que la ejecución de una instrucción consume 3 ciclos de media, 1 ciclo para *fetch* y 2 ciclos para su ejecución. Por lo tanto es necesario ejecutar al menos 3 instrucciones para ocultar totalmente la penalización de prebúsqueda, ya que hemos supuesto que acceder a una *eSRAM* tiene un coste mínimo de 7 ciclos. Bajo estas hipótesis, la prebúsqueda es efectiva cuando el tiempo de traer una línea de memoria es menor que el tiempo de ejecutar la línea que solicita la prebúsqueda, pero ese tiempo depende, en particular, de dicha línea.

Para evaluar el efecto de la penalización en la prebúsqueda, hemos estudiado el WCET de un conjunto de programas sintéticos cuyo tamaño varía entre 16 y 96 KB. El número de posibles caminos de ejecución en estos programas sintéticos varía entre 512 y 10^{65} caminos. En la Figura 5.9, donde se plasman los resultados obtenidos, se observa la importancia de la penalización de la prebúsqueda en el WCET. En la gráfica se representa el caso ideal con la etiqueta PB_0, donde la penalización de prebúsqueda es siempre de 0 ciclos. Es decir, la prebúsqueda siempre es correcta y la línea de memoria solicitada se sirve antes de que se demande una de sus instrucciones. Con la etiqueta PB_4 se muestran los resultados obtenidos cuando se fuerza una penalización de 4 ciclos en todas las prebúsquedas, incluso en los casos de acierto. Además mostramos los resultados conseguidos en una jerarquía de memoria que sólo dispone de LB (que sería equivalente al caso $LB + PB_8$) y en una jerarquía donde todos los accesos se realizan directamente a la *eSRAM*. En la Figura 5.9, la media del WCET de los programas estudiados en el caso ideal, se ha normalizado a la media del WCET de los programas para cada una de las jerarquías sin cache analizadas.

Finalmente, conviene indicar que un sistema con un LB y un PB sin penalización de prebúsqueda es equivalente al caso ideal. Mientras que un sistema con un LB y un PB con una penalización de prebúsqueda de 4 ciclos, presenta unos resultados más parecidos a un sistema con sólo un LB. Así pues, la penalización de prebúsqueda, que es un factor clave en el diseño del hardware, se debería evitar para obtener un buen rendimiento en el peor caso.

Coste computacional del *Lock-MS* con prebúsqueda

El coste computacional del algoritmo *Lock-MS* depende de la estructura y del tamaño de las tareas analizadas, así como del *solver* que se utiliza para solucionar el problema ILP, como ya se indicó en el apartado dedicado al análisis de este coste del Capítulo 4. En los experimentos realizados con los conjuntos de tareas, *small* y *medium*, de la Tabla 5.2, la solución del problema ILP se ha obtenido en unos pocos milisegundos y por lo tanto el tiempo de análisis no es

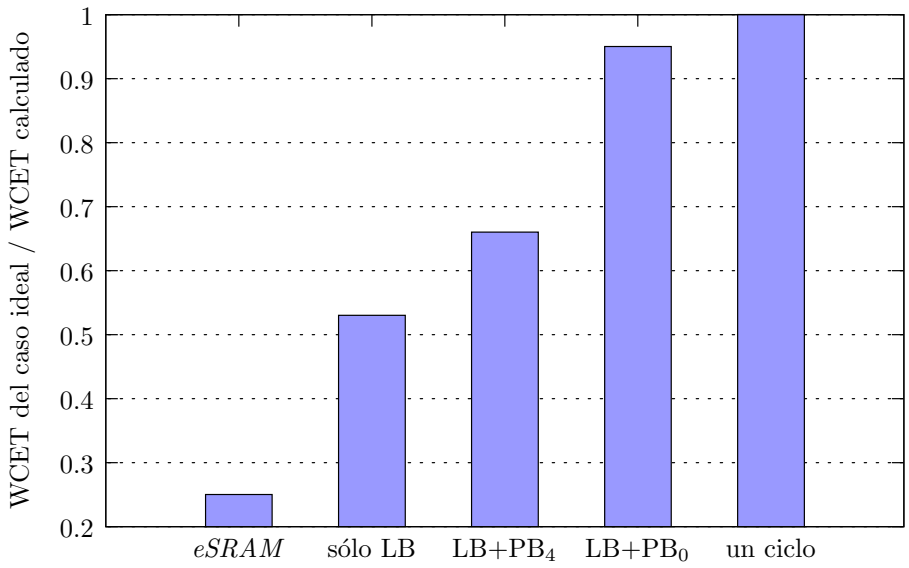


Figura 5.9: Efecto de la penalización de prebúsqueda en el WCET sin caches.

relevante.

En la extensión del algoritmo *Lock-MS* que se ha presentado en este capítulo, la prebúsqueda en una jerarquía de memoria con una cache que puede fijar su contenido, ha mejorado significativamente el rendimiento del sistema. En principio, la prebúsqueda no debería afectar al coste computacional del problema, ya que para describir el comportamiento del PB basta con añadir unas cuantas restricciones ILP al algoritmo *Lock-MS* sin prebúsqueda. Los experimentos realizados nos muestran que la prebúsqueda disminuye el número de líneas a fijar en la cache. Esto significa que es más fácil para el *solver* encontrar una solución óptima y, por lo tanto, el tiempo dedicado por éste a solucionar el problema con las nuevas restricciones de prebúsqueda debería ser equivalente o incluyo inferior al tiempo que le cuesta resolver el problema ILP original.

5.4. Consumo energético

Muchos sistemas empotrados son dispositivos móviles que funcionan con baterías. En estos sistemas el consumo de energía influye decisivamente en su diseño, ya que deben ser eficientes en consumo energético. En esta sección realizamos un estudio sobre el consumo de energía en la jerarquía de memoria propuesta (ver Figura 5.3).

El consumo de energía de una jerarquía de memoria depende de las operaciones que puede realizar. Para calcular su consumo, es necesario describir las operaciones que lleva a cabo, indicando además los circuitos de memoria que intervienen en cada una de ellas. También hay que asignar un consumo de energía a los circuitos de la memoria, en función del ancho del bus usado y del modo de acceso. Así pues, es necesario diferenciar si se accede a una palabra de 32 bits o a una línea de memoria de 128 bits, así como si el acceso es una lectura (Rd) o una escritura (Wr). Finalmente, el número concreto de operaciones que realiza la jerarquía de memoria durante la ejecución de una tarea depende del camino seguido en dicha ejecución.

Consumo de energía de las operaciones de memoria

En la Tabla 5.3 se describen las operaciones que puede realizar la jerarquía de memoria propuesta (ver Figura 5.3), los circuitos de memoria que intervienen en dichas operaciones y, tanto el modo de acceso, como la anchura necesaria del bus de datos para la operación.

Operación	Componentes de la Jerarquía de Memoria				
	IeSRAM	DeSRAM	iC	LB	PB
Acierto en *fetch*	—	—	Rd(32b)	Rd(32b)	Rd(32b)
Fallo en *fetch*	Rd(128b)	—	—	Wr(128b) +Rd(32b)	—
Prebúsqueda	Rd(128b)	—	LookUp	Wr(LB/PB)(128b)	
Leer dato	—	Rd(32b)	—	—	—
Guardar dato	—	Wr(32b)	—	—	—
Cambio contexto					
LB	Rd(128b)	—	—	Wr(128b)	—
PB	Rd(128b)	—	—	—	Wr(128b)
Cambio contexto (por línea)	Rd(128b)	—	Wr(128b)	—	—
Consumo estático	Por fuga				

Tabla 5.3: Descripción de las operaciones y de los componentes de la jerarquía de memoria.

Un ejemplo de estas descripciones es la revisión del consumo de energía cuando se produce un *Fallo en fetch*. Este consumo depende de la energía que consume una lectura de línea de memoria de la *IeSRAM* y de la energía que consume guardar dicha línea en el LB. En la última fila de la Tabla 5.3 aparece reflejado el consumo de energía estática, que corresponde a la fuga de energía, de cada componente de la jerarquía, que se produce ciclo a ciclo mientras el sistema está activo.

El algoritmo *Lock-MS* nos puede proporcionar el número de cada una de las operaciones que realiza la jerarquía de memoria durante la ejecución de una tarea, pero obviamente dependerá del camino concreto que se ejecuta. En particular puede proporcionar esta información para el camino que determina el WCET de una tarea. Así pues, en una primera aproximación proponemos extender el algoritmo *Lock-MS* para que también calcule el consumo energético del *camino más largo* de la tarea analizada.

A continuación, para cada camino $Path_{i,j}$ de una tarea $Task_i$, se definen los parámetros necesarios para que el algoritmo *Lock-MS* determine el consumo energético de dicho camino $pathEnergy_{i,j}$ y en particular el consumo de energía del camino que determina el WCET de la tarea. En primer lugar, representamos mediante las siguientes constantes el consumo de energía de cada circuito de memoria, en función del modo de acceso y del ancho de datos implicado en la operación descrita en la Tabla 5.3:

- $En_{IeSRAM_{Rd(128)}}$: Energía consumida durante la lectura de una línea en la *IeSRAM*.
- $En_{iC_{Rd(32)}}$: Energía consumida durante la lectura de una instrucción en la *Lockable iCache*.
- $En_{LB_{Rd(32)}}$: Consumo de energía durante la lectura de una instrucción en el LB.
- $En_{PB_{Rd(32)}}$: Consumo de energía durante la lectura de una instrucción en el PB.
- $En_{iC_{Wr(128)}}$: Energía consumida durante la escritura de una línea en la *Lockable iCache*.
- $En_{LB/PB_{Wr(128)}}$: Consumo de energía durante la escritura de una línea de memoria en el LB/PB
- $En_{iC_{LookUp}}$: Energía consumida durante la prebúsqueda para verificar si la línea a prebuscar ya está en la *Lockable iCache*.
- $En_{DeSRAM_{Rd(32)}}$: Consumo de energía durante la lectura de un dato en la *DeSRAM*.
- $En_{DeSRAM_{Wr(32)}}$: Energía consumida durante la escritura de un dato en la *DeSRAM*.
- *Leak*: Consumo de energía estática del sistema durante un ciclo.

Consumo de energía durante la ejecución de un camino

Dado el camino $Path_{i,j}$ de la tarea $Task_i$, el consumo de energía de un acierto de *fetch* $Energy_{i,j}^{hitFetch}$ depende del consumo de energía del circuito de memoria que contiene la instrucción solicitada. Es decir, depende de si está en la *iCache* cuyo consumo representamos con $En_{iC_{Rd(32)}}$, en el LB cuyo consumo hemos definido como $En_{LB_{Rd(32)}}$, o en el PB cuyo consumo se representa con $En_{PB_{Rd(32)}}$.

Así pues, el valor de la constante $Energy_{i,j}^{hitFetch}$ se obtiene mediante la siguiente expresión:

$$\begin{aligned} Energy_{i,j}^{hitFetch} &= En_{iC_{Rd(32)}} \cdot \sum_{k=1}^{Nlines_{i,j}} nIChit_{i,j,k} \cdot nIns_{i,j,k} \\ &+ En_{PB_{Rd(32)}} \cdot \sum_{k=1}^{Nlines_{i,j}} nICmissPBhit_{i,j,k} \cdot nIns_{i,j,k} \\ &+ En_{LB_{Rd(32)}} \cdot \sum_{k=1}^{Nlines_{i,j}} nICmissPBmiss_{i,j,k} \cdot nIns_{i,j,k} \end{aligned} \tag{5.8}$$

Durante la ejecución de un camino $Path_{i,j}$, el consumo de energía de prebúsqueda $Energy_{i,j}^{PB}$ asociado a dicho camino depende del número de veces que se realiza la prebúsqueda y de su consumo energético $Energy^{Prefetch}$, que vendrá indicado por el consumo energético de verificar que la línea no está en la *iCache*, cuyo consumo hemos representado por $En_{iC_{LookUp}}$, por la energía consumida durante la lectura de la línea prebuscada en la *IeSRAM*, que hemos definido como $En_{IeSRAM_{Rd(128)}}$, y por el consumo de energía durante la escritura de dicha línea en el LB/PB, que representamos por $En_{LB/PB_{Wr(128)}}$. Conviene indicar que las líneas que ya están en la *iCache* no se solicitan por prebúsqueda.

El valor de la constante $Energy_{i,j}^{PB}$ se calcula mediante las siguientes ecuaciones:

$$\begin{aligned} Energy^{Prefetch} &= En_{iC_{LookUp}} + En_{ISRAM_{Rd(128)}} + En_{LB/PB_{Wr(128)}} \\ Energy_{i,j}^{PB} &= Energy^{Prefetch} \cdot \sum_{k=1}^{Nlines_{i,j}} nICmiss_{i,j,k} \\ &+ En_{iC_{LookUp}} \cdot \sum_{k=1}^{Nlines_{i,j}} nIChit_{i,j,k} \end{aligned} \tag{5.9}$$

Pero si la prebúsqueda falló también hay que tener en cuenta el consumo de energía derivado de esta circunstancia $Energy_{i,j}^{misPB}$, ya que habrá que traer de

la *IeSRAM* la línea de memoria que contiene la instrucción a ejecutar, y guardar dicha línea en el LB/PB. Las constantes $En_{IeSRAM_{Rd(128)}}$ y $En_{LB/PB_{Wr(128)}}$ representan respectivamente el consumo de energía de cada una de las operaciones indicadas.

Así pues, para el camino $Path_{i,j}$, el valor de la constante $Energy_{i,j}^{missPB}$ se obtiene mediante las siguientes ecuaciones:

$$\begin{aligned} Energy^{missPB} &= En_{ISRAM_{Rd(128)}} + En_{LB/PB_{Wr(128)}} \\ Energy_{i,j}^{missPB} &= Energy^{missPB} \cdot \sum_{k=1}^{Nlines_{i,j}} nICmissPBmiss_{i,j,k} \end{aligned} \tag{5.10}$$

El consumo de energía durante el acceso a la *DeSRAM* también se ha de considerar, tanto para leer un dato, como para guardar su valor. El consumo de energía de los accesos a datos para un camino $Path_{i,j}$ se representa mediante las constantes $Energy_{i,j}^{load}$, si se accede a la *DeSRAM* para leer el dato, y $Energy_{i,j}^{store}$ si se accede para actualizar dicho dato.

Por lo tanto, para el camino $Path_{i,j}$ el valor de ambas constantes se determina mediante las siguientes ecuaciones:

$$\begin{aligned} Energy_{i,j}^{load} &= En_{DSRAM_{Rd(32)}} \cdot \sum_{k=1}^{Nlines_{i,j}} loadIns_{i,j,k} \\ Energy_{i,j}^{store} &= En_{DSRAM_{Wd(32)}} \cdot \sum_{k=1}^{Nlines_{i,j}} storeIns_{i,j,k} \end{aligned} \tag{5.11}$$

Conviene indicar que para todos los caminos $Path_{i,j}$ de una tarea $Task_i$, los valores de las constantes $loadIns_{i,j,k}$ y $storeIns_{i,j,k}$, que presentan respectivamente el número de instrucciones *load* y *store* que contiene una línea de memoria $L_{i,j,k}$, son conocidos, ya que el tiempo de ejecución de las instrucciones $texec_{i,j,k}$, que contiene una línea, depende de estos valores.

Finalmente, se ha de tener en cuenta el consumo de energía estática $Energy_{i,j}^{leak}$ que depende del tiempo de ejecución del camino considerado.

Para un camino $Path_{i,j}$ el valor de la constante $Energy_{i,j}^{leak}$ se obtiene mediante el siguiente cálculo:

$$Energy_{i,j}^{leak} = pathCost_{i,j} \cdot Leak \tag{5.12}$$

En definitiva, el consumo de energía $pathEnergy_{i,j}$ de un camino $Path_{i,j}$ depende del consumo de energía de cada una de las operaciones que se realizan

durante su ejecución. Así pues, este consumo se puede calcular sumando todos los consumos energéticos parciales descritos anteriormente.

El consumo de energía durante la ejecución de un camino $Path_{i,j}$ de la $Tarea_i$ lo representamos mediante la siguiente ecuación:

$$\begin{aligned} pathEnergy_{i,j} &= Energy_{i,j}^{hitFetch} + Energy_{i,j}^{PB} + Energy_{i,j}^{misPB} \\ &+ Energy_{i,j}^{load} + Energy_{i,j}^{store} \\ &+ Energy_{i,j}^{leak} \end{aligned} \tag{5.13}$$

En un sistema multitarea también se debe estimar el consumo de energía durante los cambios de contexto, pero en este caso depende de la tarea $Task_i$ que se va a ejecutar. Representamos con la constante $Energy_i^{Switch}$ el consumo de energía, asociado a cada cambio de contexto, que sufrirá la tarea considerada. En nuestro caso, el consumo de energía de un cambio de contexto depende del número de líneas de memoria, seleccionadas por el algoritmo *Lock-MS*, que se cargarán en la *Lockable iCache*, y del consumo energético que se necesita para actualizar, tanto el LB, como el PB.

Para una tarea $Task_i$, el consumo de energía de un cambio de contexto se obtiene mediante la siguiente ecuación:

$$\begin{aligned} Energy_i^{Switch} &= (\ En_{ISRAM_{Rd(128)}} + En_{iC_{Wr(128)}}\ ) \cdot \sum_{l=0}^{Mlines-1} cached_l \\ &+ 2 \cdot (\ En_{ISRAM_{Rd(128)}} + En_{LB/PB_{Wr(128)}}\ ) \end{aligned} \tag{5.14}$$

El consumo de energía $taskEnergy_i$ de una tarea $Task_i$ durante la ejecución de un camino $Path_{i,j}$, teniendo en cuenta también el consumo energético de los cambios que sufre la tarea durante su ejecución, se calcula mediante la siguiente ecuación:

$$taskEnergy_i = pathEnergy_{i,j} + ncSwitch_i \cdot Energy_i^{Switch} \tag{5.15}$$

Función objetivo para optimizar el WCET o el WCEC

Como ya se ha comentado, al añadir las restricciones anteriores al problema ILP definido por el algoritmo *Lock-MS*, el *solver* también proporcionará el consumo de energía del *camino más largo*, que vendrá determinado por el valor de la constante $pathEnergy_{i,j}$ asociada al camino $Path_{i,j}$, cuyo tiempo de ejecución determina el WCET de la tarea $Task_i$.

Sin embargo, las restricciones anteriores, asociadas al consumo de energía de la jerarquía de memoria propuesta, también se pueden utilizar para definir una nueva función objetivo. Así pues, proponemos el cálculo del consumo de energía en el peor caso (WCEC/ *Worst Case Energy Consumption*). En este caso, el objetivo del análisis es obtener el WCEC de cada una de las tareas del sistema, por lo tanto la función objetivo del problema ILP se debe actualizar, ya que el consumo de energía del *camino más largo* de una tarea $Task_i$ no tiene por qué coincidir con el WCEC de la tarea.

A continuación definimos la nueva función objetivo *Wenergy*, que en este caso trata de minimizar el consumo de energía de la jerarquía de memoria en el peor caso. El nuevo algoritmo *Lock-Energy* selecciona las líneas de memoria a fijar en la cache, para minimizar el consumo de energía de la jerarquía de memoria durante la ejecución de una tarea.

Así pues, en el nuevo problema ILP, el WCEC de cada tarea del sistema se modela mediante el siguiente conjunto de restricciones:

$$wcec_i \quad \geq \quad pathEnergy_{i,j} \quad \forall\, 1 \leq j \leq NPaths_i \tag{5.16}$$

Por lo tanto, para cada tarea $Task_i$ del sistema, la nueva función objetivo del algoritmo *Lock-Energy* viene determinada por la siguiente ecuación a minimizar:

$$Wenergy_i \quad = \quad wcec_i + ncSwitch_i \cdot Energy_i^{Switch} \tag{5.17}$$

Finalmente conviene indicar que las dos funciones objetivo, tanto la del algoritmo *Lock-MS*, como la del algoritmo *Lock-Energy*, se pueden agrupar en una función objetivo general donde el peso de cada algoritmo dependa de un parámetro. El diseñador del sistema podría decidir, en función de estos parámetros, aplicar uno de los dos algoritmos o una combinación de ambos y asignar un peso determinado a cada uno de ellos. Por ejemplo, considerando el parámetro α para indicar el peso del WCET de la tarea y el parámetro β para indicar el peso del WCEC, la función objetivo global asociada a cada tarea $Task_i$ del sistema se representa con la siguiente ecuación:

$$Wglobal_i \quad = \quad \alpha \cdot Wcost_i + \beta \cdot Wenergy_i \tag{5.18}$$

Así pues, la función objetivo selecciona las líneas de memoria a bloquear en la *Lockable iCache*, para conseguir la máxima *planificabilidad* del sistema cuando α sea 1 y β sea 0, es decir, el *solver* resuelve el problema ILP asociado al algoritmo *Lock-MS*. Sin embargo la función objetivo selecciona las líneas de memoria a bloquear en la cache, para reducir el consumo energético en el peor caso de la tarea cuando α sea 0 y β sea 1, es decir, el *solver* soluciona el problema ILP asociado al algoritmo *Lock-Energy*. No obstante, el *solver* también puede solucionar el problema para valores de α y β cualesquiera.

Consumo energético de los circuitos de memoria

Como ya se ha comentado anteriormente, el consumo de energía de los circuitos de memoria depende del ancho del bus de datos y del modo de acceso. Por lo tanto es necesario diferenciar si se accede a una palabra de 32 bits o a una línea de memoria de 128 bits, así como si el acceso es una lectura (Rd) o una escritura (Wr). Para la jerarquía de memoria propuesta (ver Figura 5.3) en la Tabla 5.4 se muestran los valores de consumo energético de cada uno de los circuitos de memoria, en función del modo de acceso (Rd o Wr), y del tamaño (32 o 128 bits) de los datos implicados en dicha operación. Estos coeficientes se han obtenido mediante Cacti V.6.0 [138]. Además, conviene indicar que el consumo de energía se representa en pJ.

	Rd	Wr	Rd	Wr			Area
	32 b (ins./dat.)		128 b (mem. line)		Look-up	Fuga	(μm^2)
DeSRAM	38,50	38,37	—	—	—	0,05	1216188
IeSRAM	—	—	147,92	—	—	0,05	1216188
1 KB iC	1,07	—	—	1,44	0,09	0,07	7395
512 B iC	0,67	—	—	1,27	0,06	0,04	5928
256 B iC	0,46	—	—	1,16	0,04	0,02	4933
128 B iC	0,36	—	—	1,11	0,03	0,011	1587
64 B iC	0,31	—	—	1,08	0,02	0,007	584
LB/PB	0,08	—	—	0,27	—	0,002	36

Tabla 5.4: Consumo de energía para los modos de acceso de lectura (Rd) o escritura (Wr) de una palabra de 32 bit o una línea de 128 bit.

En concreto, en las dos primeras filas se indica el consumo energético de la *DeSRAM* y de la *IeSRAM*. Los acceso a la *IeSRAM* son para leer líneas de memoria de 128 bits, incluso durante la prebúsqueda. Sin embargo, los accesos a la *DeSRAM* sólo son de 32 bits, es decir, se lee o actualiza únicamente el dato solicitado.

En las filas siguientes se indica el consumo de energía de los accesos de lectura y escritura a una cache de correspondencia directa, cuya capacidad varía desde 64 bytes hasta 1 KB. En la última fila aparece el consumo de los circuitos LB y PB. En estos circuitos, al igual que en la cache, los accesos de lectura son de 32 bits, mientras que los accesos de escritura, para actualizar su contenido, son de 128 bits. Por lo tanto, en la *iCache*, el LB y el PB, las operaciones de lectura afectan a una instrucción, mientras que las operaciones de escritura afectan a una línea de memoria. En la columna *Look-up* se indica el consumo energético de una búsqueda en la *iCache* según su capacidad. Además, en la columna etiquetada como *Fuga*, se muestra la energía estática consumida por ciclo.

Finalmente, en la última columna se indica el área que ocupa cada uno de los circuitos de la jerarquía de memoria propuesta. También conviene indicar que

los valores que aparecen en blanco no son necesarios para evaluar el consumo de energía de la jerarquía de memoria propuesta.

Análisis del WCET y WCEC en el conjunto de tareas *small*

Proponemos el estudio del WCET y del WCEC en el conjunto de tareas *small*. En los experimentos realizados se analizan cuatro configuraciones de la jerarquía de memoria propuesta (ver Figura 5.3). En primer lugar se considera que todos los accesos se realizan directamente a la *eSRAM*, a continuación supondremos que la jerarquía de memoria sólo dispone de un LB, posteriormente se considera una jerarquía de memoria con LB y una *iCache* de correspondencia directa que varía su capacidad desde 4 a 64 conjuntos, es decir, desde 64 bytes a 1 KB de capacidad total, y finalmente analizamos la jerarquía de memoria completa con el PB.

Para cada configuración de memoria estudiada se han resuelto dos problemas ILP, el primero buscando la *planificabilidad* máxima del sistema ($\alpha = 1, \beta = 0$) y el segundo minimizando el consumo energético de la jerarquía de memoria analizada ($\alpha = 0, \beta = 1$). Los resultados que se muestran en las figuras se han normalizado al WCET y al WCEC obtenidos respectivamente para la primera configuración estudiada, donde todos los accesos se realizan directamente a la *eSRAM*. Los resultados conseguidos en las jerarquías de memoria, que tienen disponible la *iCache*, se muestran con barras verticales y los resultados logrados cuando la *iCache* no está disponible se indican con una línea horizontal. Para poder observar la influencia de la prebúsqueda, tanto en el WCET, como en el WCEC de las tareas analizadas, las figuras se han dividido en dos partes. En la parte de la izquierda se estudia una jerarquía de memoria sin prebúsqueda y en la parte de la derecha una jerarquía de memoria con prebúsqueda. Conviene indicar que en el cálculo del WCET y del WCEC de cada tarea se ha tenido en cuenta el efecto de los cambios de contexto. En el análisis del tiempo de respuesta y del consumo energético del sistema completo, para este conjunto de tareas *small*, también se ha tenido en cuenta el consumo de energía cuando el procesador está ocioso.

En las Figuras 5.10 y 5.11 se analiza el WCET y el WCEC de cada una de las tareas del conjunto *small*. Como se muestra en gráficas asociadas a cada tarea, el consumo de energía varía mucho más para cada una de las tareas que el tiempo de respuesta, que es bastante uniforme. En las gráficas se observa claramente que, tanto el WCET, como el WCEC de cada tarea, disminuyen al aumentar la capacidad de la *iCache*. Sin embargo debemos tener en cuenta que cuando la *iCache* es grande, es necesario considerar los costes de los cambios de contexto, ya que pueden influir negativamente, tanto en el WCET, como en el WCEC. En particular hay que añadir, tanto el tiempo, como el consumo de energía de cargar y bloquear los contenidos seleccionados en la *Lockable iCache*.

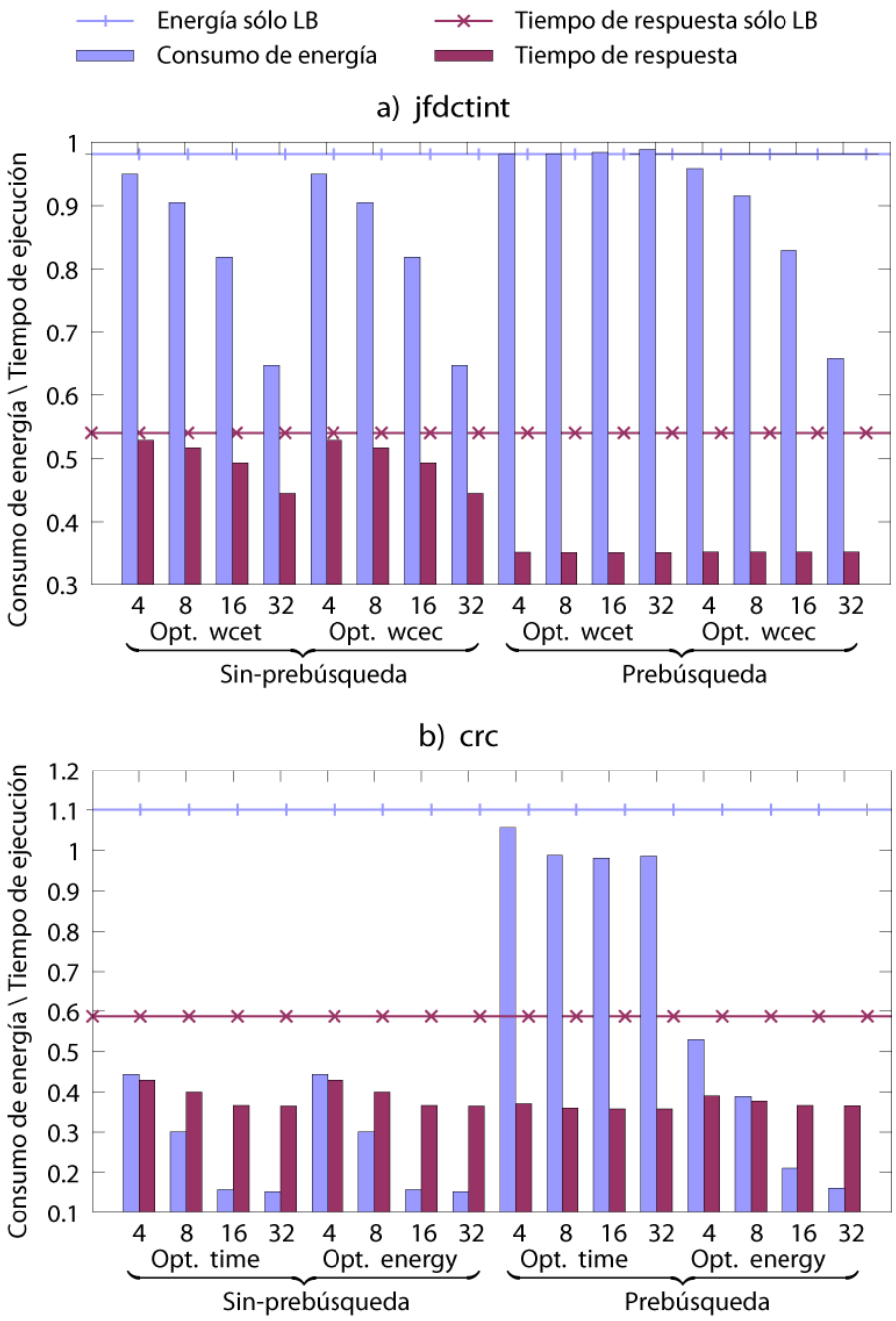


Figura 5.10: WCET y consumo de energía vs. WCEC y tiempo de ejecución de las tareas *jfdctint* y *crc*.

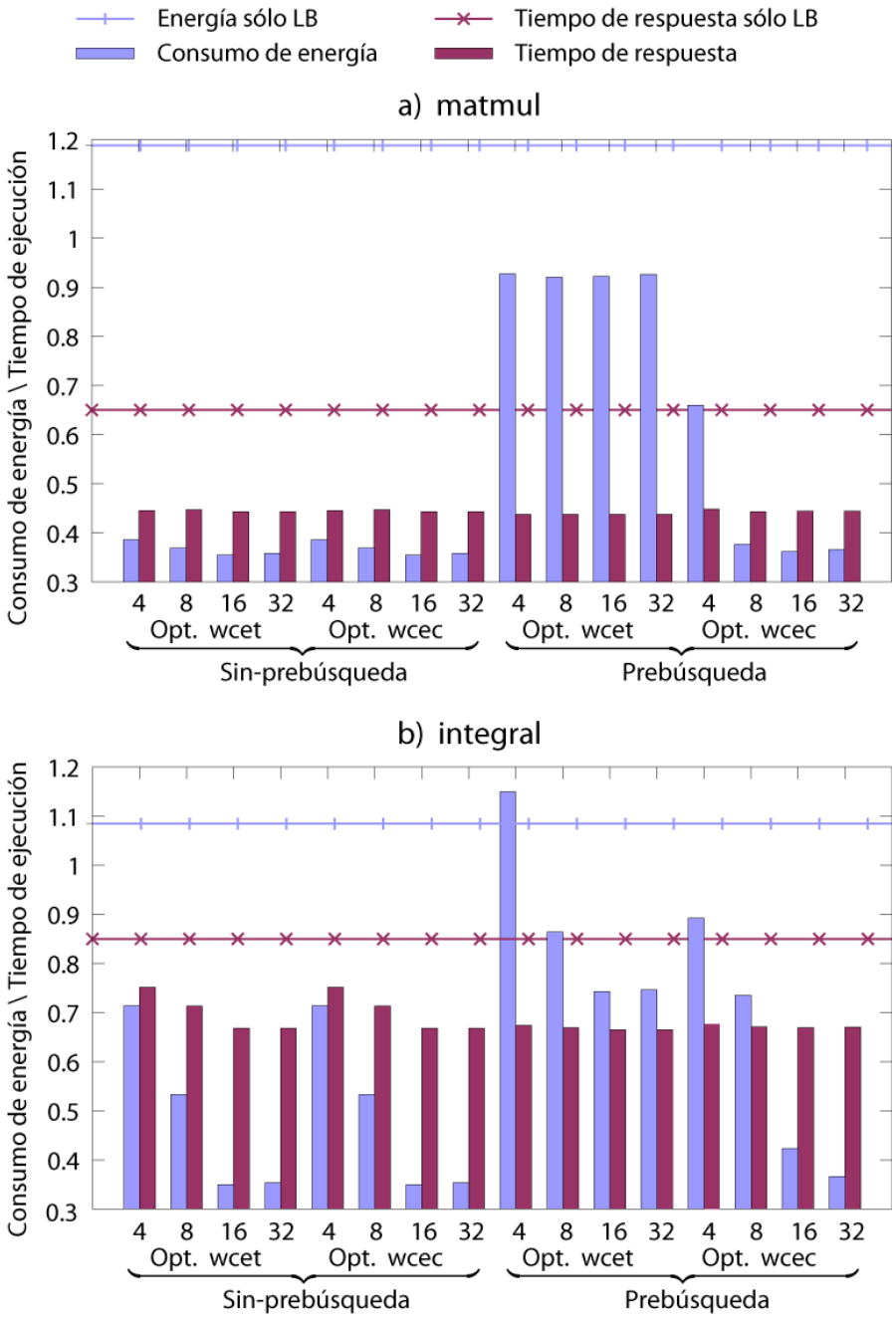


Figura 5.11: WCET y consumo de energía vs. WCEC y tiempo de ejecución de las tareas *matmult* e *integral*.

En la parte derecha de cada gráfica se analiza una jerarquía de memoria con *iCache*, LB y prebúsqueda. Cuando se optimiza el WCET disminuye el número de líneas que se bloquearán en la *iCache*, ya que la prebúsqueda es muy agresiva. En general, en códigos que contienen bucles con un único bloque básico formado por muchas líneas de memoria, se producen muchos fallos de capacidad. En estos casos, la cache no es lo suficientemente grande para almacenar estos bloques básicos. Sin embargo, en cada acceso a memoria se produce un acierto de prebúsqueda. Es decir, en las tareas donde el código es *plano*, la prebúsqueda funciona muy bien reduciendo el tiempo de ejecución de este tipo de tareas. Esta circunstancia se puede observar claramente en la Figura 5.10, donde se muestran los resultados del análisis de la tarea *jfdctint*. En los otros tres casos la prebúsqueda no tiene tanta importancia, ya que estas tareas no son códigos *planos* y el WCET es similar al conseguido sin prebúsqueda. Sin embargo el WCEC presenta resultados opuestos. La utilización de la prebúsqueda representa un importante consumo de energía que puede ser incluso mayor que cuando se accede directamente a la *eSRAM*. Esto se observa en las tareas de la Figura 5.11 *crc* y *matmul* para una jerarquía de memoria con una *Lockable iCache* de 4 conjuntos.

Cuando el objetivo es minimizar el WCEC de cada tarea, y la *iCache* es suficientemente grande, la prebúsqueda no se utiliza prácticamente, ya que los resultados son equivalentes a los obtenidos sin prebúsqueda. En estos casos, la *iCache* contiene las líneas de memoria suficientes para que la prebúsqueda influya muy poco, tanto en el WCET, como en el WCEC de las tareas. Pero en la tarea de código *plano jfdctint*, la prebúsqueda reduce considerablemente el WCET y no añade un consumo excesivo de energía.

Para concluir esta parte del estudio, conviene indicar que la prebúsqueda sólo es interesante cuando se ejecutan códigos *planos*, por lo tanto una aproximación práctica debería permitir habilitar y deshabilitar la prebúsqueda en función de las características de la tarea. Además, esta misma idea se podría aplicar a la *iCache* para configurar su capacidad a las necesidades de la tarea que se ejecuta, por ejemplo, deshabilitando algunos conjuntos. Así pues, la posibilidad de habilitar o deshabilitar los circuitos de memoria permitiría ajustar el rendimiento del sistema y ahorrar energía durante la ejecución, ya que se evitaría el consumo de energía estática.

Tanto el WCET, como el WCEC de cada tarea, dependen de sus periodos y del planificador, por lo que también es necesario estudiar el sistema completo. En la Figura 5.12 se analiza el tiempo de respuesta y el consumo de energía del sistema completo para el conjunto de tareas *small*. Para ello hemos medido el tiempo de respuesta y el consumo de energía de la tarea del conjunto con prioridad más baja.

Como se muestra en la Figura 5.12, el consumo de energía tiene mucha más variabilidad que el tiempo de respuesta. En general, la prebúsqueda incrementa

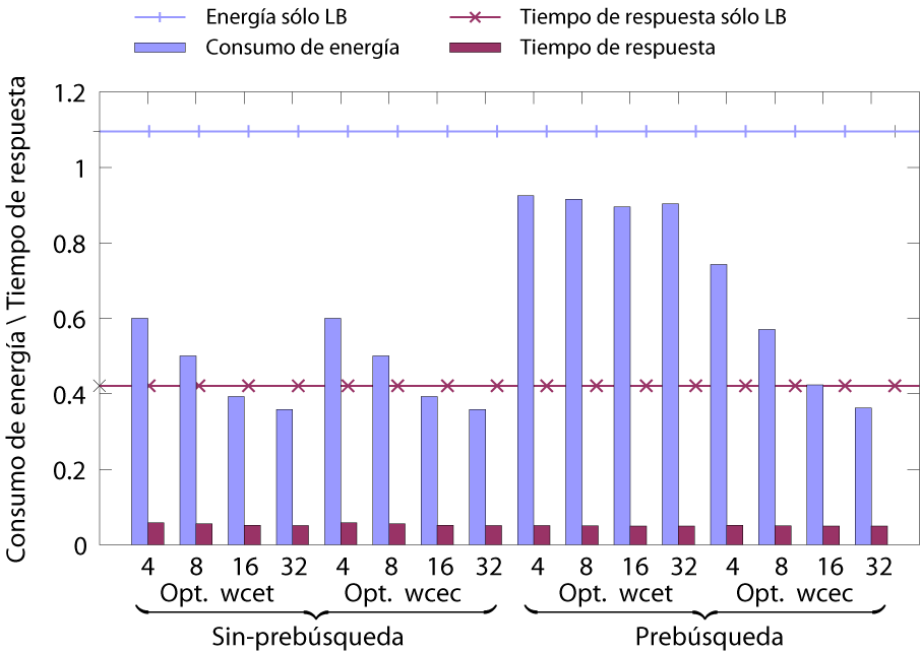


Figura 5.12: Tiempo de respuesta y consumo de energía vs. WCEC y tiempo de ejecución del conjunto de tareas *small*.

el consumo de energía del sistema y no siempre reduce el tiempo de respuesta, esto se indica claramente en la Figura 5.12. No obstante, al aumentar la capacidad de la cache debería mejorar considerablemente, tanto el tiempo de respuesta, como el consumo de energía, pero la mejora con respecto al tiempo de respuesta no se observa claramente en la Figura 5.12.

En sistemas de tiempo real es muy recomendable la utilización de un componente tan sencillo como el LB, porque reduce el tiempo de respuesta del sistema. Por ejemplo, como se observa en la Figura 5.12, el tiempo de respuesta de un sistema donde se accede directamente a la *eSRAM*, se reduce en un 60 % en un sistema con LB, pero el consumo de energía en un sistema con LB aumenta con respecto a un sistema de acceso directo a la *eSRAM*, ya que el consumo de energía en un sistema con LB está por encima de 1 y los resultados se muestran normalizados al caso donde se accede directamente a la *eSRAM*. En este caso el LB reduce el número de accesos a la *eSRAM*, mejorando el tiempo de ejecución, pero no se puede afirmar lo mismo con respecto al consumo de energía, ya que el consumo de energía cuando se accede directamente a la *eSRAM* es muy parecido al consumo de energía producido durante la ejecución de las 4 instrucciones que contiene el LB, como se puede verificar en la Tabla 5.4.

Este resultado pone de manifiesto que en un hardware sencillo como el LB,

aparecen diferencias significativas en el comportamiento del tiempo de respuesta y del consumo de energía del sistema en el peor caso.

5.5. Conclusiones

En tareas donde el código es *plano*, es decir, códigos que contienen bucles con un único bloque básico formado por muchas líneas de memoria, se producen muchos fallos de capacidad, ya que la cache no es lo suficientemente grande para almacenar todas las líneas de memoria que contiene uno de estos bloques básicos. Por lo tanto, una jerarquía de memoria formada por un LB y una *Lockable iCache* no es suficiente para reducir el tiempo de ejecución de este tipo de tareas.

En este capítulo hemos introducido la posibilidad de predecir el WCET con un hardware de prebúsqueda secuencial. Es decir, se propone una nueva jerarquía de memoria con prebúsqueda para un sistema de tiempo real (ver Figura 5.1). Para obtener la máxima *planificabilidad* del sistema, en esta nueva jerarquía de memoria, se presenta, tanto la extensión del algoritmo *Lock-MS*, como la extensión del *modelo compacto* de dicho algoritmo.

El tiempo de respuesta en un sistema ideal, donde cada acceso a una instrucción tiene un coste de un ciclo, es de aproximadamente un 67 %. Con un LB se puede obtener un rendimiento de hasta el 47 %, que representa una mejora relativa del 70 %. Mientras que con un LB y un PB se consigue un rendimiento de hasta el 60 %, que representa una mejora relativa de prácticamente el 90 %. Así pues, con una *Lockable iCache* de capacidad pequeña, de aproximadamente un 5 % del código del sistema, es posible alcanzar un rendimiento equivalente al de un sistema ideal. Además, si el código de la tarea es *plano*, se reduce significativamente el WCET sin necesidad de utilizar la *Lockable iCache*.

Finalmente hemos presentado un estudio sobre el consumo energético, como forma de decidir la configuración de esta nueva jerarquía de memoria para sistemas de tiempo real. De este estudio conviene destacar que el WCET y el WCEC no tienen correspondencia lineal, es decir, el camino del WCET de una tarea no suele coincidir con el camino del WCEC de dicha tarea. Sin embargo, el hardware de prebúsqueda aumenta considerablemente el consumo de energía. No obstante, puesto que el hardware de prebúsqueda reduce el WCET en las tareas donde el código es *plano*, cuando el consumo de energía sea una limitación importante en el diseño del sistema, una vez garantizada la *planificablidad* del mismo, se podría minimizar el consumo de energía de la jerarquía de memoria propuesta.

Capítulo 6

Conclusiones y trabajo futuro

La corrección de un sistema de tiempo real, no sólo está en función de los resultados obtenidos, sino que también depende del instante en el que dichos resultados son generados, por lo tanto, es esencial predecir su funcionamiento. Uno de los principales retos de los sistemas de tiempo real es el cálculo del tiempo de ejecución del peor caso (WCET/ *Worst Case Execution Time*), es decir, determinar el tiempo de ejecución del camino más largo. El cálculo del WCET tiene que ser seguro y también preciso, ya que la *planificabilidad* del sistema debe estar garantizada antes de su ejecución.

El mercado de los sistemas de tiempo real añade una restricción importante en el diseño de la jerarquía de memoria, la necesidad de conocer un límite máximo del tiempo de ejecución, ya que este tiempo depende en gran medida del número máximo de fallos de cache que se producirán durante la ejecución. Pero el análisis del comportamiento temporal de las memorias cache en el peor caso es complejo, por lo tanto los diseñadores de sistemas de tiempo real suelen descartar su utilización. Como solución al problema de la predicción del comportamiento de la cache de instrucciones, en esta Tesis se han propuesto técnicas de análisis y cálculo del WCET que predicen de forma exacta y precisa el funcionamiento en el peor caso de una cache de instrucciones.

Las estructuras condicionales dentro de bucles hacen que el análisis del WCET en presencia de caches adquiera complejidad exponencial, debido a las interferencias intrínsecas de la cache. En el Capítulo 3 se ha demostrado que el número de caminos que es necesario analizar, para determinar el comportamiento exacto de una cache de instrucciones con algoritmo de reemplazo LRU, no depende del número de iteraciones de los bucles, sino que está en función del número de caminos del condicional. También se ha presentado la técnica de *poda dinámica de caminos* que permite analizar y calcular el WCET de una tarea, en

presencia de una cache de instrucciones *convencional*. Para los casos en los que el número de caminos alternativos de un bucle no sea grande, la técnica de *poda dinámica de caminos*, reduce considerablemente la complejidad del problema y predice de forma exacta el comportamiento de la cache de instrucciones en el peor caso. Es decir, se puede predecir la contribución exacta al WCET de los accesos a la cache de instrucciones y se puede calcular el WCET de una tarea de una forma más precisa.

Para un sistema de tiempo real multitarea, en el Capítulo 4 se ha presentado el algoritmo *Lock-MS* para optimizar el rendimiento de una jerarquía de memoria formada por un LB (*Line Buffer*) y una cache de instrucciones que bloquea su contenido (*Lockable iCache*) durante algunos periodos de la ejecución del sistema. Al fijar el contenido de la cache su comportamiento es totalmente predecible y se evitan las interferencias de cache. El algoritmo *Lock-MS* está basado en ILP (*Integer Linear Programming*) y permite obtener un WCET seguro y preciso de cada una de las tareas del sistema. El objetivo de *Lock-MS* es seleccionar las líneas de memoria más adecuadas que se bloquearán en la cache, para obtener la máxima *planificabilidad* del sistema. Sin embargo, cuando el número de caminos de un programa es grande, no es posible representar todos los caminos mediante restricciones lineales. En esta Tesis también se ha presentado un *modelo compacto* del algoritmo *Lock-MS* que permite reducir el número de caminos del problema ILP, sin perder precisión en el WCET obtenido.

En el Capítulo 5 proponemos una nueva jerarquía de memoria con un hardware de prebúsqueda secuencial (PB/ *Prefetch Buffer*). Para obtener la máxima *planificabilidad* del sistema, en esta nueva jerarquía de memoria, presentamos, tanto la extensión del algoritmo Lock-MS, como la extensión del modelo compacto de dicho algoritmo. En general el hardware de prebúsqueda permite reducir el WCET y la capacidad de la cache de instrucciones considerablemente.

Como resultados experimentales más destacados, mostramos que el tiempo de respuesta del sistema con un LB se puede reducir hasta un 47 % aproximadamente, la cache de instrucciones podría aumentar el rendimiento en un 30 %. La reducción en un sistema con LB y PB puede suponer hasta un 60 %, que representa una mejora relativa de prácticamente el 90 %. En este caso, la cache de instrucciones sólo podría aumentar el rendimiento en un 10 %. Así pues, con una *Lockable iCache* de capacidad pequeña, de aproximadamente un 5 % del código del sistema, es posible alcanzar un rendimiento equivalente al de un sistema ideal, donde cada acceso a una instrucción tiene un coste de un ciclo.

Finalmente, en el Capítulo 5 se ha presentado un estudio sobre el consumo energético de esta jerarquía de memoria. Los resultados de este estudio indican que el camino del WCET de una tarea no coincide con el camino del WCEC (*Worst Case Energy Consumption*) de dicha tarea. Es decir, el camino que determina el WCET no suele ser el camino que consume mayor energía durante la ejecución. Así pues, se introduce la posibilidad de que el diseñador del sistema

decida si su objetivo es obtener un WCET más preciso o reducir el WCEC. El hardware de prebúsqueda reduce el WCET de los programas *planos* hasta obtener un rendimiento equivalente al caso ideal, pero aumenta considerablemente el consumo de energía de la jerarquía de memoria. En definitiva, el consumo energético debería ser un factor clave en el diseño de los sistemas empotrados, ya que muchos de estos sistemas son dispositivos móviles que funcionan con baterías.

Trabajo futuro

Actualmente estamos evaluando, en sistemas de tiempo real, las posibles mejoras en el rendimiento de la jerarquía de memoria propuesta en esta Tesis, al aplicar las técnicas de división de la cache (*cache partitioning*) [91, 92, 202]. Mediante estas técnicas se evitarían las penalizaciones por la recarga de la *Lockable iCache*. En una primera aproximación, para evaluar el máximo rendimiento que se podría conseguir, proponemos replicar todos los recursos disponibles para cada tarea del sistema. En este caso, el rendimiento obtenido es incluso mayor al que se podría lograr aplicando las técnicas de división de la cache. No obstante, en una jerarquía de memoria con prebúsqueda como la que proponemos en esta Tesis, los resultados indican que las mejoras en el rendimiento son inferiores al 10 % [162].

Como trabajo futuro proponemos analizar el comportamiento en el peor caso de los accesos a datos. Predecir el funcionamiento de la cache de datos es mucho más complejo, ya que para una misma instrucción de acceso a datos, las direcciones de memoria de los mismos pueden cambiar a lo largo de la ejecución. Por ejemplo, con una misma instrucción dentro de un bucle, se puede acceder a todos los componentes de un vector. Los accesos a datos también presentan diferentes comportamientos en función de si representan datos escalares, variables locales y globales, o bien se accede a una zona de memoria dinámica. Además, la dificultad de la predicción aumenta en los accesos a las referencias indirectas, cuya dirección no se puede determinar en tiempo de compilación, por ejemplo el acceso a datos mediante un puntero o mediante un vector de índices. En estos casos, la predicción del acceso siempre se debe considerar como fallo, ya que se podría afirmar que no se puede predecir estáticamente el funcionamiento de la cache de datos en códigos fuente con accesos a referencias indirectas y accesos a memoria dinámica. Por lo tanto, es habitual prohibir la utilización de este tipo de códigos fuente en tareas de tiempo real.

Hasta ahora, la mayor parte de los trabajos de investigación se han centrado en analizar y determinar el número de fallos de cache que se producen en los accesos a vectores y matrices dentro de bucles. Estos trabajos proponen predecir el funcionamiento de la cache de datos, aplicando la teoría del vector de reutilización [201]. Todas las propuestas presentadas se centran en resolver las denomi-

nadas ecuaciones de fallo de cache (CME/ *Cache Miss Equations*), para determinar el comportamiento de la cache de datos en el peor caso [58, 153, 183, 203]. Pero en sistemas de tiempo real multitarea, no parece ser la solución definitiva al problema, ya que este análisis sólo contempla la ejecución de una tarea aislada.

Al igual que con las caches de instrucciones, fijar o bloquear el contenido de la cache de datos puede ser una técnica interesante para predecir su funcionamiento. Pero muy pocos trabajos de investigación se han propuesto específicamente para determinar el funcionamiento de la cache de datos en el peor caso [180, 181, 182]. En estos trabajos se introduce un nuevo concepto sobre el rendimiento de la memoria en el peor caso, que se ha denominado *WCMP* (*Worst Case Memory Performance*). Esta técnica trata de analizar estáticamente el comportamiento de todos los accesos a la cache de datos y de determinar su impacto en el WCET de la tarea, pero la idea no es tan sencilla de aplicar como las presentadas para la predicción de la cache de instrucciones. En primer lugar, es necesario resolver las ecuaciones de fallo de cache, que en general sólo se suelen aplicar a bucles sin condiciones. En segundo lugar, es necesario analizar el código fuente para señalar, teniendo en cuenta las ecuaciones de fallo de cache, los diferentes bloques básicos en los que no se puede conocer, en algún instante de la ejecución, el contenido exacto de la cache de datos. Así pues, antes de ejecutar estos bloques básicos, es necesario fijar el contenido de la cache para que todos los accesos se puedan predecir de forma totalmente segura. Por lo tanto, es imprescindible la ayuda del compilador para añadir al principio de estos bloques nuevas instrucciones *lock*, para fijar la cache y al final de los mismos añadir instrucciones *unlock*, para dejar que la cache funcione con normalidad.

Sin embargo, el problema todavía no está resuelto, ya que es posible que la cache no tenga cargados los datos más adecuados a los que se accederá durante el periodo que permanezca bloqueada. Los resultados experimentales demuestran que si la cache está bloqueada y no contiene los datos más apropiados, el *WCMP* aumenta considerablemente durante la ejecución de un bloque básico. Por lo tanto, aparece un nuevo problema de difícil solución. Para reducir el *WCMP* de los bloques básicos, en los que la cache permanece bloqueada, es necesario seleccionar y cargar los datos a los que se accederá durante la ejecución de dicho bloque básico. Pero, la única propuesta para resolver este problema concreto vuelve a ser la aplicación de la técnica del vector de reutilización.

La técnica de cargar y bloquear la cache, durante la ejecución de determinados bloques básicos, hace posible el uso de las caches de datos en sistemas de tiempo real, ya que permite predecir de forma exacta su comportamiento. Desafortunadamente el impacto de añadir nuevas instrucciones *lock/unlock* y el coste de cargar los datos en la cache antes de fijar su contenido, no han sido evaluados. Obviamente el *WCMP* del programa se puede reducir considerablemente, pero el coste de ejecución de las nuevas instrucciones puede hacer que el WCET del programa, en vez de disminuir, aumente y empeore la *planificabilidad* del sistema.

Además, esta técnica se centra exclusivamente en el *WCMP* de una tarea de tiempo real que se ejecuta de forma aislada. Para resolver el problema de las interferencias extrínsecas que aparecen en sistemas multitarea con expulsiones, se podrían utilizar las técnicas para dividir o repartir la cache entre las tareas del sistema, que también se pueden aplicar a la cache de datos (*cache partitioning*) [91, 92, 202]. Sin embargo, la aproximación más sencilla para dividir la cache de datos es asignar a cada una de las tareas del sistema un trozo de cache de igual tamaño. Pero obviamente esta forma de repartir la cache no es la más adecuada, ya que no tiene en cuenta las características particulares de cada tarea.

Así pues, nuestro trabajo futuro se centrará en el análisis de la cache de datos para explotar su localidad espacial y temporal. En concreto, estamos analizando el comportamiento de una nueva organización predecible de la cache de datos para un sistema de tiempo real. Como primera solución al problema hemos propuesto una estructura ACDC (*Address-Cache*/*Data-Cache*) formada por una pequeña cache de datos (DC/ *Data-Cache*) y por una tabla (AC/ *Address-Cache*) que guarda las direcciones de las instrucciones que pueden actualizar el contenido de la DC [162]. En un sistema multitarea, la tabla AC debe actualizarse con las instrucciones de la tarea que se ejecutará para que éstas puedan actualizar la DC cuando sea necesario. La selección de estas instrucciones se obtiene a partir de la extensión del algoritmo *Lock-MS* propuesta para este fin. Por ejemplo, cuando esta nueva cache de datos se utiliza para explotar la localidad temporal en el acceso a variables escalares, funciona muy bien. También funciona bien cuando se utiliza para explotar la localidad espacial en los accesos secuenciales con paso 1. No obstante, la penalización por la recarga de la AC en cada cambio de contexto se debe tener en cuenta.

En sistemas de tiempo real, como conclusión general de la Tesis, proponemos la utilización de jerarquías de memoria cuyo funcionamiento se pueda predecir de forma exacta y precisa. En particular hemos propuesto una jerarquía de memoria formada por un LB, una *Lockable iCache* y un PB cuando el consumo de energía no representa una limitación en el diseño del sistema. Además, también es necesario predecir el comportamiento de los accesos a datos, ya que pueden reducir mucho más el WCET de las tareas y mejorar la *planificabilidad* del sistema.

Índice de figuras

Índice de tablas

Bibliografía

[1] Chart watch: High-performance embedded processor cores. *Microprocessor Report*, 22, 2008.

[2] H. Aljifri, A. Pons, and M. Tapia. Tighten the computation of worst-case execution-time by detecting feasible paths. In *In Proceedings of the IEEE International Performance, Computing and Communications Conference*, 2000.

[3] P. Altenbernd. On the false path problem in hard real-time programs. In *In Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, 1996.

[4] S. Altmeyer, C. Hümbert B. Lisper, and R. Wilhelm. Parametric timing analysis for complex architectures. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2008.

[5] L.C. Aparicio, J. Segarra, C. Rodríguez, and V. Viñals. Combining prefetch with instruction cache locking in multitasking real-time systems. In *Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2010.

[6] L.C. Aparicio, J. Segarra, C. Rodríguez, and V. Viñals. Improving the wcet computation in the presence of a lockable instruction cache in multitasking real-time systems. *Journal of Systems Architecture*, 2011.

[7] L.C. Aparicio, J. Segarra, C. Rodríguez, J. L. Villarroel, and V. Viñals. Avoiding the WCET overestimation on LRU instruction cache. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2008.

[8] A. Arnaud and I. Puaut. Towards a predictable and high performance use of instruction caches in hard real-time systems. In *Proceedings of the the work-in-progress session of the 15th Euromicro Conference on Real-Time Systems*, 2003.

[9] R. Arnold, F. Mueller, D. Whalley, and H. Harmon. Bounding worst-case instruction cache performance. In *Proceedings of the 15th IEEE Real-Time Systems Symposium*, 1994.

[10] P. Atanassov, S. Haberl, and P. Puschner. Heuristic worst-case execution time analysis. In *Proceedings of the 10th European Workshop on Dependable Computing*, 1999.

[11] N. Audsley and A. Burns. Real-time systems scheduling. Technical report, University of York, Departament of Computer Sciencie, York, United Kingdom, 1990.

[12] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 2002.

[13] T. Baker and A. Shaw. The cyclic executive model and ada. In *Proceedings of the 9th IEEE Real-Time Systems Symposium*, 1988.

[14] S. Basumallick and K. Nilsen. Cache issues in real-time systems. In *Proceedings of the 1st ACM SIGPLAN Workshop Languages, Compilers, and Tools for Real-Time Systems*, 1994.

[15] I. Bate and U. Khan. WCET analysis of modern processors using multi-criteria optimisation. *Empirical Software Engineering*, 2011.

[16] I. Bate and R. Reutemann. Worst-case execution time analysis for dynamic branch predictors. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, 2004.

[17] I. Bate and R. Reutemann. Efficient integration of bimodal branch prediction and pipeline analysis. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2005.

[18] G. Bernat. WCET a tool for probabilistic WCET analysis. In *Proceedings of the 3rd International Workshop on WCET analysis*, 2003.

[19] G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *Proceedings of the 25th Workshop on Real-Time Programming*, 2000.

[20] G. Bernat, A. Colin, and S. Petters. WCET analysis of probabilistic hard real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, 2002.

[21] G. Bernat, A. Colin, and S. Petters. pWCET: A tool for probabilistic worst-case execution time analysis of real-time systems. Technical report, University of York, Departament of Computer Sciencie, York, United Kingdom, 2003.

[22] G. Bernat, M. Newby, and A. Burns. Probabilistic timing analysis: an approach using copulas. *Journal of Embedded Computing*, 2005.

[23] A. Betts and G. Bernat. Tree-based WCET analysis on instrumentation point graphs. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, 2006.

[24] F. Bodin and I. Puaut. A WCET-oriented static branch prediction scheme for real time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, 2005.

[25] C. Burguiere and C. Rochange. A contribution to branch prediction modeling in wcet analysis. In *Proceedings of the conference on Design, Automation and Test in Europe*, 2005.

[26] C. Burguiere, C. Rochange, and P. Sainrat. A case for static branch prediction in real-time systems. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2005.

[27] A. Burns. Preemptive priority-based scheduling: an appropriate engineering approach. *Advances in real-time systems*, 1995.

[28] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[29] J. Busquets, D. Gil, P. Gil, and A. Wellings. Techniques to increase the schedulable utilization of cache-based preemptive real-time systems. *Journal of Systems Architecture*, 2000.

[30] J. Busquets and J. Serrano. The impact of extrinsic cache performance on predictability of real-time systems. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, 1995.

[31] J. Busquets, J. Serrano, and A. Wellings. Hybrid instruction cache partitioning for preemptive real-time systems. In *Proceedings of the 9th Euromicro Workshop on Real-Time Systems*, 1997.

[32] J. Busquets, A. Wellings, J. Serrano, R. Ors, and P. Gil. Adding instruction cache effect to an exact schedulability analysis of preemptive real-time systems. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, 1996.

[33] S. Bygde and B. Lisper. Towards an automatic parametric WCET analysis. In *Proceedings of the 8th International Workshop on Worst-Case Execution Time Analysis*, 2008.

[34] R. Chapman, A. Burns, and A. Welling. Integrated program proof and worst-case timing analysis of SPARK ada. In *Proceedings of the Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, 1994.

[35] T. Chen, T. Mitra, A. Roychoudhury, and V. Suhendra. Exploiting branch constraints without exhaustive path enumeration. In *Proceedings of the 5th International Workshop on WCET Analysis*, 2005.

[36] V. Chvátal. *Linear Programming.* W.H. Freeman & Company, 1983.

[37] J. Coffman, C. Healy, F. Mueller, and D. Whalley. Generalizing parametric timing analysis. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, 2007.

[38] A. Colin and G. Bernat. Scope-tree: A program representation for symbolic worst-case execution time analysis. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, 2002.

[39] A. Colin and I. Puaut. Worst case execution time analysis for a processor withbranch prediction. *Real-Time Systems*, 2000.

[40] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, 1977.

[41] C. Cullmann and F. Martin. Data-flow based detection of loop bounds. In *Proceedings of the 7th International Workshop on WCET Analysis*, 2007.

[42] J. Deverge and I. Puaut. Safe measurement-based WCET estimation. In *Proceedings of the 5rd International Workshop on WCET analysis*, 2005.

[43] J. Eisinger, I. Polian, B. Becker, S. Thesing, R. Wilhelm, and A. Metzner. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In *Proceedings of the 2006 IEEE Design and Diagnostics of Electronic Circuits and systems*, 2006.

[44] J. Engblom. *Worst-Case Execution Time Analysis For Optimized Code.* Master's thesis, The Faculty of Science and Technology, Uppsala University, Sweden, 1997.

[45] J. Engblom. Analysis of the execution time unpredictability caused by dynamic branch prediction. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2003.

[46] J. Engblom, P. Altenbernd, and A. Ermedahl. Facilitating worst-case execution time analysis for optimized code. In *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, 1998.

[47] J. Engblom and A. Ermedahl. Pipeline timing analysis using a trace-driven simulator. In *Proceedings of the 6th IEEE International Conference on Real-Time Computing Systems and Applications*, 1999.

[48] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, 2000.

[49] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Uppsala, Sweden, 2003.

[50] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. VDM Verlag, Saarbrücken, Germany, 2008.

[51] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *Proceedings of the 7th International Workshop on WCET Analysis*, 2007.

[52] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 1988.

[53] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise wcet determination for a real-life processor. In *Proceedings of the 1st International Workshop on Embedded Software*, 2001.

[54] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, 1997.

[55] C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, 1999.

[56] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-timesystems. *Real-Time Systems*, 1999.

[57] S. Gheorghitaan, S. Stuijk, T. Basten, and H. Corporaal. Automatic scenario detection for improved wcet estimation. In *Proceedings of the 42nd annual Design Automation Conference*, 2005.

[58] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 1999.

[59] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[60] H.G. Groß. *Measuring Evolutionary Testability of Real-Time Software*. PhD thesis, University of Glamorgan, Pontypridd, Wales, UK, 2000.

[61] H.G. Groß. A prediction system for evolutionary testability applied to dynamic execution time analysis. *Information and Software Technology*, 2001.

[62] H.G. Groß. An evaluation of dynamic, optimisation-based worst-case execution time analysis. In *Proceedings of the International Conference on Information Technology: Prospects and Challenges in the 21st Century*, 2003.

[63] H.G. Groß, B. Jones, and D. Eyres. Evolutionary algorithms for the verification of execution time bounds for real-time software. *IEE Colloquium on Applicable Modelling, Verification and Analysis Techniques for Real-Time Systems*, 1999.

[64] M. Grochtmann and J. Wegener. Evolutionary testing of temporal correctness. In *Proceedings of the 2nd International Software Quality Week Europe*, 1998.

[65] D. Grund, J. Reineke, and G. Gebhard. Branch target buffers: WCET analysis framework and timing predictability. In *Proceedings of the 2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2009.

[66] D. Grund, J. Reineke, and G. Gebhard. Branch target buffers: WCET analysis framework and timing predictability. *Journal of Systems Architecture*, 2011.

[67] J. Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Department of Computer Engineering, Mälardalen University, Västerås, Sweden, and Department of Computer Systems, Information Technology, Uppsala University, Uppsala, Sweden, 2000.

[68] J. Gustafsson. A prototype tool for flow analysis of object-oriented programs. In *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2002.

[69] J. Gustafsson. Worst case execution time analysis of object-oriented programs. In *Proceedings of the 6th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, 2002.

[70] J. Gustafsson and A. Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. In *Proceedings of the Joint Workshop on Parallel and Distributed Real-Time Systems*, 1997.

[71] J. Gustafsson and A. Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. *Journal of Parallel and Distributed Computing Practices*, 1998.

[72] J. Gustafsson and A. Ermedahl. *Automatic derivation of path and loop annotations in object-oriented real-time programs*. Nova Science Publishers, Inc., Commack, NY, USA, 2001.

[73] J. Gustafsson and A. Ermedahl. Merging techniques for faster derivation of WCET flow information using abstract execution. In *Proceedings of the 8th International Workshop on Worst-Case Execution Time Analysis*, 2008.

[74] J. Gustafsson, A. Ermedahl, and B. Lisper. Algorithms for infeasible path calculation. In *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis*, 2006.

[75] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, 2006.

[76] J. Gustafsson, B. Lisper, N. Bernmudo, C. Sandberg, and L. Sjöberg. A prototype tool for flow analysis of c programs. In *Proceedings of the 2th International Workshop on WCET Analysis*, 2002.

[77] J. Gustafsson, B. Lisper, C. Sandberg, and N. Bernmudo. A tool for automatic flow analysis of c-programs for wcet calculation. In *Proceedings of the 8th International Workshop on Object-Oriented Real-Time Dependable Systems*, 2003.

[78] P. Harter. Response times in level-structured systems. *ACM Transactions on Computer Systems*, 1987.

[79] C. Healy, R. Arnold, F. Mueller, M. Harmon, and D. Walley. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 1999.

[80] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium*, 1998.

[81] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 2000.

[82] C. Healy, D. Whalley, and M. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, 1995.

[83] J. Hennessy and D. Patterson. *Computer Arquitecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Francisco, CA, Fourth edition 2007.

[84] N. Holsti, T. Langbacka, and S. Saarinen. Worst-case execution-time analysis for digital signal processors. In *Proceedings of the 10th European Signal Processing Conference*, 2000.

[85] W. Hsu and J. Smith. A performance study of instruction cache prefetching methods. *IEEE Transactions on Computers*, 1998.

[86] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th annual international symposium on Computer architecture*, 1997.

[87] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 1986.

[88] R. Kamal. *Embedded Systems: Architechture, Programming and Design.* Programming and Design. McGraw-Hill Higher Education, Boston, London, 2008.

[89] D. Kebbal. Automatic flow analysis using symbolic execution and path enumeration. In *Proceedings of the 2006 International Conference Workshops on Parallel Processing*, 2006.

[90] U. Khan and I. Bate. WCET analysis of modern processors using multi-criteria optimisation. In *Proceedings of the 1st International Symposium on Search Based Software Engineering*, 2009.

[91] D. Kirk. Process dependent static partitioning for real-time systems. In *Proceedings of the 9th IEEE Real-Time Systems Symposium*, 1988.

[92] D. Kirk. SMART (strategic memory allocation for real-time) cache design. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, 1989.

[93] R. Kirner. *Integration of static runtime analysis and program compilation.* Master's thesis, Institut für Technische Informatik, University of Technology, Vienna, Austria, 2000.

[94] R. Kirner. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis.* PhD thesis, Institut für Technische Informatik, University of Technology, Vienna, Austria, 2003.

[95] R. Kirner and P. Puschner. Transformation of path information for wcet analysis during compilation. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, 2001.

[96] R. Kirner and P. Puschner. Timing analysis of optimized code. In *Proceedings of the 8th International Workshop on Object-Oriented Real-Time Dependable Systems*, 2003.

[97] R. Kirner, P. Puschner, and I. Wenzel. Measurement-based worst-case execution time analysis using automatic test-data generation. In *Proceedings of the 4th Euromicro International Worshop on WCET Analysis*, 2004.

[98] R. Kirner, I. Wenzel, B. Rieder, and P. Puschner. Using measurements as a complement to static worst-case execution time analysis. *Intelligent Systems at the Service of Mankind*, 2005.

[99] A. Kountouris. Safe and efficient elimination of infeasible execution paths in WCET estimation. In *Proceedings of the 3rd International Workshop on Real-Time Computing Systems Application*, 1996.

[100] C. Lee, J. Hahn, Y. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim. Analysis of cache-related preemption delay in fixed-priority preemtive scheduling. *IEEE Transactions on Computers*, 1998.

[101] C. Lee, K. Lee, J. Hahn, Y. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim. Bounding cache-related preemption delay for real-time systems. *IEEE Transactions on Software Engineering*, 2000.

[102] X. Li, T. Mitra, and A. Roychoudhury. Modeling control speculation for timing analysis. *Real-Time Systems*, 2005.

[103] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for software timing analysis. In *Proceedings of the 25th IEEE Real-Time Systems Symposium*, 2004.

[104] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for wcet analysis. *Real-Time Systems*, 2006.

[105] Y. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, 1995.

[106] Y. Li, S. Malik, and A. Roychoudhury. Accurate timing analysis by modeling caches, speculation and their interaction. In *Proceedings of the 40th annual Design Automation Conference*, 2003.

[107] Y. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, 1996.

[108] S. Lim, Y. Bae, G. Jang, B. Rhee, S. Min, C. Park, H. Shin, K. Park, S. Moon, and C. Kim. An accurate worst case timing analysis for risc processors. *IEEE TRansactions on Software Engineering*, 1995.

[109] S. Lim, J. Kim, and S. Min. A worst case timing analysis technique for optimized programs. In *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications*, 1998.

[110] B. Lisper. Fully automatic, parametric worst-case execution time analysis. In *Proceedings of the Workshop on Worst-Case Execution Time Analysis*, 2003.

[111] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 1973.

[112] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2009.

[113] C. Luk and T. Mowry. Cooperative prefetching: compiler and hardware support for effective instruction prefetching in modern processors. In *Proceedings of the ACM/IEEE International symposium on Microarchitecture*, 1998.

[114] T. Lundqvist and P. Stenström. Integrating path and timing analysis using instruction-level simulation techniques. In *Proceedings of the SIGPLAN Worksop on Languages, Compilers and Tools for Embedded Systems*, 1998.

[115] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, 1999.

[116] T. Lundqvist and P. P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 1999.

[117] A. Martí. *Utilización de memorias cache con bloqueo en sistemas de tiempo real.* PhD thesis, Departamento de Informática de Sistemas y Computadores. Univeridad Politécnica de Valencia, España, 2003.

[118] A. Martí, A. Perles, and J. Busquets. Static use of locking caches in multitask preemptive real-time systems. In *Proceedings of the IEEE Real-Time Embedded Systems Workshop*, 2001.

[119] A. Martí, A. Perles, and J. Busquets. Dynamic use of locking caches in multitask, preemptive real-time systems. In *Proceedings of the 15th World Congress of the International Federation of Automatic Control*, 2002.

[120] A. Martí, A. Perles, F. Rodriguez, and J. Busquets-Mataix. Static use of locking caches vs. dynamic use of locking caches for real-time systems. In *Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering*, 2003.

[121] A. Martí, A. Perles, S. Sáez, and J. Busquets. Performance analysis of the static use of locking caches. In *Proceedings of the 3rd WSEAS International Conference on Automation and Information*, 2002.

[122] A. Martí, A. Perles, S. Sáez, and J. Busquets. Performance comparison of use of locking caches under static and dynamic schedulers. In *Proceedings of the 27th IFAC/IFIP/IEEE Workshop in Real-Time Programming*, 2003.

[123] A. Martí, I. Puaut, A. Ivars, and J. Busquets. Cache contents selection for statically-locked instruction caches: an algorithm comparison. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, 2005.

[124] A. Marti, A. Perez, A. Perles, and J. Busquets. Using genetic algorithms in content selection for locking-caches. In *Proceedings of the IASTED International Conference on Applied Informatics*, 2001.

[125] P. Marwedel. *Embedded System Design*. Kluwer Academic Publishers, 2003.

[126] M. Michiel, A. Bonenfant, H. Cassé, and P. Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2008.

[127] T. Mitra and A. Roychoudhury. A framework to model branch prediction for wcet analysis. In *Proceedings of the Workshop on Worst-Case Execution Time Analysis*, 2002.

[128] T. Mitra, A. Roychoudhury, and X. Li. Timing analysis of embedded software for speculative processors. In *Proceedings of the 15th international symposium on System Synthesis*, 2002.

[129] J. Mogul and A. Borg. The effect of context switches on cache performance. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[130] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[131] F. Mueller. *Static Cache Simulation and its Applications*. PhD thesis, Departamet of Computer Science, Florida State University, USA, 1994.

[132] F. Mueller. Generalizing timing predictions to set-associative caches. In *Proceedings of the 9th Euromicro Workshop on Real-Time Systems*, 1997.

[133] F. Mueller. Timing predictions for multi-level caches. In *In ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, 1997.

[134] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 2000.

[135] F. Mueller and J. Wegener. A comparison of static analysis and evolutionary testing for the verification of timing constraints. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium*, 1998.

[136] F. Mueller and D. Whalley. Fast instruction cache analysis via static cache simulation. In *Proceedings of the 28th Annual Simulation Symposium*, 1995.

[137] F. Mueller, D. Whalley, and M. Harmon. Predicting instruction cache behavior. In *In ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, 1993.

[138] N. Muralimanohar, T. Balasubramonian, and N. Jouppi. Cacti 6.0: A tool to understand large caches. Technical report, University of Utah and Hewlett Packard Laboratories, 2007.

[139] H. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/software codesign and system synthesis*, 2003.

[140] M. O'Sullivan, S. Vössner, and J. Wegener. Testing temporal correctness of real-time systems - a new approach using genetic algorithms and cluster analysis. In *Proceedings of the 6th European Conference on Software Testing, Analysis & Review*, 1998.

[141] G. Ottosson and M. Sjodin. Worst-case execution time analysis for modern hardware architectures. In *Proceedings ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, 1997.

[142] C. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 1993.

[143] G. Park, O. Kwon, T. Han, S. Kim, and S. Yang. An improved lookahead instruction prefetching. In *Proceedings of the High-Performance Computing on the Information Superhighway*, 1997.

[144] H. Pohlheim and J. Wegener. Testing the temporal behavior of real-time software modules using extended evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 1999.

[145] G. Pospischil, P. Puschner, A. Vrchoticky, and R. Zainlinger. Developing real-time tasks with predictable timing. *IEEE Software*, 1992.

[146] I. Puaut. Cache analysis vs static cache locking for schedulability analysis in multitasking real-time systems. In *Proceedings of 2nd International Workshop on Worst-Case Execution Time Analysis*, 2002.

[147] I. Puaut and D Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings on the 23rd IEEE Real-Time Systems Symposium*, 2002.

[148] P. Puschner. Worst-case execution time analysis at low cost. *Control Engineering Practice*, 1998.

[149] P. Puschner and Ch. Koza. Calculating the maximum, execution time of real-time programs. *Real-Time Systems*, 1989.

[150] P. Puschner and R. Nossal. Testing the results of static worst-case execution-time analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, 1998.

[151] P. Puschner and A. Schedl. A tool for the computation of worst case task execution times. In *Proceedings of the 5th Euromicro Workshop on Real-Time Systems*, 1993.

[152] P. Puschner and A. Schedl. Computing maximum task execution times - A graph-based approach. *Real-Time Systems*, 1997.

[153] H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *Proceedings on the 11th IEEE Real Time and Embedded Technology and Applications Symposium*, 2005.

[154] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 2007.

[155] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. In *Proceedings on the 6th International Workshop on Worst-Case Execution Time Analysis*, 2006.

[156] G. Reinman, B. Calder, and T. Austin. Fetch directed instruction prefetching. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, 1999.

[157] R. Reutemann. *Worst-Case Execution Time Analysis for Dynamic Branch Predictors*. PhD thesis, Departamet of Computer Science, York University, UK, 2008.

[158] F. Rossi, P. Beek, and T. Walsh. *Handbook Of Constraint Programming*. Elsevier, 2006.

[159] C. Sandberg, A. Ermedahl, J. Gustafsson, and B. Lisper. Faster WCET flow analysis by program slicing. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems*, 2006.

[160] J. Schneider and C. Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. In *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, 1999.

[161] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, Inc., New York, NY, USA, 1986.

[162] J. Segarra, C. Rodríguez, R. Gran, L.C. Aparicio, and V. Viñals. A small and effective data cache for real-time multitasking systems. In *Proceedings of the 18th IEEE Real Time and Embedded Technology and Applications Symposium*, 2012.

[163] L. Sha, T. Abdelzaher, K. Arzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 2004.

[164] A. Smith. Sequential program prefetching in memory hierarchies. *Computer*, 1978.

[165] J. Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, 1981.

[166] J. Souyris, E. Le Pavec, G. Himbert, V. Jégu, G. Borios, and R. Heckmann. Computing the wcet of an avionics program by abstract interpretation. In *Proceedings of the 5th International Workshop on Worst-Case Execution Time Analysis*, 2005.

[167] V. Srinivasan, E. Davidson, G. Tyson, M. Charney, and T. Puzak. Branch history guided instruction prefetching. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001.

[168] F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 2000.

[169] F. Stappert, A. Ermedahl, and J. Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2001.

[170] J. Staschulat and R. Ernst. Multiple process execution in cache related preemption delay analysis. In *Proceedings of the 4th ACM international conference on Embedded software*, 2004.

[171] J. Staschulat and S. Schlieckerand R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, 2005.

[172] I. Stein and F. Martin. Analysis of path exclusion at the machine code level. In *Proceedings of the 7th International Workshop on Worst-Case Execution Time Analysis*, 2007.

[173] E. Tamura, F. Rodríguez, J. Busquets, and A. Martí. High performance memory architectures with dynamic locking cache for real-time systems. In *Proceedings of the 16th Work in Progress Euromicro Conference on Real-Time Systems*, 2004.

[174] H. Theiling and C. Ferdinand. Combining abstract interpretation and ilp for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, 1998.

[175] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. *Real-Time Systems*, 2000.

[176] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software. *Proceedings of the International Conference on Dependable Systems and Networks*, 2003.

[177] N. Tracey. *A Search-Based Automated Test-Data Generation Framework for Safety Critical Software*. PhD thesis, Departamet of Computer Science, York University, UK, 2000.

[178] N. Tracey, J. Clark, J. McDermid, and K. Mander. A search-based automated test-data generation framework for safety-critical systems. In *Systems engineering for business process change*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[179] J. Tse and A. Smith. Cpu cache prefetching: Timing evaluation of hardware implementations. *IEEE Transactions on Computers*, 1998.

[180] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *Procedings of the International Conferebce on Measurement and Modeling of Computer Systems*, 2003.

[181] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, 2003.

[182] X. Vera, B. Lisper, and J. Xue. Data cache locking for tight timing calculations. *ACM Transactions on Embedded Computing Systmes*, 2007.

[183] X. Vera and J. Xue. Let's study whole program cache behaviour analytically. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, 2002.

[184] E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric timing analysis. In *Proceedings of the 2001 ACM SIGPLAN workshop on Optimization of middleware and distributed systems*, 2001.

[185] A. Vrchoticky. Compilation support for fine-grained execution time analysis. Technical report, 1/1994, Technische Universität Wien, Institut für Technische Informatic, Treitlst. Vienna, Austria, 1994.

[186] J. Wegener, K. Grimm, M. Grochtmann, and H. Sthamer. Systematic testing of real-time systems. In *Proceedings of the 4th European Conference on Software Testing, Analysis & Review*, 1996.

[187] J. Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, 1998.

[188] J. Wegener, M. Grochtmann, and B. Jones. Testing temporal correctness of real-time systems by means of genetic algorithms. In *Proceedings of the 10th International Software Quality Week*, 1997.

[189] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 2001.

[190] J. Wegener, H. Pohlheim, and H.Sthamer. Testing the temporal behavior of real-time tasks using extended evolutionary algorithms. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, 1999.

[191] J. Wegener, H. Sthamer, B.F. Jones, and D.E. Eyres. Testing real-time systems using genetic algorithms. *Software Quality Control*, 1997.

[192] I. Wenzel. *Measurement-Based Timing Analysis of Superscalar Processors.* PhD thesis, Institut für Technische Informatik, Treitlstr. Viena, Austria, 2006.

[193] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Proceedings of the Fifth International Conference on Quality Software*, 2005.

[194] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner. Measurement-based worst-case execution time analysis. In *Proceedings of the 3rd IEEE Worshop on Software Technologies for Future Embedded and Ubiquitous Systems*, 2005.

[195] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner. Measurement-based time analysis. In *Proccedings of the 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, 2008.

[196] I. Wenzel, B. Rieder, R. Kirner, and P. Puschner. Automatic timing model generation by cfg partitioning and model checking. In *Proceedings of the conference on Design, Automation and Test in Europe*, 2005.

[197] R. Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In *VMCAI*, 2004.

[198] R. Wilhelm, J. Engbolhm, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution time problem: Overview of the methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 2008.

[199] N. Williams. WCET measurement using modified path testing. In *Proceedings of the 5th International Workshop on WCET Analysis*, 2005.

[200] N. Williams, B. Marre, P. Mouy, and M. Roger. PathCrawler: Automatic generation of path tests by combining static and dynamic analysis. In *Proceedings of the 5th European Dependable Computing Conference*, 2005.

[201] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, 1991.

[202] A. Wolfe. Software-based cache partitioning for real-time applications. In *Proceedings of the International Worksop on Responsive Computer Systems*, 1993.

[203] J. Xue and X. Vera. Efficient and accurate analytical modeling of whole-program data cache behavior. *IEEE Transactions on Computers*, 2004.

[204] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *Real Time Systems*, 1993.

[205] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 1997.