Obtención del WCET óptimo con caches de instrucciones bloqueables (Lock-MS) en Otawa*

Alba Pedro-Zapater†, Clemente Rodríguez‡, Juan Segarra†, Rubén Gran†, y Víctor Viñals-Yúfera†

†Dpt. Informática e Ingeniería de Sistemas, Universidad de Zaragoza, España
Instituto de Investigación en Ingeniería de Aragón (I3A), U. Zaragoza, España
‡Dpt. Arquitectura y Tecnología de Computadores, U. País Vasco, España
†‡Red de Excelencia HiPEAC-3 (European FET FP7/ICT 287759)
†{albapz,jsegarra,rgran,victor}@unizar.es, ‡acprolac@ehu.es

Resumen El WCET de un programa es difícil de calcular debido a la falta de predictibilidad de las caches convencionales. En este trabajo hemos implementado un módulo de análisis del WCET desde el binario del programa para la herramienta Otawa para una cache bloqueable con el método *Lock-MS*. Este módulo automatiza la creación de las restricciones ILP para el cálculo del WCET y de las lineas seleccionadas de la cache bloqueable. Esta automatización nos permite disponer de un entorno de experimentación productivo y de amplia aplicabilidad. En este trabajo hemos utilizado este entorno para estudiar el impacto en el WCET de los niveles de optimización, la configuración de cache y la latencia a memoria. Los resultados obtenidos muestran las posibilidades del uso de este módulo para benchmarks más grandes y complejos.

Keywords: WCET, Cache de Instrucciones, Lock-MS, Otawa

1. Introducción

Uno de los principales retos en el estudio de los sistemas de tiempo real estricto (HRTS: *Hard Real Time Systems*) es el cálculo del tiempo de ejecución en el peor caso (WCET: *Worst Case Execution Time*) o en su defecto una cota superior lo más ajustada posible. Aunque existen métodos probabilísticos que calculan el WCET basándose en medidas realizadas sobre ejecuciones concretas, dichos métodos no garantizan formalmente que los valores obtenidos sean una cota superior del WCET (e.g. [5,6]). Para garantizar formalmente dichas cotas, es necesario utilizar métodos de análisis estático (e.g. [3,7,9,11]). Estos métodos se basan en el análisis del código del programa y en su grafo de flujo de control, partiendo del código fuente o desde el binario. El análisis del código fuente es

* Este trabajo ha sido parcialmente financiado por los proyectos TIN2013-46957-C2-1-P, Consolider NoE TIN2014-52608-REDC (Gob. España), gaZ: grupo de investigación T48 (Gob. Aragón y European ESF), Unizar (JIUZ-2015-TEC-06), y la beca FPU14/02463.

problemático ya que el compilador puede transformar el flujo original del programa (optimizaciones). En cambio, el análisis del binario puede suponer perder información semántica que está presente en el código fuente.

Una de las principales dificultades para obtener un WCET preciso mediante el análisis del código es que algunos componentes hardware tienen una latencia variable, por ejemplo la jerarquía de memoria. Aunque las memorias cache convencionales reducen significativamente el tiempo medio de acceso a memoria, el coste de cada acceso es difícil de predecir ya que depende de los accesos anteriores y los parámetros de cache (reemplazo, asociatividad, tamaño, etc.). Como resultado, la diferencia entre el WCET real y la cota calculada puede ser excesiva.

Para evitar la dificultad de predecir de forma precisa el comportamiento de las caches convencionales, una propuesta es usar caches cuyo contenido está prefijado (lock-caches) y su mecanismo de reemplazo está deshabilitado. De esta forma, dado que el contenido es conocido y no cambia, el cálculo de aciertos y fallos es trivial y no depende de la secuencia de accesos. El inconveniente de fijar los contenidos en cache es, por una parte determinar buenos contenidos a fijar, y por otra parte que al no actualizarse los contenidos en ejecución se va a limitar la explotación del reuso temporal y espacial de la aplicación. En el caso de la cache de instrucciones, se ha demostrado que el uso de un *line-buffer* es analizable para el cálculo de WCET y permite adaptar el contenido en la jerarquía de memoria al flujo de ejecución del programa [3,13]. El *line-buffer* tiene el tamaño de un bloque de cache y actúa como una cache convencional de un solo bloque.

Lock-MS es un método análisis estático de WCET sobre caches bloqueables [3]. Este método, además de calcular el WCET, obtiene el contenido óptimo de la cache bloqueable. Es decir, dada cierta cache bloqueable, obtiene el contenido que minimiza el WCET para cierta tarea. Este método se basa en generar un modelo de programación lineal entera (ILP) donde los grados de libertad del modelo son, para cada uno de los bloques de memoria, precargarlo en cache o no. Además de para caches de instrucciones bloqueables, este método ha sido extendido para analizar otros componentes hardware tales como prebuscadores de instrucciones [2], caches de datos específicas para tiempo real [14,15] y también para optimizar el WCET teniendo en cuenta el consumo energético en el peor caso [8].

Una de las mayores desventajas de los métodos de análisis estático es la falta de herramientas automáticas que los implementen. Esto se debe a que los métodos a implementar son relativamente recientes y orientados a campos muy especializados, y las implementaciones no suelen reconocerse como méritos científicos. Además, los métodos de análisis estático suelen ser métodos formales complejos y requieren información difícil de conseguir a partir de un simple fichero ejecutable.

En este trabajo, hemos implementado un módulo de análisis del WCET desde el binario del programa para la herramienta Otawa [4]. Este módulo es capaz de generar las restricciones ILP (*Integer Linear Programming*) del método Lock-MS [3], que permiten calcular el contenido de la cache de instrucciones que

minimiza el WCET del programa. A diferencia de herramientas anteriores, conociendo las cotas en el número de iteraciones de los bucles, el análisis y generación de las restricciones ILP se realiza de manera automática. Esto permite analizar el WCET de un gran número de experimentos de forma productiva y sencilla variando tanto métricas hardware como software. A resaltar que vamos a poder observar la influencia del compilador en el WCET, algo poco usual debido a la dificultad de trabajar con optimizaciones.

La estructura del articulo es la siguiente: en la Sección 2, hacemos una breve introducción a la herramienta de Otawa y se explica el módulo implementado para la generación de las restricciones ILP; en la Sección 3 presentamos nuestro entorno experimental; en la Sección 4 presentamos los resultados de los experimentos realizados y finalmente en la Sección 5 recogemos las conclusiones y presentamos el trabajo futuro.

2. Otawa

Otawa es un entorno de análisis estático de binarios desarrollado por el grupo TRACES del *Institut de Recherche en Informatique* de Toulouse [4]. Otawa permite analizar programas ejecutables de múltiples arquitecturas (PowerPC, ARM, SPARC o M68HCS) con el objetivo de analizar su WCET en presencia de diferentes estructuras de procesador y de memoria cache. Para ello Otawa genera el grafo del flujo de control (CFG, *Control Flow Graph*) que incorpora la información de todos los bloques básicos de instrucciones y sus relaciones de precedencia. En la Figura 1 podemos ver un ejemplo del CFG que genera Otawa para un pequeño código de ejemplo.

Todas las técnicas de análisis del WCET incorporadas en Otawa se basan en enriquecer y manipular el CFG para alcanzar su objetivo. Cada módulo de Otawa es un paso de transformación que va añadiendo información al CFG y resultados parciales. Esta estructura modular permite combinar módulos existentes (detección de bucles en el CFG, por ejemplo) con módulos nuevos. Estos módulos también aportarán lo necesario para la formulación de las restricciones ILP con las que se calcula el WCET del método *Lock-MS*.

2.1. Implementación del módulo Lock-MS

El módulo que hemos implementado en Otawa analiza el CFG de un binario y elabora el sistema ILP correspondiente. Su resolución proporciona tanto el WCET del programa como los bloques de cache que deben seleccionarse para la cache bloqueable de Instrucciones. El sistema ILP se genera siguiendo el método *Lock-MS*, lo cual asegura que el WCET obtenido es el menor posible con el mínimo número de bloques en la cache bloqueable [3]. En la Figura 2 podemos ver el grafo enriquecido por nuestro módulo del programa de la Figura 1 (a). El módulo *Lock-MS* realiza una búsqueda en profundidad en el CFG para obtener cada uno de los caminos posibles (dos caminos en el ejemplo de la Figura 2 (a)). Después calcula el número máximo de veces que se va a ejecutar cada bloque

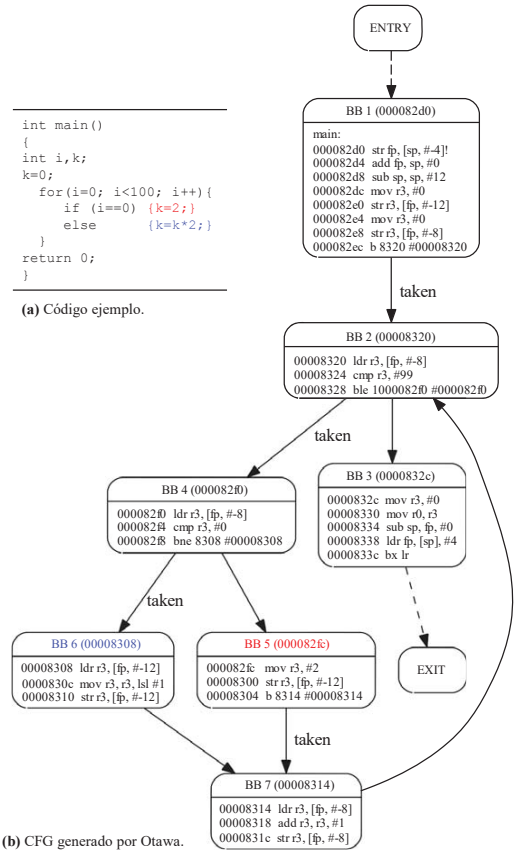

Figura 1. Ejemplo de código fuente (a) junto con su representación CFG de OTAWA a partir de un binario ARM (b).

básico que compone el CFG (por ejemplo el bloque básico 6 (BB6) se ejecuta un máximo de 100 veces). También se calcula toda la información referente a la configuración de la cache (tamaño de bloque, tamaño de cache y asociatividad), como podemos ver en la Figura 2 (a) donde aparece la información sobre a qué bloque de memoria pertenece cada instrucción y a qué conjunto pertenece cada bloque. Por ejemplo, en el bloque básico 6 (BB 6), la última instrucción pertenece al bloque 4 al cual le corresponde el conjunto 0 (S0 L4).

Con toda esta información se construye el sistema ILP y la representación gráfica del CFG que estamos estudiando, que nos facilitará el análisis de los datos obtenidos. En la Figura 2 (b) se muestra parte del sistema ILP correspondiente al programa de ejemplo. Podemos ver las restricciones correspondientes a cada camino y aquellas correspondientes a la configuración de cache; el conjunto al que pertenece cada bloque, y cuantos bloques puede haber en cada conjunto (uno en este caso).

Finalmente, como ya señalábamos, la resolución de este sistema nos proporciona tanto el WCET final, como los bloques que deben ser cacheados para conseguir un configuración óptima (WCET mínimo). Cabe destacar que este sistema minimiza también el número de bloques seleccionados.

2.2. Reducción del número de caminos

Algunos de los benchmarks analizados en este trabajo por su tamaño o por su complejidad, presentan un número muy alto de caminos (*crc*, *integral*, y *qurt*, ver Sección 3). En estos casos hemos contabilizado cientos de caminos, llegando a más de 50000 con nivel de optimización -O0 en *qurt*. Como consecuencia el tiempo necesario para resolver el sistema ILP aumenta considerablemente.

Para estos casos, hemos hecho una modificación del módulo para reducir el número de caminos, y por lo tanto el número de restricciones y complejidad del sistema ILP. El número de caminos aumenta de forma exponencial debido a la existencia de condicionales concatenados. En ciertos casos, el CFG original puede verse como varios subgrafos concatenados (cada uno de ellos con un único nodo de entrada y otro de salida). En esos casos, las propiedades de las caches bloqueadas hacen que el WCET global sea igual a la suma de los WCETs de cada subgrafo [3]. Siendo así, la modificación consiste en compactar el sistema ILP analizando los subgrafos independientemente para luego sumar los WCETs resultantes de cada uno, con lo que se reduce el tiempo para su resolución. La mejora será más o menos significativa dependiendo del número de caminos y complejidad del programa, y de los subgrafos que se hayan podido detectar. La división en subgrafos ha reducido notablemente el coste tal y como se puede ver en la Tabla 1 para *crc* compilado con -O0. Cabe destacar que incluso *qurt* compilado con -O0 no se puede analizar sin compactar caminos debido a que, como ya hemos dicho, tiene más de 50000 caminos.

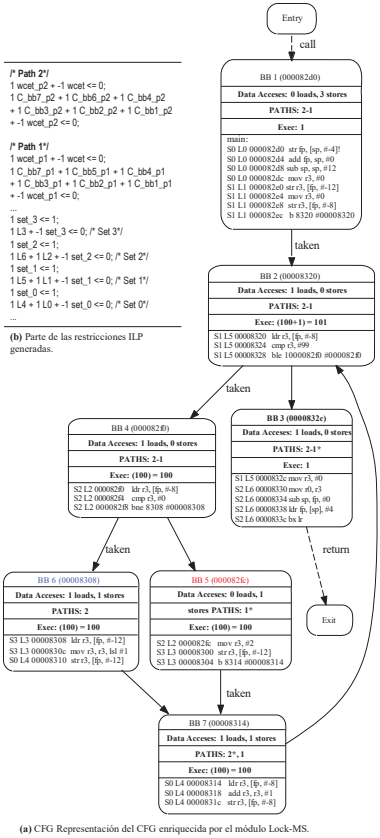

Figura 2. (a) CFG enriquecido del programa de la Figura 1 con los caminos numerados, el número máximo de ejecuciones por bloque básico, el bloque y conjunto de cache de cada instrucción y el número de instrucciones de datos. (b) Parte de las instrucciones ILP generadas por el módulo para este CFG. (a) y (b) tienen la siguiente configuración de cache: Tamaño Bloque 16B, Tamaño Cache 64B y Asociatividad 1.

Tiempo (s)	Calculando Caminos	Construyendo ILP	Resolviendo ILP
No compacto	0,036	3,925	162,62
Compacto	0,002	0,009	0,17

Tabla 1. Tiempos de procesamiento del WCET para *crc* con nivel de optimización -O0 medido en un portátil con un procesador Intel CORE i7

3. Entorno experimental

Para nuestros experimentos hemos utilizado parte de la colección SNU-RT Benchmark Suite for Worst Case Timing Analysis [16]: *jfdctint*, *lms*, *crc*, *matmult*, *integral*, *qurt* y *fibcall*. Además hemos utilizado dos benchmarks propios: *Matmult_opti* que es una versión optimizada en cuanto al acceso a datos de *matmult* y *gauss* que resuelve un sistema de ecuaciones lineales por el método de Gauss. Todos los límites de los bucles de estos benchmarks son conocidos, y no hay recursión.

Los binarios están generados para la arquitectura ARM v5t, usando el compilador cruzado *arm-none-eabi-gcc* versión 4.8.4 con distintos niveles de optimización. Este compilador cruzado considera por defecto que las operaciones con números reales se hacen por software. Sin embargo nosotros forzamos la generación de aritmética con instrucciones de coma flotante. También se ha configurado el compilador para evitar las sustituciones de parte del código con funciones de bibliotecas optimizadas (e.g. que sustituya asignaciones a cero en bucles por *memset*). Esto nos permite acotar adecuadamente los bucles al impedir el uso de funciones externas desconocidas. Respecto al procesador asumimos una segmentación ideal de dos etapas (E1: búsqueda de instrucción; E2: ejecución y lectura/escritura de operandos/resultados), añadiendo un ciclo a las instrucciones de memoria para que tengan más peso. También asumimos que no hay otros componentes con latencia variable (cache de datos, predictor de saltos, ejecución fuera de orden. etc.) más allá de la cache de instrucciones. Muchos sistemas para tiempo real pueden modelarse así, o bien porque no disponen de los mecanismos citados, o bien porque estos mecanismos se desconectan para impedir su variabilidad temporal. Por ejemplo el procesador Leon 4 permite desconectar su predictor de saltos [12]. Estas consideraciones también se asumen en estudios previos [10,13].

Para demostrar la capacidad de análisis del nuevo módulo lock-MS hemos realizado un barrido experimental amplio de opciones software y hardware. Las variables de experimentación y sus rangos son los siguientes:

- Niveles de optimización en compilación: -O0, -O1, -O2 y -O3.
- Sistema cache bloqueada y *line-buffer*:
  - Tamaño de bloque: 16, 32 y 64 Bytes.
  - Tamaño de cache bloqueada: Sin cache (solo *line-buffer*), infinita, 16, 32, 64, 128, 256, 512 y 1024 Bytes.

  - Asociatividad: 1, 2, 4 y completamente asociativa
- Acceso a datos:
  - 1 ciclo (e.g. datos precargados en cache/scratchpad)
  - Tiempo de acceso a memoria (sistema sin cache de datos)
- Acceso a memoria (fallo en instrucciones o acceso a datos): 1 (igual al coste de acierto), 5, 10 y 20 ciclos.

4. Resultados

Para cada configuración de la cache bloqueable, latencia de memoria y nivel de optimización hemos utilizado el módulo *Lock-MS* para calcular la selección de bloques a fijar con la que se obtiene el WCET óptimo. Ahora bien, puesto que los benchmarks analizados, por su pequeño tamaño, no son del todo representativos de tareas reales en aplicaciones actuales, no es intención de este trabajo extraer conclusiones sobre la interacción entre compilación y jerarquía de memoria. Este interesante objetivo se abordará en un trabajo posterior que considerará aplicaciones de prueba de mayor entidad como las contenidas en la colección TACLe [1].

Sin embargo el análisis de los resultados de estos benchmarks, aunque preliminar, aporta conclusiones significativas. Un caso interesante es *crc*. Las Tablas 2 y 3 muestran el comportamiento de *crc* variando la mayoría de los parámetros descritos. En estas tablas el tamaño de bloque es 16B, la latencia de accesos a datos es 1 ciclo y la latencia de fallo de instrucciones es 5 ciclos (excepto para el WCET ideal, que se especifica que es 1 ciclo).

En la Tabla 2 para cada nivel de optimización tenemos el número de bloques y de caminos en la segunda y tercera columna. A continuación, en la cuarta y quinta columna, tenemos el WCET ideal (latencia de acceso de instrucciones en caso de fallo 1 ciclo, ningún bloque de cache resulta seleccionado) y el WCET con cache bloqueable infinita. La diferencia entre ellos es que con la cache bloqueable infinita se tiene en cuenta el coste de cargar los bloques. En la última columna, aparece el número de bloques seleccionados para la cache infinita; ya que como nuestro sistema ILP minimiza el número de bloques cargados, este número no corresponde necesariamente al total de bloques del programa (columna 2). De esta tabla ya podemos extraer interesantes conclusiones sobre el sistema, ya que si el WCET requerido es menor que los mostrados en la tabla (en este caso el menor es el obtenido con nivel de optimización -O3) será necesario replantear el sistema. También observamos que el número de caminos y de bloques varía para cada nivel de optimización, y que un número mayor de bloques y caminos no implica un mayor WCET. Por ejemplo, tanto para el WCET ideal como para el WCET con cache infinita, el nivel de optimización -O3 tiene menor WCET que los niveles de optimización -O1 y -O2 pero tiene más caminos y más bloques.

En la tabla 3 cada columna corresponde a un tamaño de cache desde 0 hasta 64 bloques, y dentro de cada columna tenemos el WCET para cada nivel de optimización (O0-O3) agrupadas por las distintas asociatividades (1, 2, 4 y completamente asociativa). Para los tamaños de cache 32 y 64 (512 y 1024 Bytes) la

				Cache Bloqueable Infinita	
Opt	**Tamaño (bloques)**	**Núm. caminos**	**WCET ideal (Lat. mem.=1)**	**WCET**	**Núm. Bloques seleccionados**
O0	68	1296	203829	204144	55
O1	29	576	60361	60503	26
O2	23	576	54521	54637	20
O3	38	784	45943	46122	31

Tabla 2. En las columnnas: Tamaño (en bloques de 16 B), número de caminos posibles, WCET con latencia a memoria 1 (en ciclos), y WCET (en ciclos) y número de bloques seleccionados para la cache bloqueable de tamaño infinito y en las filas: distintos niveles de optimización del benchmark *crc*.

segunda columna recoge el número bloques seleccionados en la cache bloqueable; como ya pasaba en el caso de cache infinita, se seleccionan menos bloques que el total de bloques del programa (columna 2 de la tabla 2). Cuando no aparece esta segunda columna significa que se usan todos los bloques de la cache bloqueable.

Como ya hemos señalado nuestro sistema ILP minimiza el número de bloques en la cache, es decir, si tiene el mismo coste poner un bloque o no ponerlo, no lo pone. Esto nos ofrece la posibilidad de desconectar bloques de cache para ahorrar energía estática.

El WCET con cache infinita (columna 5 de la tabla 2) es una cota inferior de lo que se puede llegar a conseguir con una cache. El número de bloques seleccionados con cache infinita (columna 6 de la tabla 2) es el mínimo número de líneas necesarias en la cache para conseguir el mejor WCET, siempre y cuando no haya problemas de contención por usar una configuración de cache (número de conjuntos y grado de asociatividad) inadecuada para el benchmark. Estas cotas nos permiten verificar que los datos obtenidos son coherentes ya que se verifica que dado un nivel de optimización ningún WCET para ninguna configuración de cache de la tabla 3 es menor que el de la cache infinita (en negrita los que han alcanzado esa cota). De manera similar, ninguna selecciona un mayor número de bloques que los seleccionados en la cache infinita.

También podemos observar la influencia de la capacidad y de la asociatividad en el aumento del WCET. Por ejemplo para tamaño 16 (256B) con nivel de optimización -O1 observamos que hay aumento de WCET por debajo de asociatividad 4 (en cursiva los datos del ejemplo). De manera similar podemos ver la influencia de la capacidad. Por ejemplo, en el nivel de optimización -O2 y tamaños de cache menores que 32 (512B) nunca se alcanza el WCET obtenido con la cache infinita.

También hay que resaltar que el nivel de optimización -O3 no siempre es el mejor. Por ejemplo, en el caso del tamaño de cache 4 (64B) y asociatividad 1 el nivel de optimización -O2 es el que nos da un menor WCET. Esto implica que se deben estudiar todos los niveles de optimización para saber dónde se encuentra el menor WCET posible, aumentando el espacio de experimentación

		Tamaño de cache bloqueable, en número de bloques (de 16B)									
		0	1	2	4	8	16	32		64	
1-asoc	O0	376673	358246	339819	307061	269049	246849	224347	30	204147	54
	O1	144205	125770	107343	85697	78389	*70181*	**60503**	26	**60503**	26
	O2	128977	110550	92123	70477	63153	54649	**54637**	20	**54637**	20
	O3	91855	88636	86193	82083	73879	57471	46128	29	46125	30
2-asoc	O0			339819	307061	267873	246825	216461	32	204147	54
	O1			107343	85697	73125	*67837*	**60503**	26	**60503**	26
	O2			92123	70477	58777	54649	**54637**	20	**54637**	20
	O3			86193	82083	73879	57471	46128	29	**46122**	31
4-asoc	O0				307061	267865	244497	210341	32	204147	54
	O1				84825	73125	*60533*	**60503**	26	**60503**	26
	O2				70477	58777	54649	**54637**	20	**54637**	20
	O3				82083	73879	57471	46128	29	**46122**	31
Cpl-asoc	O0					267865	242177	209461	32	**204144**	55
	O1					71949	*60533*	**60503**	26	**60503**	26
	O2					58769	54649	**54637**	20	**54637**	20
	O3					73879	57471	**46122**	31	**46122**	31

Tabla 3. En las columnas, tamaño de la cache bloqueable en bloques, y en las filas el WCET (en ciclos) para cada nivel de optimización agrupadas por las distintas asociatividades. Si el número de bloques seleccionados es menor que los del programa, aparecen en una columna junto al WCET correspondiente

a analizar. Por último, generamos tablas similares a la anterior (no mostradas en este artículo) para valores distintos de latencia a memoria, tanto de instrucciones como de datos, y diferentes tamaños de bloque tanto para *crc* como para el resto de los benchmarks sin observar comportamientos cualitativamente diferentes a los descritos. Queda por analizar si con benchmarks más grandes y complejos se encontrarán comportamientos distintos.

5. Conclusiones y trabajo futuro

En este trabajo se ha presentado la implementación de un módulo de Otawa que permite analizar el WCET de binarios en presencia de una cache bloqueable junto a un *Line-Buffer*. Como resultado de este análisis se obtiene tanto el WCET como los bloques que se cargarán inicialmente en la cache bloqueable. Automatizar el proceso de análisis del WCET y de selección de los bloques nos permite barrer una gran cantidad de parámetros software y hardware para ver la sensibilidad del WCET. Debido al pequeño tamaño de los benchmarks analizados se ha realizado un análisis de poca profundidad que sin embargo nos permite hacernos una buena idea general de todo el potencial que puede tener el uso de este módulo para el posterior análisis de benchmarks con más entidad y complejidad.

Los resultados obtenidos resultan coherentes lo cual nos permite presumir la corrección del módulo. Además como aportación novedosa de este trabajo, hemos realizado un análisis de la influencia en el WCET del nivel de optimización aplicado por el compilador.

Como continuación de este trabajo, vamos a abordar los siguientes problemas: metodología a seguir en caso de benchmarks grandes e incluir en nuestro módulo el análisis de la jerarquía de memoria para datos.

5.1. Algoritmos para programas más grandes

Como ya hemos explicado en la Sección 2, uno de los problemas con los que nos hemos encontrado ha sido que debido al tamaño y/o complejidad, algunos benchmarks tienen un gran número de caminos. Esto tiene un impacto directo en el tiempo necesario para su enumeración y para la posterior resolución del sistema ILP asociado a todos estos caminos. Por ello, y para poder resolver benchmarks más grandes y complejos, estamos trabajando en un algoritmo basado en la teoría de grafos para simplificar el CFG detectando subgrafos. La versión del módulo utilizada en este trabajo permite incluir manualmente esta propuesta de división para los programas más complejos (Sección 2). En la versión final, vamos a ser capaces de automatizar la división del grafo y el posterior análisis de cada uno de los subgrafos. Para cada uno de estos subgrafos el número de caminos a considerar sería sustancialmente menor, facilitando la resolución del ILP. Esto también nos permitirá detectar distintas fases en el programa, pudiendo elegir distintos contenidos de la cache bloqueable para cada fase. Durante la ejecución del programa, estos puntos de cambios de fase serían candidatos para realizar la actualización del contenido de la cache bloqueable, y así, ajustar mejor su contenido a los requerimientos de la nueva fase y por lo tanto mejorar el WCET.

5.2. Integración de ACDC

De manera similar a cómo hemos planteado la integración de la cache bloqueable y el *line-buffer* para instrucciones en Otawa, se añadiría la cache de datos ACDC [15] para conseguir un análisis de WCET para accesos a memoria más preciso y completo. Además esto aportará una herramienta para que esta propuesta de diseño de cache de datos para tiempo real pueda ser evaluada con distintos benchmarks, caches de instrucciones, pipelines, etc.

Referencias

1. Timing analysis on code-level (tacle). http://www.tacle.eu/.
2. 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 IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications*, pages 319–328, Macau SAR, China, August 2010. IEEE Computer Society Press.

3. 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*, 57(7):695–706, August 2011.
4. Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Otawa: An open toolbox for adaptive wcet analysis. In *Proceedings of the 8th IFIP WG 10.2 International Conference on Software Technologies for Embedded and Ubiquitous Systems*, SEUS'10, pages 35–46, Berlin, Heidelberg, 2010. Springer-Verlag.
5. Guillem Bernat, Antoine Colin, and Stefan M. Petters. WCET analysis of probabilistic hard real-time systems. In *Proceedings of the 23rd Real-Time Systems Symposium RTSS 2002*, pages 279–288, 2002.
6. Francisco J. Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery Berger, Jaume Abella, Franck Wartel, Michael Houston, Luca Santinelli, Leonidas Kosmidis, Code Lo, and Dorin Maxim. Proartis: Probabilistically analyzable real-time systems. *ACM Trans. Embed. Comput. Syst.*, 12(2s):94:1–94:26, May 2013.
7. 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*, 21(4):703–746, 1999.
8. R. Gran, J. Segarra, C. Rodríguez, L. C. Aparicio, and V. Viñals. Optimizing a combined wcet-wcec problem in instruction fetching for real-time systems. *Journal of Systems Architecture*, 59(9):667–678, October 2013.
9. Y. T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 254–264, December 1996.
10. A. Martí Campoy, Á. Perles Ivars, and J. V. Busquets Mataix. Static use of locking caches in multitask preemptive real-time systems. In *IEEE Real-Time Embedded System Workshop*, December 2001.
11. F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2-3):217–247, May 2000.
12. Quad Core LEON4 SPARC V8 Processor. Data Sheet and User's Manual. `http://www.gaisler.com/doc/LEON4-N2X-DS.pdf`, 2015. [Online; accessed 21-Juny-2016].
13. I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proc. of the IEEE Real-Time Systems Symp.*, December 2002.
14. J. Segarra, C. Rodríguez, R. Gran, L. C. Aparicio, and V. Viñals. A small and effective data cache for real-time multitasking systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 45–54, Beijing, China, April 2012.
15. Juan Segarra, Clemente Rodríguez, Rubén Gran, Luis C. Aparicio, and Víctor Viñals. ACDC: Small, predictable and high-performance data cache. *ACM Trans. Embed. Comput. Syst.*, 14(2):38:1–38:26, February 2015.
16. Seoul National University Real-Time Research Group. SNU-RT benchmark suite for worst case timing analysis, 2008.

