Proyecto Fin de Máster.
Curso 2007-2008.

Técnicas HW/SW para reducir la presión sobre la jerarquía de memoria

Autor:
Rodrigo González Alberquilla

Directores del proyecto:
Francisco Tirado Fernández
Luis Piñuel Moreno

Facultad de Informática.
Universidad Complutense de Madrid.

Índice general

Capítulo 1

Introducción

En este capítulo inicial primero se define el problema al que nos queremos enfrentar y a continuación se muestran las dos perspectivas bajo las que puede abordarse, la software y la hardware. Después se exponen la motivación y los objetivos concretos del proyecto, y se finaliza con una descripción de la organización de esta memoria.

1.1. El *memory gap* y la jerarquía de memoria

La diferencia de velocidad entre el procesador y la memoria ha supuesto un freno importante al rendimiento de los computadores prácticamente desde sus inicios. Para superar este obstáculo, los diseñadores han propuesto un sinfín de técnicas a lo largo de las últimas cuatro décadas, de entre las cuales, la más destacada es, sin duda alguna, la organización jerárquica de la memoria, empleando diferentes niveles de memorias cache. Sin embargo, a medida que los transistores disminuían de tamaño, fruto de la evolución tecnológica, esta

diferencia no ha cesado de incrementarse, haciendo necesarias soluciones más efectivas.

Esta diferencia de velocidad, a la que se suele denominar con el término anglosajón de *memory gap*, ha contribuido en gran medida al cambio radical que han sufrido los procesadores en los últimos tiempos, y que ha conducido a los actuales diseños multi-núcleo. Hasta hace poco, la mayor capacidad de integración, proporcionada por cada nueva tecnología, se solía aprovechar para incrementar la frecuencia de reloj y dotar al procesador de mecanismos más sofisticados de extracción de paralelismo a nivel de instrucción (ILP). Sin embargo, se ha llegado a un punto en el cual el elevado consumo de energía, unido al escaso ILP de la mayoría de los programas, ha hecho tambalear la aplicación de la ley de Moore al rendimiento, originando un punto de inflexión en el diseño de procesadores. Como es de común conocimiento, en el contexto actual, es más eficiente incorporar varios procesadores dentro del chip (incluso más sencillos) antes que elevar la frecuencia o complicar el diseño.

La responsabilidad del *memory gap* en este cambio tiene que ver con la merma del paralelismo a nivel de instrucción que supone la elevada latencia de los accesos a los niveles más bajos de la jerarquía de memoria (los más alejados del procesador). Cuando uno de estos accesos se produce, el procesador tan solo puede continuar su ejecución hasta agotar sus recursos de planificación (ventana de instrucciones) y posteriormente se para hasta que el acceso termina. Cuanto mayor sea la latencia, mayor número de ciclos tendrá que aguardar y menor será el ILP efectivo. Una solución obvia a este problema consiste en incrementar dichos recursos de planificación de instrucciones. No obstante, a pesar de los esfuerzos de muchos investigadores por aumentar el tamaño de

la ventana de instrucciones de manera efectiva, esta solución no suele ser suficiente por sí sola. Para aumentar el paralelismo a nivel de instrucción y por ende el rendimiento, resulta imprescindible mejorar la jerarquía de memoria, bien reduciendo las ocurrencias de tales accesos, bien mermando su latencia.

1.2. Hardware vs Software

El problema del *memory gap* puede ser atacado desde dos perspectivas distintas, la hardware y la software. Como se ha citado previamente, la solución más notable a este problema ha sido la jerarquización de la memoria, que es una solución puramente hardware. No obstante, una inadecuada planificación de los accesos por parte del compilador/programador puede entorpecer el trabajo de ésta, hasta el punto de convertirla en un obstáculo para el rendimiento del computador. Aunque existen soluciones complementarias que permiten, en cierta medida, contrarrestar los efectos perniciosos de semejantes códigos, como por ejemplo la prebúsqueda y el *bypass* de niveles, la parte software del problema no puede ser desdeñada. Es más, en el contexto actual, en el que la jerarquía de memoria consume la mayor parte de los recursos del procesador, tanto en área como en consumo de energía, las soluciones software e híbridas resultan especialmente atractivas.

1.3. Motivación

Como de es de común conocimiento, la mayoría de variables de un programa se emplean para guardar resultados temporales y comunicar datos entre

funciones. Idealmente, estas variables deberían alojarse en los registros del procesador, ya que de ese modo, tanto el rendimiento como el consumo serían óptimos. Sin embargo, este recurso es escaso y rara vez permite almacenarlas todas, debiendo el resto permanecer en memoria. Es obvio, por lo tanto, que las decisiones respecto a dónde se aloja cada variable en cada momento, y cuándo se transfiere de un tipo de almacenamiento a otro, tienen un importante efecto en la jerarquía de memoria ya que determinan en gran medida la cantidad y secuencia de los accesos que se realizan sobre ella. No obstante, eclipsado por el incesante incremento de capacidad y niveles de la jerarquía de memoria, auspiciado por la evolución de la tecnología de fabricación de procesadores, este aspecto no ha recibido la suficiente atención.

1.4. Objetivos

Tomando como plataforma de evaluación uno de los tipos de procesador más extendidos, los RISC de 32 bits para sistemas empotrados, este trabajo se plantea dos objetivos distintos y complementarios. El primero tiene una perspectiva software y consiste en evaluar el impacto en la jerarquía de memoria de la asignación de registros llevada a cabo por el compilador. Para ello se estudiarán comparativamente diversos algoritmos de asignación empleando el mismo entorno experimental. El segundo objetivo, cuyo enfoque es hardware, consiste en diseñar un mecanismo de filtrado para reducir los accesos a la región de memoria de pila, originados por el desalojo de datos ubicados en registros y el paso de parámetros a funciones. La finalidad de este filtro es reducir el consumo del primer nivel de cache, una de las principales fuentes de

disipación de energía del procesador.

1.5. Organización del resto del documento

A continuación se presenta la organización del resto de la presente memoria de proyecto.

- Este Capítulo presenta una breve introducción, junto con la motivación y los objetivos de este trabajo.
- El Capítulo 2 contiene una recapitulación histórica de los orígenes de la jerarquía de memoria, su funcionamiento y los problemas actuales que presenta.
- En el Capítulo 3 se aborda el primer objetivo del proyecto, que consiste en la evaluación del impacto del algoritmo de asignación de registros en la jerarquía de memoria.
- En el Capítulo 4 se describe y evalúa experimentalmente el mecanismo de filtrado de los accesos a pila, que constituye el segundo objetivo del presente trabajo.
- En el Capítulo 5 se recapitulan los principales trabajos relacionados, en especial aquellos que comparten nuestro mismo propósito, el de reducir el tráfico de memoria originado tanto por el desalojo de registros como por el paso de parámetros a funciones.
- Finalmente, en el Capítulo 6 se exponen las principales conclusiones y aportaciones de este trabajo.

Capítulo 2

La jerarquía de memoria

Cuando se empezaron a fabricar los primeros computadores, el tiempo de acceso a las estructuras de memoria era suficientemente pequeño como para realizarse en un ciclo. Pero con la evolución de la tecnología el tiempo de ciclo de CPU disminuyó más rápido que el tiempo de acceso a memoria originando un problema, el *memory gap*, que aún perdura en los actuales computadores. La principal solución a este problema ha consistido y consiste, en emplear una jerarquía de varios niveles de memoria.

Este capítulo tiene por propósito introducir algunos conceptos básicos relacionados con esta jerarquía, como la memoria *cache*. En primer lugar, comenzaremos con una reseña histórica sobre el IBM System/360 Model 85, precursor de este tipo de memoria, y repasaremos brevemente el funcionamiento de la memoria cache. A continuación describiremos superficialmente la jerarquía de memoria en los procesadores actuales. Finalmente, nos detendremos en el banco de registros incidiendo en su papel e interacción con el resto de la jerarquía de memoria.

2.1. System/360 Model 85: La primera jerarquía de memoria

En 1967 IBM empezó a diseñar el computador System/360 Model 85 con el propósito de crear una máquina compatible con la familia System/360 con un alto rendimiento y una alta productividad. Uno de los requisitos de los computadores de alta productividad es una memoria principal de tamaño elevado. Sin embargo, con la tecnología del momento no era posible fabricar una memoria grande con tiempo de acceso menor al tiempo de ciclo del Model 85, que era de 80 nanosegundos. Un tiempo de acceso a memoria mayor que el ciclo de procesador podría verse compensardo parcialmente incrementando el solapamiento de accesos, incrementando el número de bancos en los que la memoria está dividida, contando con un manejo de saltos más sofisticado u otras mejoras. No obstante, estas soluciones no parecían suficientes y los diseñadores del Model 85 decidieron usar un sistema jerárquico de almacenamiento.

Esta jerarquía contaba con una memoria principal con un tiempo de acceso de 1.04 microsegundos (13 ciclos), y con una memoria pequeña y rápida, a la que bautizaron como cache, integrada en la propia CPU. La cache no podía ser direccionada por el programa, si no que se usaba para mantener los contenidos de la porción de memoria principal que estaba siendo usada en cada momento. De esa manera la mayoría de las peticiones del procesador a memoria tenían tiempo de acceso reducido. Cuando el programa comenzaba a operar en datos de otra región de memoria, los datos de esa nueva región tenían que ser cargados, y los datos de otra región desalojados. Esta actividad tenía que llevarse a cabo sin la ayuda del software, ya que el Model 85 tenía que ser compatible a

nivel de binario con el resto de computadores de la familia System/360. J. S. Liptay [Lip68] fue el responsable de esta idea y el pionero de lo que hoy día conocemos como jerarquía de memoria.

2.2. Funcionamiento de una cache

En sistemas como el introducido en el System/360 Model 85, cuando el procesador necesita leer o escribir un dato en memoria, primero consulta la cache, a ver si tiene una copia de ese dato. Esto se hace comparando la dirección del dato en memoria con todas las etiquetas de los bloques de cache que pudieran contener ese dato. Las etiquetas citadas son un subconjunto de los bits más significativos de la dirección. Si existe una copia en cache, se dice que ha habido un acierto, en caso contrario se habla de un fallo. En el caso de acierto, el procesador lee o escribe inmediatamente el dato en cache.

En caso fallo, habitualmente se crea una nueva entrada en la cache, compuesta por la etiqueta de la dirección que acaba de fallar, y una copia del correspondiente bloque de memoria. A continuación la petición se sirve como en el caso de que hubiera sido un acierto. Los fallos de cache son notablemente más lentos que los aciertos, ya que requieren que un bloque entero sea transferido desde memoria o desde el siguiente nivel de la jerarquía en caso que se esté empleando más de un un nivel de memoria cache. Reducir la tasa de fallos resulta por lo tanto crucial para el rendimiento del computador, ya que esta transferencia es mucho más lenta.

Los fallos en la cache pueden deberse a 3 causas. La primera es que el dato nunca haya sido referenciado y sólo se encuentra en memoria, *fallos iniciales*.

El segundo motivo de fallo es el tamaño limitado de la cache. Una cache comienza vacía, pero eventualmente llega un momento en el que se llena, y para traer el siguiente bloque referenciado hay que desplazar otro. Cualquier referencia posterior al bloque desplazado generará un fallo que podría haberse evitado si la cache fuese más grande, *fallos de capacidad*. El tercer y último tipo de fallos es originado por las restricciones en el emplazamiento. Cuando hay que crear una nueva entrada y hay bloques libres en la cache, pero el bloque a emplazar no puede colocarse en dichos bloques y tiene que desplazar otro bloque, entonces cualquier referencia al bloque desplazado dará lugar a un fallo que podría haberse evitado si la cache fuese menos restrictiva, *fallos de conflicto*.

Reducir los fallos por conflicto es uno de los primeros objetivos que debe abordarse para mejorar el rendimiento de la cache y por extensión de la jerarquía de memoria en su conjunto. A este respecto, las políticas de emplazamiento y remplazo juegan un papel esencial.

Para colocar un bloque en cache cuando es leído de memoria, la cache tiene que decidir en cual o cuales de las entradas puede ir emplazado. Esta decisión viene determinada por la política de emplazamiento. La política menos restrictiva es la *totalmente asociativa*, que no impone ninguna restricción. Cualquier bloque puede ser emplazado en cualquier entrada. La política opuesta es el *emplazamiento directo*, en la que cada bloque puede ir en una única entrada que viene determinada por los bits menos significativos de su dirección. La primera política otorga mucha libertad, a cambio de una mayor latencia y un mayor consumo de energía. La segunda política reduce drásticamente el consumo y la latencia, a cambio de incrementar el número de fallos

por conflicto. Existe una solución de compromiso que es la política *asociativa por conjuntos*, que divide la cache en c conjuntos con e entradas cada uno. La elección de conjunto se hace mediante emplazamiento directo, pero dentro de cada conjunto la selección de la entrada se hace de manera asociativa. Esta solución aporta parte de la flexibilidad de una cache totalmente asociativa con la latencia reducida de una cache de emplazamiento directo.

Para hacer sitio a las nuevas entradas en caso de fallos de cache, la cache normalmente tiene que *desplazar* (eliminar) una de las entradas existentes. La heurística que guía este proceso recibe el nombre de política de reemplazo y su objetivo es predecir cual de los bloques candidatos a a ser reemplazado es el que menos probabilidad tiene de ser usado en un futuro próximo. Predecir el futuro es difícil, especialmente si tratamos de obtener un circuito de complejidad reducida, por lo que normalmente se usan heurísticas que no son perfectas pero dan buenos resultados. Una política popular es *LRU, least recently used*, que reemplaza la entrada que menos recientemente ha sido usada.

Reducir el número de accesos a los niveles inferiores de la jerarquía de memoria es otro factor clave para mejorar el rendimiento y la eficiencia (consumo de energía). Cuando se escriben datos en a cache, en algún momento se debe escribir también en el nivel inferior. El momento de esta segunda escritura viene determinado por lo que se conoce como *p*olítica de escritura. En una cache *w*rite-through cada escritura en cache produce una escritura en el nivel inferior de la jerarquía. Alternativamente, una cache *w*rite-back lo cambios no se reflejan inmediatamente en el nivel inferior. La cache memoriza qué entradas han sido modificadas. Los datos de estas entradas se escriben en el nivel inferior cuando la entrada es desplazada de la cache. Por este motivo los

fallos en escritura de una cache *write-back* a menudo requieren dos accesos al nivel inferior: uno para escribir el bloque modificado (sucio) que se está desplazando, y otro para leer los nuevos datos.

Durante los 30 últimos años han sido numerosos investigadores los que han dedicado esfuerzos a mejorar las caches en múltiples aspectos, principalmente en reducir la latencia, los fallos y el consumo por acceso. Revisar todas y cada una de estas mejoras es un labor que excede con mucho el ámbito de este trabajo. Es preciso mencionar que, hoy día, las alternativas generales de comportamiento de una cache no difieren substancialmente de lo expuesto aquí.

2.3. La jerarquía de memoria en los procesadores actuales

Esta organización jerárquica de la memoria permite que un procesador cuente con una gran cantidad de almacenamiento con el precio y el tamaño de una tecnología lenta y barata, pero con el tiempo de acceso de la tecnología más rápida y cara. Por este motivo, casi todos los computadores actuales cuentan con una organización jerárquica de la memoria. Sólo los sistemas creados para computación en tiempo real prescinden de esta organización jerárquica debido a la dificultad que plantea predecir su latencia de acceso.

Cualquier procesador de propósito general cuenta con dos, o hasta tres niveles de cache integrados en el chip. Intel, por ejemplo, distribuye los procesadores de la gama Core 2 Quad con hasta 12 MB de cache L2.

En estos sistemas con varios niveles de cache, cada nivel sólo tiene acceso al nivel inmediatamente inferior, y es accedido únicamente por el nivel superior. Es decir, en un sistema con dos niveles de cache, la cache L2 contendrá bloques de datos traídos de memoria, y la L1 operará con bloques más pequeños que leerá del la cache L2.

2.4. El banco de registros y la jerarquía de memoria

A mediados de los años 70 varios investigadores en distintos proyectos demostraron que la mayoría de las combinaciones de los modos de direccionamiento y las instrucciones no eran usadas en los códigos generados por los compiladores.

También se dieron cuenta de que en las implementaciones micro-codificadas de ciertas arquitecturas las operaciones complejas tardaban más en ejecutarse que secuencias de instrucciones más simples que hicieran la misma operación. Esto se debía en parte a que muchos de los diseños eran precipitados y no había habido tiempo para optimizar las operaciones, excepto las más frecuentes. El ejemplo más sonado fue la instrucción INDEX de VAX que se ejecutaba más lenta que un bucle implementando el mismo código.

Estas dos razones junto con el hecho de que la memoria era cada vez más lenta en relación con la CPU, crearon un nuevo tipo de arquitectura, más sencilla, sin las operaciones tan complejas y con el requisito de que los operandos deben residir en los registros. El área ahorrada se invirtió en una ampliación

del banco de registros, ya que en ese tipo de arquitectura las operaciones que no fuesen load o store sólo podían tomar sus operandos del banco de registros o ser inmediatos codificados en la instrucción. Por eso estas arquitecturas conocidas como RISC también son referidas con el nombre de arquitecturas de carga-almacenamiento.

Estas arquitecturas convierten, en cierta manera, el banco de registros en la parte superior de la jerarquía de memoria, ya que las instrucciones que en un principio operaban con posiciones de memoria, en este modelo tienen que llevar los operandos a un nivel más arriba (registros) operar y luego hacer una escritura en memoria del dato con una operación de almacenamiento.

Por lo tanto, en el contexto de este tipo de arquitecturas, que es en el que se enmarca este trabajo, la gestión banco de registros tiene importantes repercusiones en el resto de la jerarquía de memoria. Para empezar, cuantos más resultados intermedios se alojen en registros menos serán los accesos a memoria. Una solución obvia para reducir estos accesos consistiría en incrementar el número de registros arquitectónicos. Sin embargo, esta alternativa no resulta atractiva ya que implica incrementar el tamaño de las instrucciones y del código.

El reto que nos planteamos en este trabajo consiste en reducir los accesos a memoria evitando toda modificación del repertorio de instrucciones. Para lograrlo, contemplamos dos estrategias distintas: actuar a nivel de la asignación de registros, e introducir un nivel intermedio en la jerarquía que actúe a modo de filtro. En los próximos capítulos mostraremos los resultados obtenidos por cada una de ellas.

Capítulo 3

Asignación de registros

El proceso de compilación tiene una primera fase de análisis sintáctico del código, en la que se genera una representación intermedia del programa. A continuación se aplican pasos de optimización y se realiza la planificación de instrucciones. Hasta este punto, el compilador trabaja con registros virtuales (variables) para almacenar los valores y las operaciones intermedias. El número de registros virtuales con los que puede trabajar un compilador es, potencialmente, infinito. Tras la asignación de registros, el compilador trabajará con posiciones de memoria y registros arquitectónicos, cuyo número y tipo viene determinado por la arquitectura objetivo. Por último se produce la generación de código y el ensamblado del binario final. El proceso que para cada variable decide en qué lugar (memoria o registros) estará alojada es lo que se conoce como asignación de registros.

Desde el desarrollo de los lenguajes de alto nivel y los compiladores han sido varios los autores que han desarrollado diferentes algoritmos para llevar a cabo esta tarea. Han creado o mejorado algoritmos para reducir la cantidad de

variables que se alojan en memoria, o el número de transacciones entre memoria y registros. Sorprendentemente, los efectos de la asignación de registros en la jerarquía de memoria (comportamiento, consumo de energía, etc.) no han sido tenidos en consideración. Nuestro objetivo es aportar algo de información al respecto, al menos para la plataforma objetivo considerada.

En este capítulo se presenta un estudio experimental de diversos algoritmos de asignación de registros. Su estructura es la siguiente. En primer lugar, se describen las características de los algoritmos de asignación de registros (AAR) y se presentan los cuatro algoritmos que son objeto de estudio. A continuación, se describe el entorno experimental y se analizan comparativamente sus resultados. Por último, se esbozan las principales conclusiones que pueden derivarse de este estudio.

3.1. Características de los Algoritmos de Asignación de Registros

Los algoritmos de asignación deben maximizar el uso de los registros, que son el nivel con menor latencia y menor consumo energético de la jerarquía, por ello tienen que intentar mantener el mayor número de variables en ellos.

Cada vez que se define (asigna) una variable, si no hay ningún registro libre, hay que elegir uno de los ocupados y desalojar la variable almacenada, guardandola en memoria. Esto se conoce como desbordamiento o *spilling*. Los algoritmos de asignación de registros también deben intentar minimizar este desbordamiento, por varios motivos. El primero es el rendimiento. Cuando se

produce un desbordamiento hay que hacer varios (al menos dos) accesos a memoria, que en el mejor de los casos tienen la misma latencia que una operación de enteros, pero si se produce un fallo de cache la latencia será superior. Otro motivo es el consumo energético. Los accesos a memoria son costosos en términos de consumo de potencia, más costosos cuanto más bajamos en la jerarquía. Por último, otro factor a tener en cuenta, especialmente en el contexto de los sistemas empotrados, es el tamaño del binario. Cuanto más numerosas sean las transferencias de datos entre registros y memoria, mayor será el tamaño del código.

Una de las características de los algoritmos de asignación que está ganado importancia, es el tiempo de ejecución. Habitualmente la compilación es un proceso que suele llevarse a cabo una vez, previamente a la ejecución, *offline*, pero en los últimos años a florecido la tendencia a la compilación en tiempo de ejecución o *Just-in-Time compiling (JIT)*. Este proceso ocurre durante la ejecución de códigos interpretados. La máquina virtual determina cuando un fragmento de código es ejecutado un numero suficientemente elevado de veces como para que sea rentable compilarlo en tiempo de ejecución. Cuando detecta un fragmento de estas características lanza una compilación para que esa sección se ejecute como código nativo. En el caso de la compilación JIT el tiempo de compilación es crucial. No se trata de un proceso *offline* en el que se puedan invertir grandes cantidades de tiempo, si no que se trata de un proceso en el que el tiempo de compilación tiene que ser lo más rápido para poder sacar el máximo provecho a la ejecución del código nativo generado.

3.2. Algoritmos de Asignación de Registros

Con el algoritmo de asignación más simple el valor de una variable se carga desde memoria a un registro justo antes de cada uso, y se almacena de nuevo en memoria después de cada vez que se redefine. Este algoritmo es sencillo y rápido, pero produce un código de baja calidad, dado que añade al programa final muchos accesos a memoria. Lo cual hace que tenga bajo rendimiento, un elevado consumo energético y además el tamaño del binario generado sea elevado. Por esto necesitamos algoritmos más sofisticados que traten de reducir la cantidad de operaciones de carga/almacenamiento ejecutadas.

Hay dos tipos principales de algoritmos de asignación de registros. El primero surge como evolución y mejora de este algoritmo de asignación sencillo. El segundo enfoca el problema desde otra perspectiva, ya que consiste en plantear la asignación como un coloreado de grafos.

Escaneado Lineal: Los algoritmos de *escaneado lineal* se fundamentan en el hecho de que no es necesario recargar una variable si ya está en un registro. El programa se modela como una colección de secuencias de instrucciones ordenadas de manera lineal, también llamadas intervalos. El rango de vida de cada variable está representado por un intervalo. Una vez construida esta representación, el escaneado lineal asigna registros a variables aplicando un algoritmo voraz a la secuencia de intervalos. Este algoritmo no es óptimo por dos motivos. El primero es que flujo de control de datos del programa fuente no es, en general, una línea, y el segundo es que los rangos de vida de las variables normalmente contienen huecos entre el último uso de una definición y la siguiente redefinición, esto es, no son intervalos contiguos.

Coloreado de Grafos: Para superar las limitaciones de los algoritmos de escaneado lineal, los algoritmos basados en *coloreado de grafos* [Cha82] incluyen todas las variables del programa en un grafo de interferencias G, $G = (V, E)$. Cada nodo $p \in V$ del grafo representa una variable. Una arista $e \in E$ existe entre dos nodos, $p_i - p_j$, si los rangos de vida de p_i y p_j se solapan, y significa que ambas variables no pueden ser asignadas al mismo registro. Por lo tanto, si G puede ser coloreado con k colores diremos que G es *k-coloreable*. En ese caso todas la variables pueden ser asignadas a k registros sin interferencias. Cuando el grafo no puede ser coloreado, se determina heurísticamente la variable p_i que será desalojada de los registros, y se genera un nuevo grafo $G' = (V', E'), V' = V \setminus \{p_i\}, E' = \{(p_j, p_k) \in E | p_i \neq p_j \wedge p_i \neq p_k\}$ que será usado por el algoritmo en vez de G. Este procedimiento se repite hasta que el grafo puede ser coloreado, después el algoritmo vuelve a cargar las variables desalojadas y las almacena cuando es necesario. Estos algoritmos consiguen un código de mejor calidad, ya que reducen considerablemente la cantidad de Código de Desalojo. La desventaja de estos algoritmos es que su complejidad temporal es bastante más elevada.

Para este trabajo he elegido 4 algoritmos de asignación de registros representativos para su evaluación. Dos de ellos pertenecen a la familia de algoritmos de escaneado lineal: Escaneado Lineal, Asignación Basada en el Contador de Uso. Los otros dos pertenecen a la familia de coloreado de grafos: Fusión de Registros Iterada y Coloreado de Grafos Cordales.

3.2.1. Escaneado Lineal (LS)

Este algoritmo fue propuesto por Polleto y Sarkar [PS99] es popular por su velocidad. El algoritmo asume una representación lineal del programa de entrada. El funcionamiento del algoritmo es el siguiente: al principio de cada nuevo intervalo el algoritmo comprueba si el número de intervalos vivos es menor que que la cantidad de registros disponibles. Si es así, entonces asigna a cada rango alguno de los registros disponibles. En caso de que el número de intervalos vivos sea mayor que la cantidad de registros se desaloja uno de los rangos de vida para liberar un registro, es decir, se introduce una instrucción de almacenamiento, y entonces se asigna el registro liberado al nuevo rango de vida.

El conjunto de intervalos desalojados se almacena como un conjunto de pares variable-instrucción (p, i) que denotan que la variable p está viva en la instrucción i pero que ha sido desalojada, esto es, reside en memoria. Cada vez que es necesario desalojar una variable, la variable candidata para el desalojo es aquella cuyo último uso es el más lejano, es decir, se favorece la localidad espacial. Para cada intervalo desalojado (p, i), si p es necesaria en i, debemos volver a cargarla en algún registro antes de su uso. Análogamente, si i define p, entonces debemos escribir p en un registro y desalojarlo a memoria, después de i.

3.2.2. Basado en Contador de Uso (UCB)

Este algoritmo fue propuesto por Freiburghouse [Fre74] es otro algoritmo de escaneado lineal. La información del contador de uso guía las decisiones de

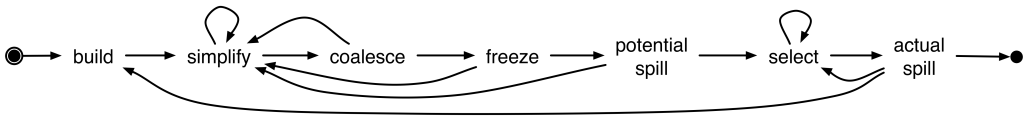

FIGURA 3.1: Fusión de Registros Iterada.

desalojo o spilling. Una variable puede ser desalojada sin necesitar ser recargada cuando su contador de uso llega a cero. Para llevar a cabo la asignación de registros usando contadores de uso, el algoritmo mantiene una tabla con las asignaciones de variables a registros en cada punto del programa. Además, es necesario hacer una primera pasada por el código para realizar una estimación del contador de uso.

En la segunda pasada, el algoritmo asigna el contador de uso máximo a cada variable cuando ésta es definida. Mientras el algoritmo examina instrucciones actualiza la tabla de asignación y va decrementando el contador de uso de las variables cada vez que son referenciadas. Cuando una variable tiene su contador de uso reducido a cero el registro al que está asignada es liberado, y puede ser usado para alojar en él a cualquier otra variable. Si, a la hora de de definir o cagar una variable, no hay registros libres, entonces hay que desalojar una variable para liberar un registro. La variable cuyo contador de uso sea el menor entre las variables que tengan un registro asignado es el candidato para el desalojo. Las instrucciones posteriores al desalojo ven a esta variable como *no disponible*. Las variables no disponibles deben ser recargadas antes de su siguiente uso, desalojando, si es necesario, otra variable para ser recargadas.

3.2.3. Fusión de Registros Iterada, IRC

Este es un algoritmo de asignación basado en coloreado de grafos, propuesto en [GA96]. Los autores mejoran el método original de Chaitin y Briggs añadiendo fusión entre los nodos del grafo de interferencia. La fusión consiste en asignar el mismo registro cuando es posible a variables que esté relacionadas por una operación de movimiento, reduciendo así la complejidad del grafo de interferencia. También se reduce el número de instrucciones de movimiento, y por tanto, el tiempo de ejecución. La fusión se aplica de manera conservativa, sólo se aplica si el nodo resultante tiene grado menor que k, siendo k el número de registros disponibles. El objetivo del algoritmo es identificar tantas oportunidades como sea posible para fusionar nodos del grafo, vinculando las variables fusionadas al mismo registro. Esto además de eliminar instrucciones de movimiento, reduce la presión sobre los registros. El proceso de asignación mostrado en la Figura 3.2, incluye las 5 fases principales sobre las que el algoritmo itera de manera selectiva.

1. Construcción: En esta fase el algoritmo construye el grafo de interferencia e identifica los operandos que participan en instrucciones de movimiento. A los nodos que pertenecen a variables involucradas en instrucciones de movimiento se les pone una marca de *implicado-en-movimiento*.

2. Simplificación: En este paso se realizan modificaciones al grafo de interferencia. Se elimina un nodo, que puede corresponder a una variable o a varias fusionadas, que tenga grado menor que k y que no esté marcado como implicado-en-movimiento. De esta manera damos más importancia a las variables que tienen mayor número de interferencias, que es

proporcional a la vida de la variable.

3. Fusión: Cuando no es posible simplificar más el grafo se realiza la fusión de nodos marcados como implicado-en-movimiento de una manera conservadora. A continuación se repiten los pasos 2 y 3 hasta que se obtiene un grafo en el que cada nodo, o bien tiene un grado mayor que el número de registros disponibles, o bien es parte de una instrucción de movimiento.

4. Bloqueo (*Freeze*) + desalojo potencial: Si los pasos 2 y 3 no pueden seguir siendo aplicados, entonces eliminamos la marca de implicado-en-movimiento de algún nodo de grado bajo, y volvemos al paso 2.

5. Selección + desalojo real: En esta última fase el algoritmo intenta asignar colores a las variables del grafo en orden inverso al de simplificación. Si el grafo no es k-coloreable donde k es el número de registros disponibles, entones se elige una variable que será desalojada y se vuelve al paso 1, comprobando si el desalojo realizado da como resultado un grafo k-coloreable.

Aunque teóricamente el problema es NP-completo como se demuestra en [BDR07], en la práctica itera pocas veces y por lo tanto el tiempo de ejecución medio está muy por debajo del máximo teórico.

3.2.4. Coloreado de Grafos Cordales (CGC)

Propuesto en [PP05], también es un algoritmo iterativo. El algoritmo funciona tanto con grafos cordales como con no-cordales. Un grafo cordal o trian-

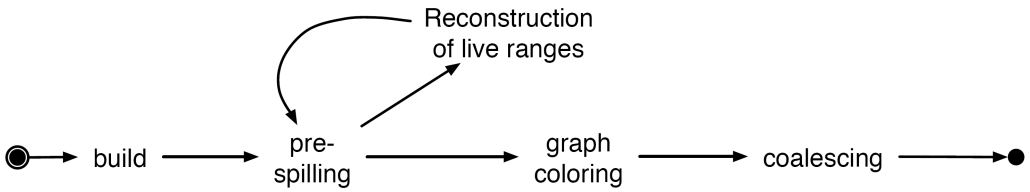

FIGURA 3.2: Fusión de Registros Iterada.

gulado es aquel que cumple la propiedad de cordalidad, que consiste en que todo ciclo de 4 o más aristas de longitud posee una arista entre dos nodos no consecutivos en el ciclo, es decir, una cuerda. Este algoritmo, cuando el grafo de interferencia es cordal, puede encontrar una asignación óptima de registros si existe. Pereira y Pallsberg trabajan con el compilador JoeQ para java. En su trabajo muestran cómo la mayoría (91 %) de los grafos de las funciones de la librería estándar de Java 1.5 son cordales. Además muestran cómo la mayoría de los grafos necesitan 8 colores o menos para colorearse.

El algoritmo itera por las siguientes cuatro fases:

1. Desalojo: El algoritmo busca potenciales desalojos antes de la fase de coloreado. Para minimizar el número de desalojos, el algoritmo intenta eliminar nodos que son parte de varios cliques. Si se ejecuta la fase de desalojo, entonces será necesario reconstruir los rangos de vida de las variables. Es posible demostrar que después de la fase de desalojo no serán necesarios más desalojos.

2. Reconstrucción de los rangos de vida: El algoritmo ejecuta esta fase sólo si es necesario. En ella se reconstruyen los rangos de vida de las variables desalojadas y se reconstruye el grafo de interferencia adaptado a esos rangos de vida.

3. Coloreado: El algoritmo intenta colorear el grafo. Un grafo cordal $G = (V, E)$ puede ser óptimamente coloreado en un tiempo $O(|V| + |E|)$. Lo que está por debajo de la complejidad de IRC.

4. Fusión: El último paso es la fusión de instrucciones de movimiento. La fusión se realiza de manera voraz: por cada par de nodos que intervengan en un movimiento el algoritmo intenta asignarles el mismo color.

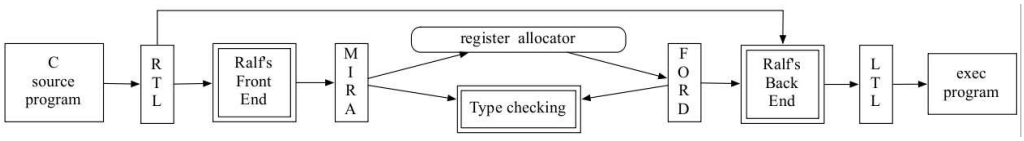

FIGURA 3.3: Diagrama de Flujo de RALF.

3.3. Entorno Experimental

En esta sección se describen las herramientas que componen el entorno experimental que hemos utilizado: el compilador usado, el simulador empleado para llevar a cabo la evaluación de los programas generados y los benchmarks usados en nuestras mediciones.

3.3.1. Simulador

El simulador usado es Sim-PAnalyzer [KAMG02], que es un simulador de consumo construido sobre la utilidad de simulación SimpleScalar/ARM [ALE02]. El simulador está configurado para modelar un procesador SA-110 StrongARM. Los principales parámetros de configuración estan recogidos en la Tabla 3.1. Como parámetros de consumo de energía se han empleado los suministrados con la propia herramienta.

3.3.2. Compilador

La infraestructura de compilación se basa en RALF [NPP07] (*Register ALlocation Framework*). Este entorno de trabajo, construido sobre gcc-2.95.2 para ARM, permite a los investigadores usar un nuevo algoritmo de asignación de registros sin necesidad de que tengan conocimientos de los detalles de

implementación de gcc. El funcionamiento de RALF podemos verlo esbozado en la Figura 3.3. RALF usa el front-end de gcc para crear el código RTL (*Register Transference Language*) del programa, ejecuta los pasos de optimización pertinentes y, a continuación, escribe en un archivo la información necesaria para el algoritmo de asignación descrita usando un lenguaje específico llamado MIRA (*Mathematical Intermediate representation for Register Allocation*). A continuación se ejecuta un programa externo que es el encargado de realizar la asignación de registros. Este programa es proporcionado por el investigador, y tiene que leer el archivo de entrada en formato MIRA, realizar la asignación y escribir el resultado en un archivo que siga el formato FORD (*FOrmat for Register allocation Directives*), que es un lenguaje específico para comunicarle a RALF las decisiones de asignación. A continuación RALF toma este archivo FORD como entrada para transformar el código RTL, y posteriormente el *back-end* de gcc lleva a cabo la generación de código. La asignación de registros se realiza localmente a cada función. El archivo MIRA contiene, entre otros datos, información sobre cuales son los registros arquitectónicos disponibles, cuales son guardados por el proceso invocador y cuales por el invocado, cual es el coste de las operaciones de carga y almacenamiento e información sobre la vida de las variables.

3.3.3. Benchmarks

Los benchmarks empleados para la evaluación de los algoritmos de asignación son: *dijkstra* y *patricia*, que han sido extraídos de MiBench [GRE+01], y *adpcm*, *g721*, *mesa* y *mpeg2dec*, de MediaBench [LPMS97]. Todos los bench-

marks han sido compilados usando los algoritmos de asignación descritos en la Sección 3.2 con las opciones de optimización estándar "-O2 -fshort-double". Las simulaciones se han llevado a cabo con las cargas de trabajo de referencia distribuidas con los propios benchmarks.

TABLA 3.1: Parámetros de simulación

Característica	**Descripción**	
clk	200Mhz	
tipo de predictor	no tomado	
tamaño de la pila de direcciones de retorno	8	
branch target buffer	vías	512
	entradas	4
tamaño de la cola IF	2	
relación de velocidad front-end/núcleo	1	
$il1	vías	16
	entradas	32
	tamaño de bloque	32 bytes
	reemplazamiento	FIFO
itlb	vías	32
	entradas	32
anchura de decode,	2	
tamaño de RUU	2	
anchura de issue		
tipo de issue	en orden	
unidades funcionales	ALU de enteros	2
	multiplicación de enteros	1
	ALU de punto flotante	2
	multiplicación de punto flotante	1
	puertos de memoria	1
$dl1	vías	16
	entradas	32
	tamaño de bloque	32 bytes
	reemplazamiento	FIFO
dtlb	vías	32
	entradas	32
tamaño de LSQ	2	
anchura de commit		
latencia de $1 cache	1 ciclo	
latencia de $2 cache	6 ciclos	
latencia de memoria	1^{er} bloque	10 ciclos
	interbloque	1 ciclo
anchura del bus con memoria	4 bytes	
tamaño de página	4096	

3.4. Resultados

A continuación expondremos y analizaremos los resultados obtenidos de la evaluación de los algoritmos de asignación descritos en la Sección 3.2 fijándonos en distintas métricas relevantes para la plataforma objetivo. Dado que LS es el algoritmo más sencillo de los que se consideran, todos los resultados de las tablas han sido normalizados con respecto a este para facilitar el análisis. La media utilizada es la media geométrica, ya que como muestran en [HP06] cuando trabajamos con datos normalizados la media geométrica tiene mejor comportamiento en cuanto a consistencia e independencia del valor de referencia.

El AAR empleado por defecto en el compilador *gcc* no se ejecuta como un programa externo invocado por RALF. Por lo tanto, al no ejecutarse en las mismas condiciones que el resto de algoritmos se incluye únicamente como referencia pero no se incluye en el análisis comparativo. Además, dispone de información detalla sobre la arquitectura objetivo y la usa para hacer una asignación de registros más eficiente, por lo que cuenta con ventaja frente los demás algoritmos. A su favor cuenta también con ciertas técnicas complementarias como el *life range splitting* que consiste en partir el rango de vida de una variable en distintos tramos, de esta manera se hace un análisis por rangos de vida en vez de por variables, lo que facilita el proceso de asignación, y permite que un algoritmo sencillo consiga buenos resultados.

TABLA 3.2: Tiempo de ejecución, normalizado a LS

AA	adpcm	dijkstra	g721	mesa	mpeg2	patricia	media
LS	1	1	1	1	1	1	1
CGC	3.967	1.845	2.831	1.690	2.353	4.191	2.649
IRC	2.890	1.924	2.726	1.834	2.105	2.130	2.234
UCB	2.866	1.790	2.137	1.503	1.735	2.474	2.034

3.4.1. Tiempo de ejecución de la asignación de registros

La Tabla 3.2 recoge los tiempos de ejecución relativos de los algoritmos estudiados. Como cabe esperar, cuanto más simple es el algoritmo, menor es su tiempo de ejecución. LS se ejecuta, en media, en menos de la mitad de tiempo que cualquier otro de los algoritmos considerados. UCB, el otro algorimo de escaneado lineal, presenta también mejores tiempos que los algoritmos de coloreado de grafos. Un hecho sorprendente es que, en nuestro entorno experimental, IRC aventaja a CGC, lo que contradice los resultados publicados en [PP05] y pone en cuestión los beneficios de este tipo de algoritmo de asignación.

Cabe señalar que, en términos generales UCB, IRC y CGC no difieren demasiado entre sí. Por lo tanto, si uno de ellos fuese apto para JIT, el resto también lo serían.

3.4.2. Instrucciones ejecutadas

En la Tabla 3.3 podemos comprobar como la rapidez de ejecución de la compilación utilizando LS produce un código más pobre en términos del número de instrucciones ejecutadas por el procesador. Los algoritmos más sofisticados consiguen una reducción en torno al 20 % respecto a LS en el número de instrucciones ejecutadas. Esto, como tendremos ocasión de comprobar a

TABLA 3.3: Número de instrucciones ejecutadas, normalizado a LS

AA	adpcm		dijkstra	g721		mesa	mpeg2	patricia	media
	dec	enc		dec	enc		dec		
LS	1	1	1	1	1	1	1	1	1
CGC	0.716	0.724	0.707	0.791	0.793	1.000	0.973	0.999	0.828
IRC	0.600	0.622	0.689	0.781	0.785	0.999	0.996	1.001	0.793
UCB	0.623	0.622	0.708	0.781	0.784	1.000	1	0.999	0.800
gcc	0.656	0.622	0.687	0.773	0.778	0.999	0.916	0.997	0.791

continuación, reduce el consumo de energía y tiempo de ejecución. Por lo demás, las diferencias entre CGC, IRC, UCB y gcc no son muy significativas. En la Tabla 3.4 podemos ver cómo el número de instrucciones ejecutadas está estrechamente relacionado con el tiempo de ejecución de los códigos generados. De nuevo CGC, IRC y UCB se distingen mucho de LS, pero entre sí son bastante similares.

TABLA 3.4: Tiempo de ejecución del código generado, normalizado a LS

AA	adpcm		dijkstra	g721		mesa	mpeg2	patricia	media
	dec	enc		dec	enc		dec		
LS	1	1	1	1	1	1	1	1	1
CGC	0.605	0.643	0.639	0.739	0.739	0.999	0.954	0.989	0.771
IRC	0.465	0.487	0.624	0.697	0.709	1.001	0.938	1.019	0.713
UCB	0.499	0.474	0.647	0.728	0.733	0.999	1	1.012	0.732
GCC	0.529	0.474	0.623	0.678	0.682	1.000	0.684	0.975	0.684

3.4.3. Referencias a memoria

TABLA 3.5: Número de referencias a memoria, normalizado a LS

AA	adpcm		dijkstra	g721		mesa	mpeg2	patricia	media
	dec	enc		dec	enc		dec		
LS	1	1	1	1	1	1	1	1	1
CGC	0.437	0.461	0.494	0.579	0.582	1.002	0.948	1.001	0.650
IRC	0.192	0.206	0.461	0.519	0.527	0.988	0.926	1.006	0.510
UCB	0.238	0.163	0.494	0.561	0.569	1.001	1	1.001	0.529
gcc	0.263	0.163	0.455	0.468	0.472	0.999	0.834	0.994	0.494

Pasemos a analizar la presión que ejerce cada algoritmo de asignación sobre la jerarquía de memoria. La tabla 3.5 muestra los accesos a memoria para cada uno de los algoritmos considerados. Se aprecia claramente que, en 5 de los 8 benchmarks hay una gran reducción en el los accesos a memoria (50 % o más), respecto al algoritmo más sencillo, LS. Como cabía esperar, *gcc* aparte, el algoritmo que más reduce la presión sobre la jerarquía de memoria es IRC, seguido de cerca por UCB. CGC obtiene unos resultados más pobres, especialmente para *adpcm*.

Las diferencias observadas entre los distintos algoritmos se deben a dos factores: el número de variables desalojadas, y el coste de desalojo de cada variables (accesos a memoria inducidos). Es preciso señalar que, tal y como ilustraremos más adelante, incluso con el mismo número de variables desalojadas se pueden apreciar diferencias en la cantidad dinámica de cargas/almacenamientos. Dependiendo de la rama de control dónde se ubicase la variable desalojada, se ejecutan más o menos instancias de las instrucciones de carga/almacenamiento.

3.4.4. Fallos de cache

TABLA 3.6: Número de fallos de cache, normalizado a LS

AA	**adpcm**		**dijkstra**	**g721**		**mesa**	**mpeg2**	**patricia**	**media**
	dec	**enc**		**dec**	**enc**		**dec**		
LS	1	1	1	1	1	1	1	1	1
CGC	1.054	1.020	0.999	1.027	1.032	1.000	0.006	1.009	1.018
IRC	1.000	1.011	1.003	1.024	1.039	0.999	.010	1.022	1.014
UCB	1.057	0.997	1.001	1.024	1.030	0.999	1	1.012	1.015
GCC	1.093	0.994	0.975	0.995	1.002	0.999	1.002	1.007	1.008

Si bien el número de accesos a la jerarquía de memoria varía significativa-

mente de un algorimo a otro, el comportamiento de esta no se vé prácticamente afectado como ilustra la Tabla 3.6. El número de fallos de la cache de datos L1 no cambia significativamente. Esto se debe a lo siguiente, los registros desalojados se realojan en la pila, que es una estructura con una elevada localidad tanto temporal como espacial. Por lo tanto las cargas/almacenamientos debidos a desalojos no producen fallos de cache. Es más, estas operaciones incrementan de manera artificial la tasa de aciertos, sin mejorar ni el rendimiento, ni el consumo. Por esto no se incluyen medidas de tasas de aciertos y tasas de fallos, ya que podrían dar una impresión equivocada al lector. De hecho la tasa de aciertos más elevada la consigue LS, a pesar de que todos los algoritmos tienen un número similar de fallos de cache.

3.4.5. Consumo de potencia

TABLA 3.7: Potencia disipada, normalizada a LS

AA	**adpcm**		**dijkstra**	**g721**		**mesa**	**mpeg2**	**patricia**	**media**
	dec	**enc**		**dec**	**enc**		**dec**		
LS	1	1	1	1	1	1	1	1	1
CGC	0.651	0.665	0.676	0.770	0.771	1.000	0.981	0.995	0.801
IRC	0.511	0.512	0.663	0.742	0.748	0.999	0.985	1.007	0.746
UCB	0.544	0.504	0.685	0.767	0.772	1.000	1	1.013	0.760
gcc	0.571	0.504	0.659	0.732	0.736	0.999	0.899	0.987	0.741

La Tabla 3.7 muestra el consumo total de potencia por benchmark y algoritmo de asignación. Las diferencias que se observan entre los distintos algoritmos se deben a dos causas: las instrucciones ejecutadas y los accesos a memoria. Cuanto menor es el número de instrucciones ejecutadas menor es el tiempo de ejecución, pero también menor es el consumo de la cache de instrucciones ya que se realizan menos accesos a ella. Por su parte, la reducción

en el numero de accesos a memoria se traduce en un menor consumo de la cache de datos. Cabe señalar además que cuantas menos instrucciones y datos se lean de memoria, menos traducciones de direcciones hay que hacer y menor es el consumo de las TLBs correspondientes. Aunque sean estructuras de menor tamaño que las caches, su complejidad es elevada y su consumo no es despreciable.

Como puede apreciarse claramente, los resultados de consumo siguen la misma tónica señalada anteriormente: LS tiene un comportamiento más pobre que los demás algoritmos, y estos tan solo presentan ligeras diferencias entre ellos.

3.4.6. Tamaño del binario

Finalmente, fijándonos en el tamaño del binario generado, no hemos apreciado diferencias significativas entre los distintos algorimos de asignación (menos del 5 % en media con respecto a LS). Al menos para los códigos estudiados, el algoritmo de asignación no parece ser de gran ayuda al compilador en lo relativo a la reducción del tamaño del binario generado, ya que la cantidad de intrucciones estáticas es reducida.

3.4.7. ADPCM

Para ilustrar las diferencias entre distintos algoritmos de asignación vamos a fijarnos detenidamente en el programa de decodificación de ADPCM, adpcmdec. La función principal contiene un bucle extenso con un flujo de control complejo como puede verse en el Algoritmo 3.4.7. Este bucle ocupa la

mayor parte del tiempo de ejecución. En la Tabla 3.8 se pude observar, para este bucle, el número de variables desalojadas y la cantidad de operaciones de carga y almacenamiento introducidas como código de desalojo.

Como se puede ver, el algoritmo LS no es capaz de tratar eficientemente el flujo de control complejo del bucle, y genera una cantidad excesiva de código de desalojo. El resto de algoritmos apenas difiere en cuanto al número de operaciones de carga/almacenamiento estáticas. De hecho, la cantidad de variables desalojadas es la misma. Sin embargo, esto no tiene los mismos costes en número de accesos a memoria, como muestra la Tabla 3.5. Diferentes variables desalojadas conllevan la inclusión de operaciones de carga/almacenamiento en distintos caminos del grafo de flujo de control, produciendo una cantidad distinta de accesos a memoria dinámicos.

TABLA 3.8: Características del código de desalojo de adpcmdec.

AA	**#variables desalojadas**	**código de spill**	
		#cargas	**#almacenamientos**
LS	9	34	14
CGC	3	4	1
IRC	3	3	1
UCB	3	3	1

Algorithm 1 Código del bucle interno de adpcmdec

```
outp=outdata;
inp=(signed char *)indata;

valpred=state->valprev;
index=state->index;
step=stepsizeTable[index];

bufferstep=0;

for (;len>0;len--){
        if(bufferstep){
                delta=inputbuffer & 0xf;
        }else{
                inputbuffer=*inp++;
                delta=(inputbuffer>>4) & 0xf;
        }
        bufferstep=!bufferstep;

        index+=indexTable[delta];
        if(index<0) index=0;
        if(index>88) index=88;

        sign=delta & 8;
        delta=delta & 7;

        vpdiff=step>>3;
        if(delta&4) vpdiff+=step;
        if(delta&2) vpdiff+=step>>1;
        if(delta&1) vpdiff+=step>>2;

        if(sign)
                valpred-=vpdiff;
        else
                valpred+=vpdiff;

        if(valpred>32767)
                valpred=32767;
        else if(valpred<-32768)
                valpred=-32768;

        step=stepsizeTable[index];

        *outp++=valpred;
        }

state->valprev=valpred;
state->index=index;
```

3.5. Conclusiones

El estudio comparativo de los algoritmos de asignación que hemos llevado a cabo nos permite extraer, para la plataforma objetivo considerada, las siguientes conclusiones:

- Como cabía esperar, el principal efecto de los algorimos de asignación en la jerarquía de memoria tiene que ver con el número de accesos a la misma, ya sea para leer instrucciones o bien para leer/escribir datos.

- El comportamiento de la cache no se ve afectado por los algoritmos de asignación ya que las variables desalojadas de los registros se almacenan en la pila, y acceder a ellas rara vez produce fallo de cache.

- Excepto para el más simple, LS, se observan pocas diferencias entre los algorimos de asignación tanto en términos de consumo como de rendimiento. Por lo tanto, un sencillo algoritmo de escaneado lineal como UCB puede resultar suficiente tanto para compiladores *off-line* como *JIT*.

- El comportamiento de CGC con respecto a IRC difiere substancialmente de lo indicado por otros autores en el contexto de otro tipo de programas y computadores. Esto sugiere que los algorimos de asignación deben ser reevaluados en cada contexto particular, ya que tanto los estudios teóricos (los más frecuentes) como los que se efectuan para otras plataformas o cargas de trabajo pueden dar lugar a conclusiones erroneas o no generalizables a todos los ámbitos.

- Incorporar información de la arquitectura y/o diseñar algorimos más específicos sin duda alguna mejorararía substancialmente la asignación de registros. Sin embargo, esto objetivo excede el ámbito del presente trabajo y se he dejado como trabajo futuro.

Capítulo 4

Filtro de accesos a pila

En el capítulo anterior se analizó el impacto de los algorimos de asignación en la jerarquía de memoria y se mostró el potencial que ofrencen para reducir los accesos a esta. Inevitablemente, sea cual sea el algoritmo que se emplee, es necesario desalojar alguna variable cuando no quedan registros libres. Como se puso de manifiesto anteriormente, los accesos a memoria resultantes de este desalojo presentan una elevada localidad y no deterioran el comportamiento de la cache, aunque si su consumo. En este capítulo presentaremos una técnica hardware que, aprovechando la localidad, filtra gran parte de estos accesos, ahorrando energía.

Comenzaremos por presentar la motivación y la idea sobre la que se sustenta nuestro diseño. A continuación, describiremos una primera implementación del mismo y analizaremos los potenciales problemas que plantea así como sus soluciones. Finalmente, evaluaremos experimentalmente nuestra propuesta y esbozaremos las principales conclusiones que se derivan de ella.

4.1. Motivación

En el Capítulo 3 se explica que el algoritmo de asignación de registros aloja a las variables en registros siempre que es posible. Cuando esto no es posible, las variables son alojadas en memoria. Sin embargo, las posiciones de memoria asignadas no pueden ser fijas, porque entonces no se permitirían llamadas recursivas a una función, ya que los resultados de la llamada 'hija' modificarían las variables de la llamada 'padre', y este no es, en general, un efecto deseado. Por ello, los programas cuentan con una zona de pila en la memoria donde se almacenan estas variables. El espacio de almacenamiento local de las funciones se conoce como marco de activación, y contiene, además de las variables temporales alojadas en memoria, los parámetros de la función, el valor devuelto por esta,y ciertos datos del contexto de la función llamante que deben ser guardados.

Como se dijo en la sección 3.4.3 la pila es una región que tiene una elevada localidad de memoria, tanto espacial como temporal. Aunque los accesos a ella suelen resultar aciertos de cache y apenas afectan al rendimiento, la elevada frecuencia de los mismos supone un importante consumo de energía. Cabe preguntarse por lo tanto si es posible aprovechar su extraordinaria localidad para alojarlos en otra extructura, reduciendo el consumo de la cache.

4.2. Filtro de accesos a pila

Como hemos mencionado previamente, los accesos a la región de pila tienen una localidad especial muy elevada. El estudio del puntero de pila, *sp*, así lo

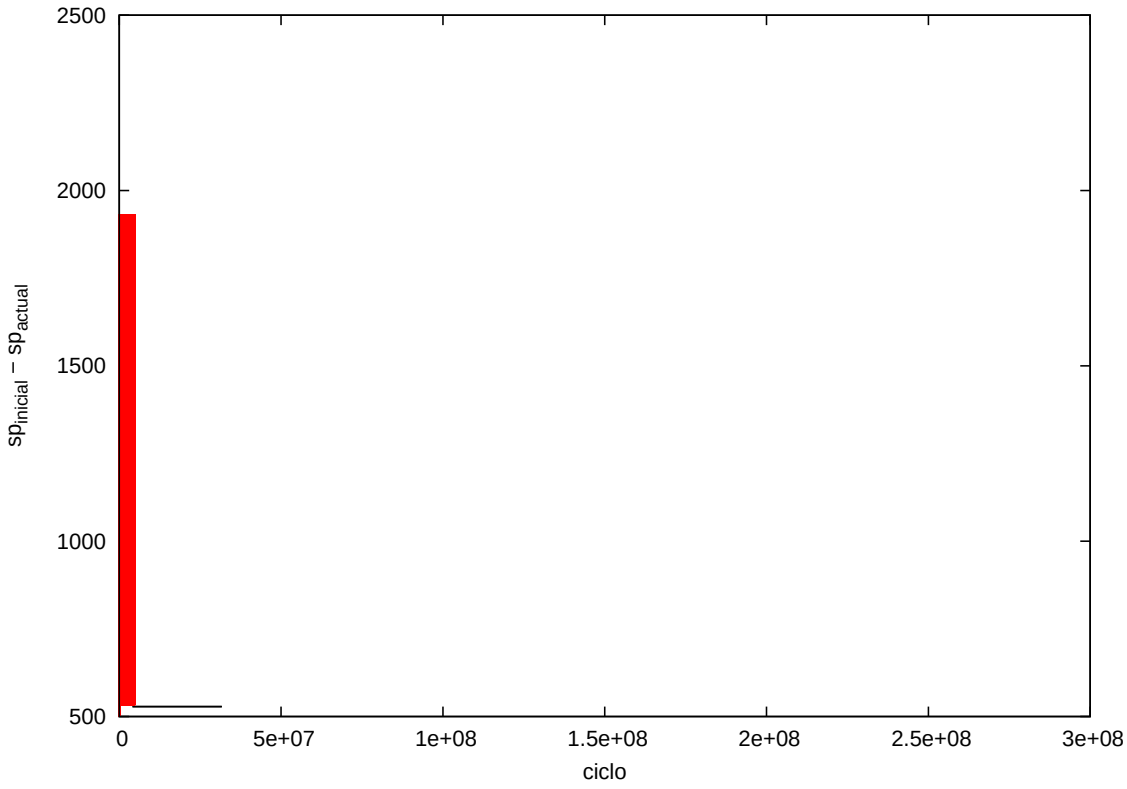

FIGURA 4.1: Evolución del *sp* durante la ejecución del benchmark dijkstra.

pone de manifiesto. Para todos los benchmarks considerados el *sp* se mueve en rangos de entre 40 y 128 bytes (entre 10 y 32 palabras) durante la mayor parte de su ejecución. Los accesos a la pila además se realizan en su mayoría en la proximidad del puntero de pila, en concreto, en el rango comprendido entre $sp - 64$ y $sp + 256$, es decir, 16 palabras por encima y 64 por debajo. A modo de ejemplo, mostramos para dos de los benchmarks empleados en nuestro estudio experimental, *dijkstra* y *adpcm*, la evolución del puntero de pila (Figuras **??** y 4.2) y el histograma de distancias respecto al *sp* (Figuras **??** y 4.4).

Esta extraordinaria localidad especial, sugiere que la mayor parte de lec-

turas/escrituras a pila podrían satisfacerse mediante una pequeña cache o estructura de almacenamiento alternativa, evitando el desperdicio energético que suponen los accesos a la cache.

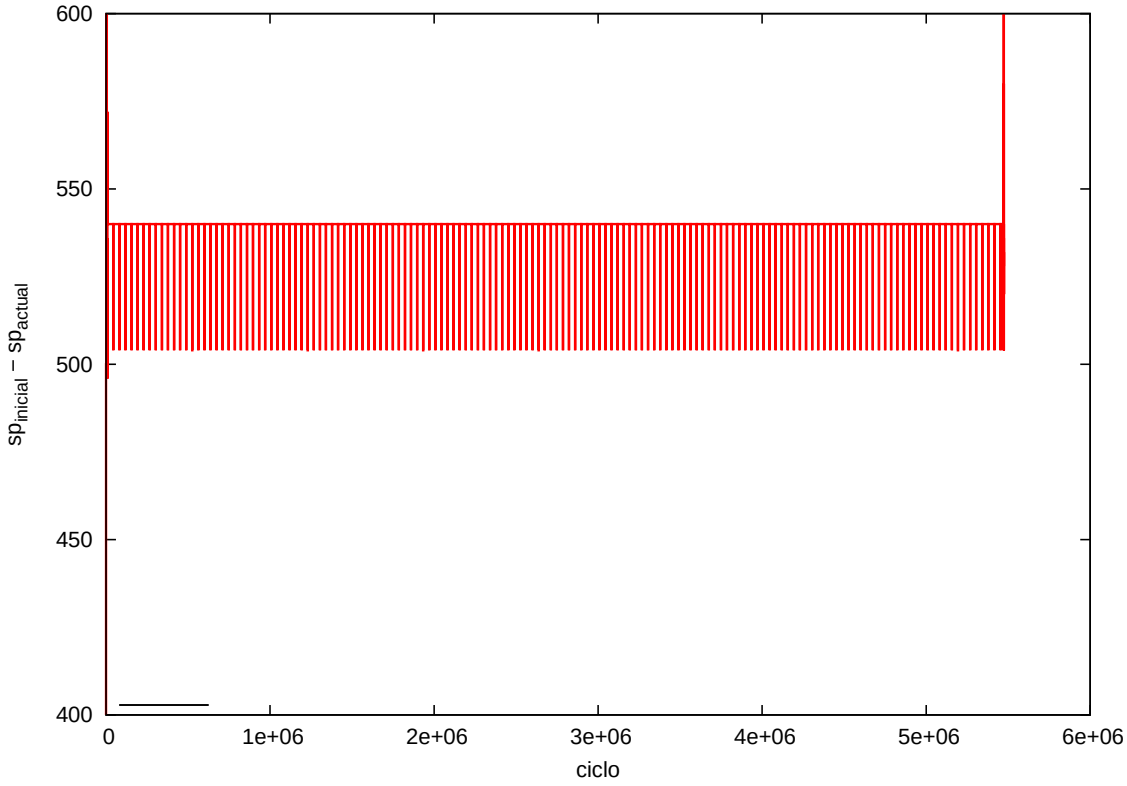

FIGURA 4.2: Evolución del *sp* durante la ejecución del benchmark adpcmdec.

4.3. Implementación

Con el fin de evitar el uso de lógica asociativa, nosotros hemos optado una implementación del filtro mediante registros tal y como detallamos a continuación.

En un procesador SA-110 StrongARM como el descrito en la tabla 3.1 se

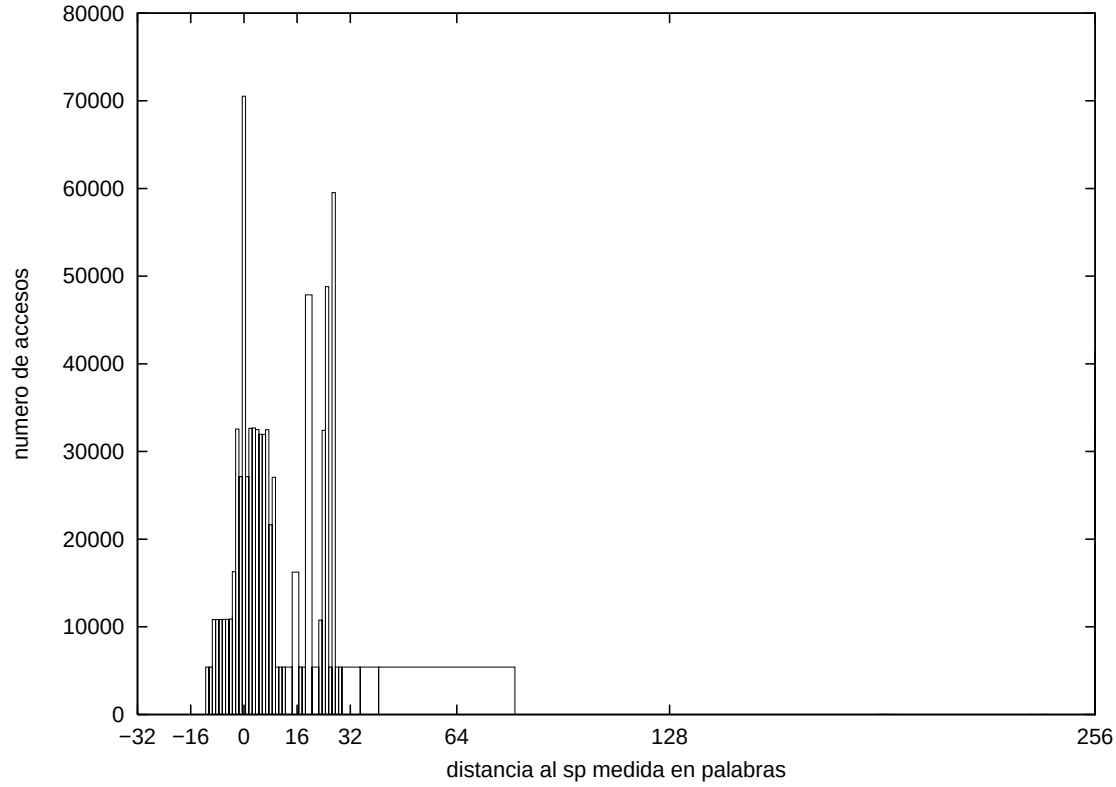

FIGURA 4.3: Histograma de las distancias entre el *sp* y las direcciones de acceso a la pila durante la ejecución del benchmark dijkstra.

introduce una una cola circular de registros que actúa como nivel previo a la cache para los accesos a pila. La cola cuenta con N registros, cada uno de los cuales lleva asociado un bit de presencia. Además se emplea un registro, r_{base} de $log_2 N$ bits que nos indica cual de todos los registros es el que corresponde a la dirección apuntada por el *sp*.

Los accesos a se hacen calculando la distancia entre la palabra que queremos acceder y la palabra apuntada por el *sp*, d. Una vez calculada d, si d cae dentro de nuestro filtro, el registro al que tenemos que acceder es $(r_{base}+d)\,\%N$. A diferencia de otras propuestas como las *cache de pila* nuestro filtro es di-

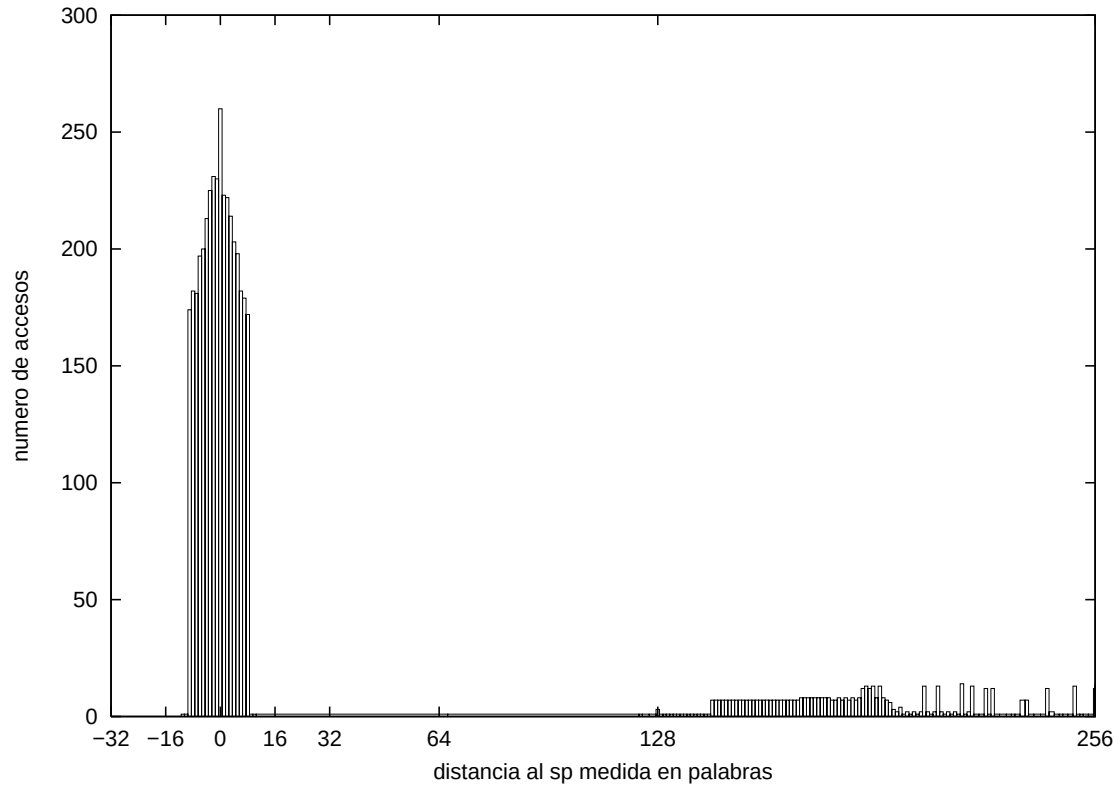

FIGURA 4.4: Histograma de las distancias entre el *sp* y las direcciones de acceso a la pila durante la ejecución del benchmark adpcmdec.

reccionado por la distancia al *sp*, por lo que no requiere de etiquetas y puede prescindir de toda lógica asociativa.

Cada vez que se realiza un acceso a memoria se calcula, en paralelo a la generación de la dirección efectiva, la distancia en palabras entre la dirección accedida y el *sp*, empleado para ello una unidad específica. Si la distancia es menor, en valor absoluto, que $\frac{N}{2}$ entonces accedemos al filtro. De lo contrario, se accede a la cache.

Si al acceder al filtro el dato está presente, el procesador no realiza el acceso a la cache ni la tlb de datos. Si, de lo contrario el dato no está, se realiza el

acceso a ambas estructuras, en el siguiente ciclo. Posteriormente, se copia el dato en el filtro y se activa su bit de presencia. En el caso de las escrituras, si estas son de tamaño palabra y el dato no está en el filtro tampoco es necesario acceder a la cache ni a la tlb.

En definitiva, el filtro que hemos diseñado almacena localmente las palabras situadas en la vecindad $N/2$ del *sp*. Por ello, cuando se modifica el *sp* es necesario actualizarlo de manera adecuada. Cada vez que se modifica, hay que calcular cual ha sido la variación del valor del puntero en número de palabras. Una vez conocido el valor, hay que guardar en memoria todas las posiciones válidas que se desalojan del filtro debido a esta modificación y a continuación se rota el buffer circular. Esta rotación se lleva a cabo incrementando o decrementando el registro r_{base}. El bit de presencia de las nuevas entradas se desactiva.

Esta actualización, que tiene una complejidad no despreciable, se lleva a mediante la inyección de instrucciones de un modo análogo a como se tratan las instrucciones complejas de movimiento *stm* y *ldm*. De esta manera, el hardware introducido se reduce a una memoria de $N * 4$ bytes más N bits de presencia y un restador para calcular la distancia entre la dirección efectiva y el *sp*.

4.4. Problemas a resolver

El diseño que hemos presentado tiene que hacer frente a ciertos problemas. El primero y más notable es que al introducir un nuevo nivel en la jerarquía tendríamos que mantener la coherencia también en dicho nivel. Sin embargo,

esto no es necesario ya que la pila es una región privada de memoria y el resto de procesadores no comparten esos datos.

Otro problema que presenta nuestro diseño es que rompe propiedad de inclusión de la jerarquía de memoria. Un dato puede estar presente en el filtro pero haber sido reemplazado en la cache. Al no informar a la cache de aciertos en el filtro, la política de remplazo puede hacer que los bloques de la cima de pila sean sacados prematuramente. Este comportamiento, lejos de representar un problema puede incluso ser beneficioso, ya que se evita que los bloque de la cima de la pila entren en conflicto con otros.

Existe un tercer problema que tiene que ver con la utilización de un único bit de estado para las entradas del filtro. Cuando una palabra es desplazada del filtro siempre se escribe en cache, aunque no haya sido modificada. Por esto, si un programa tiene un patrón de accesos que lee los datos alojados en un extremo de la estructura y, posteriormente, modifica el puntero a pila desalojando esos datos, entonces estaría haciendo escrituras en memoria que no son necesarias, lo cual conlleva un coste tanto en tiempo como en energía, que es contrario a nuestros intereses. Como se puede ver en la sección 4.6.2 este patrón es inusual, aunque no inexistente. Este problema puede ser solucionado añadiendo a la estructura un segundo bit que nos indique si el valor ha sido modificado o no, y la escritura a $dl1 se haga sólo en caso de que el dato que reside en la estructura haya sido modificado.

4.5. Entorno Experimental

En esta sección se describen las herramientas que componen el entorno experimental: el compilador usado, el simulador empleado para llevar a cabo la evaluación de los programas generados y los benchmarks usados en dichas mediciones.

4.5.1. Simulador

El simulador usado es es el mismo que en el capítulo anterior, el SimPAnalyzer [KAMG02], pero ha sido modificado para incluir nuestro filtro tanto en la simulación micro-arquitectónica como en el modelo de consumo. La configuración del procesador ha sido la misma que la descrita en la tabla 3.1 incluida en la sección 3.3.1. Hemos considerado los siguentes tamaños de filtro: 32, 48, 64, 96 128, 192 y 256 palabras (32 bits).

4.5.2. Compilador

Los cambios introducidos en la microarquitectura no conllevan ningún cambio en el ISA ni en el proceso de compilación, por lo que el compilador no ha tenido que ser modificado. El compilador usado es gcc-2.95.2, distribuido con SimpleScalar/ARM [ALE02]. Los programas han sido compilados con la opción de optimización estándar "-O2".

4.5.3. Benchmarks

Los benchmarks usados son *adpcm*, *bitcount*, *CRC32*, *dijkstra*, *jpeg*, *sha*, *stringsearch*, *tiff2bw*, *tiff2rgba* y *tiffmedian* de MiBench [GRE+01], y *mpeg2dec*

de MediaBench2 [LPMS97]. Las cargas de trabajo usadas han sido las distribuidas con los propios benchmarks.

4.6. Resultados

En esta sección se presentan y se discuten los resultados obtenidos por nuestro mecanismo en la ejecución de los benchmarks. La estructura propuesta se analiza desde dos puntos de vista estrechamente relacionados. Por un lado se estudian los resultados funcionales, en cuanto al uso y aprovechamiento del filtro introducido:

1. **Porcentaje de accesos:** porcentaje del total de referencias a la pila que se resuelven dentro del filtro.

2. **Tasa de aciertos:** porcentaje del número de accesos al filtro que se resuelven sin necesidad de acceder a memoria.

Por otro lado, se estudian los resultados arquitectónicos, principalmente el consumo de la cache de datos, el tlb de datos, y el consumo total de la arquitectura, aunque también es importante el tiempo de ejecución.

4.6.1. Resultados Funcionales

Primero se van a comentar los aspectos funcionales de la estructura. En la Figura 4.5 podemos observar como varía la tasa de aciertos de la estructura según aumentamos el tamaño de ésta, para los diferentes benchmarks. En dicha figura podemos ver cómo incrementar el tamaño de la estructura es beneficioso en pocos casos, ya que sólo *sha*, *stringsearch* y *jpeg* ven su tasa de aciertos aumentada significativamente con el incremento del tamaño de la estructura. También podemos ver que los benchmarks están fuertemente divididos en un grupo que tiene una tasa de aciertos cercana a 0 sin importar

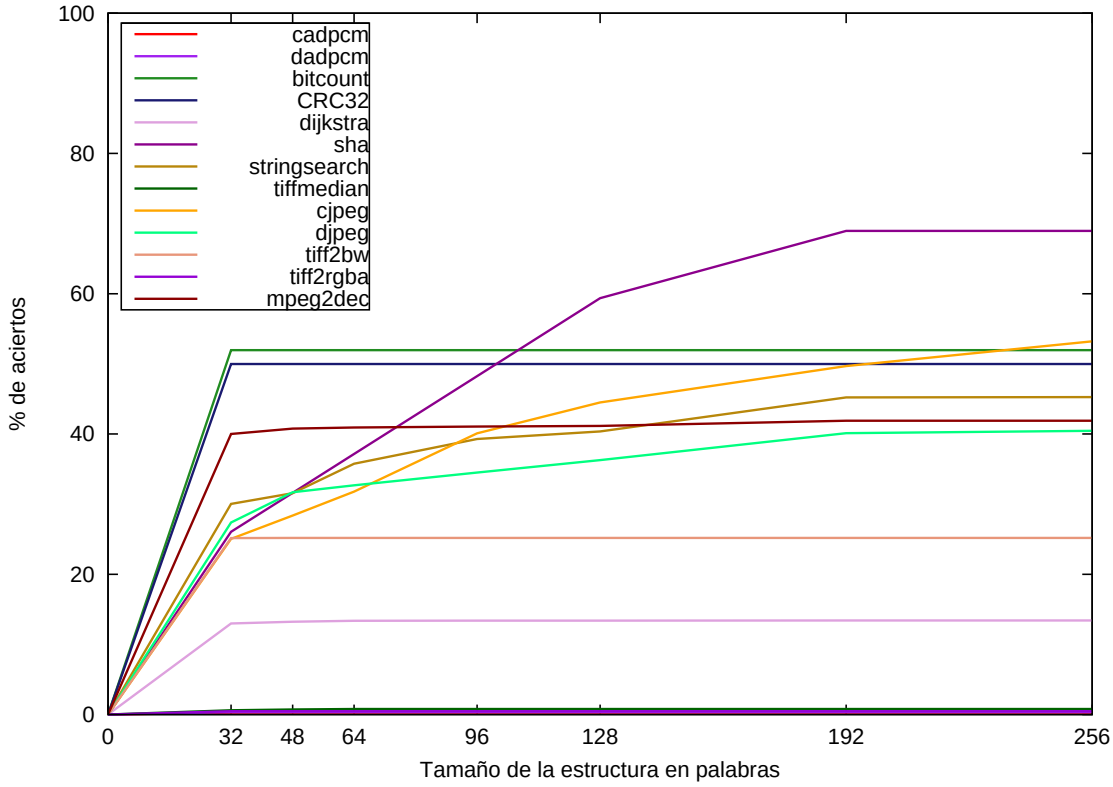

FIGURA 4.5: Tasa de aciertos.

el tamaño de la estructura, y otro grupo, que con 32 palabras tiene una tasa de éxitos entre el 10 % y el 50 %, y que esta se mantiene con pocas variaciones aunque aumentemos el tamaño de la estructura usada.

Si prestamos atención a la colección de programas que no se beneficia de la existencia de la estructura, que son: *adpcm* (codificación y decodificación), *bitcnts*, *mpeg2dec*, *tiffmedian*, *tiff2rgba* y *tiff2bw*. Estos siete programas realizan una pasada por un fichero de entrada y escriben un fichero de salida. Al no iterar sobre colecciones de datos alojadas en la pila no sacan provecho de la estructura, sin importar cuan grande sea ésta. En cambio, los programas que sí se benefician es porque trabajan con datos que residen en la pila. Los

TABLA 4.1: Porcentaje de accesos que son a la pila

Benchmark	%	**Benchmark**	%
adpcm enc	0.44 %	**adpcm dec**	0.54 %
bitcnts	1.21 %	**crc**	91.20 %
jpeg enc	55.73 %	**jpeg dec**	40.16 %
dijkstra	66.05 %	**mpeg2dec**	0.02 %
sha	90.59 %	**string search**	48.43 %
tiff2bw	25.33 %	**tiff2rgba**	0.89 %
tiffmedian	0.38	$media_{aritmetica}$	**32.64**

programas que trabajan con marcos de activación que ocupen 32 palabras o menos son aquellos cuya tasa de aciertos no mejora, en cambio aquellos que tienen marcos de activación más grandes, o con mayor profundidad en el árbol de llamadas a función, sí ven una mejora en la tasa de aciertos cuando se incrementa el tamaño de la estructura. Todo esto se puede observar en la Tabla 4.1, donde se muestra el porcentaje de accesos que se producen a la pila respecto al total de accesos. Esta tabla es a su vez la cota superior a la tasa de aciertos mostrada en la Figura 4.5.

Los fallos en los accesos al filtro pueden ser caracterizados, pero en este caso en sólo 2 tipos:

1. *Fallos por rango:* Son los fallos que se producen cuando el dato al que queremos acceder está demasiado lejos del *sp*, y no tenemos espacio suficiente en nuestra estructura. Aparecen representados en la Tabla 4.6

2. *Fallos por desplazamiento:* Son los fallos que se producen cuando el dato al que queremos acceder ha sido desalojado de la estructura debido a las rotaciones ocasionadas por movimientos del *sp*. La figura 4.7 muestra la tasa de este tipo de fallos para los distintos benchmarks.

Nuestra estructura no distingue entre accesos a stack y accesos a otra zona.

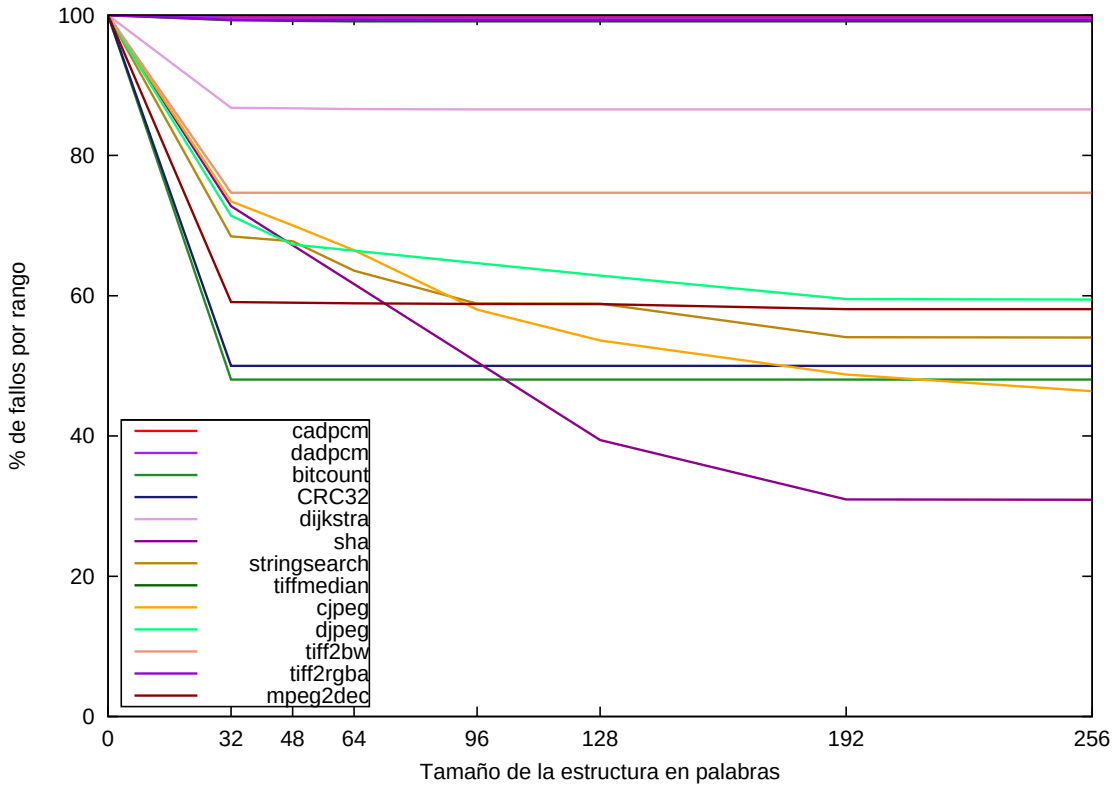

FIGURA 4.6: Porcentaje de accesos a memoria que caen fuera de los límites de la estructura.

Sólo distingue entre accesos que provocan un fallo por rango y aciertos. Por este motivo la mayoría de fallos por rango son producidos por accesos al heap. Esta afirmación se fundamenta en que en 9 de los 13 benchmarks aumentar el tamaño de la estructura no reduce el número de fallos. Lo cual indica que los datos accedidos está realmente lejos (más de 128 palabras = 512 bytes) del puntero a pila.

Los fallos por desplazamiento son mucho menores que los fallos por rango. Menos del 2 % frente a más de un 50 % en la mayoría de los casos. Además su comportamiento es más aleatorio, ya que si el filtro es pequeño, los fallos por rotación son enmascarados por los fallos por rango. Según aumenta el tamaño

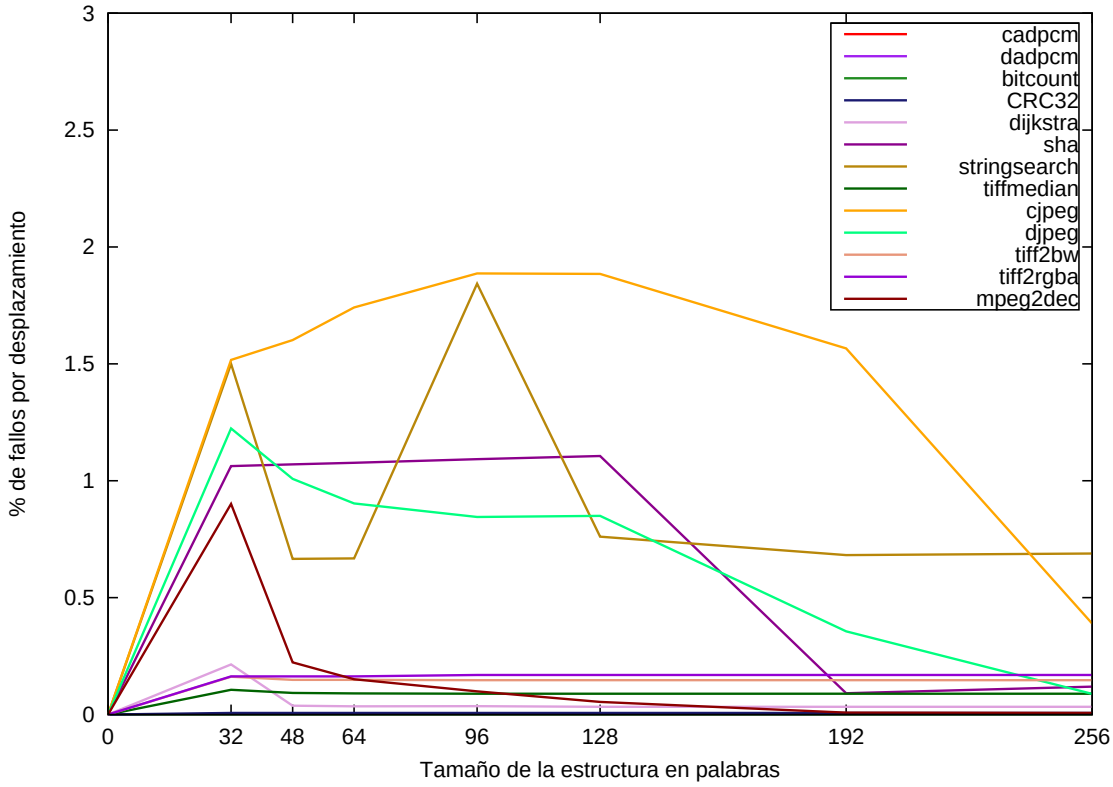

FIGURA 4.7: Porcentaje de accesos a la estructura que no tienen el dato presente.

del filtro caben más datos que generan fallos por rotación. Si aumentamos aún más el tamaño del filtro, este llega a ser suficientemente grande como para que quepan todos los datos que son desplazados por las rotaciones, y por lo tanto el número de fallos por conflicto se reduce. Esto puede observarse en la Figure 4.7 en la gráfica correspondiente a *jpeg*.

En virtud de estos resultados es razonable pensar que esta modificación va a producir un ahorro energético, ya que en varios casos se tiene una tasa de aciertos entre el 25 % y el 50 %. El coste de acceso a una cache y a un tlb, que son ambas estructuras con lógica asociativa, y de tamaño significativo (16 Kb y 2.5 Kb respectivamente) es elevado. En cambio el coste de acceso

a una estructura sin lógica asociativa de entre 128 y 1024 bytes es bastante reducido en comparación. No es esperable, por otro lado, ninguna mejora en el rendimiento de las aplicaciones, ya que en nuestro procesador tiene un coste de acceso al primer nivel de cache de 1 ciclo. Si acaso, podríamos esperar quizá una leve degradación del rendimiento, ya que las instrucciones que modifican el valor del registro *sp* requieren de la inyección de microoperaciones para hacer la rotación de la estructura y las escrituras de los datos que son desalojados.

4.6.2. Resultados Arquitectónicos

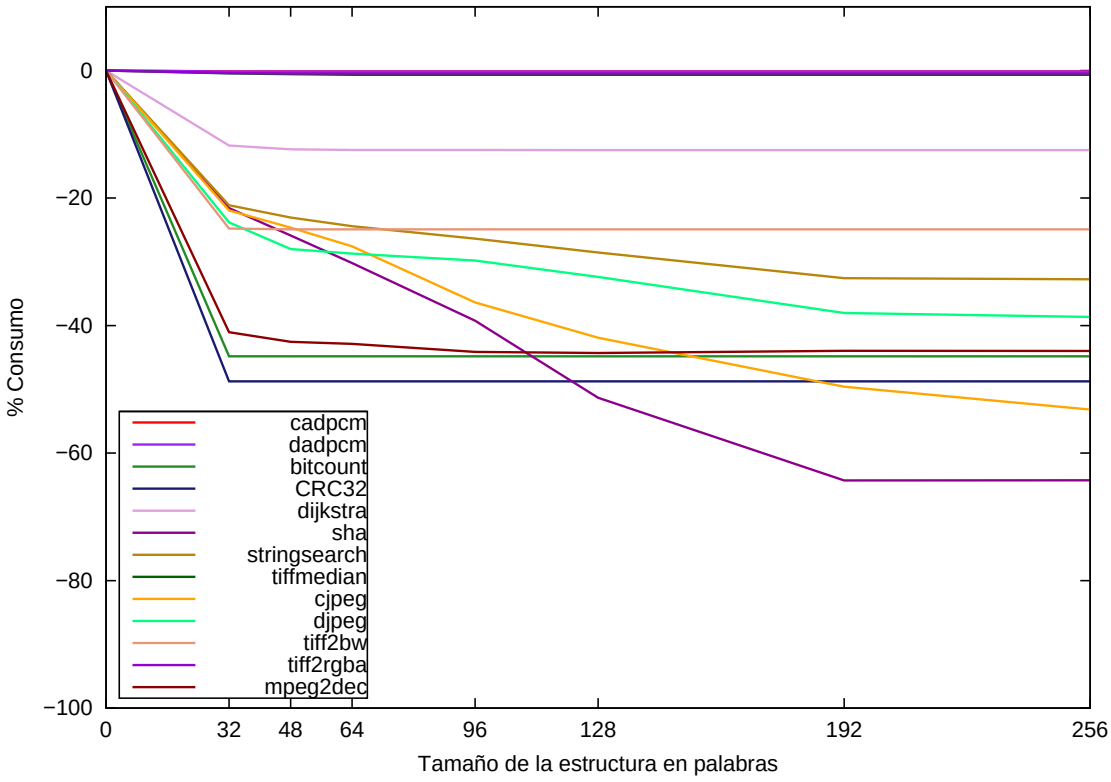

FIGURA 4.8: Variación del consumo total de la $dl1.

Esta sección muestra los resultados más relacionados con la arquitectura, como son el consumo y el rendimiento. En las Figuras 4.8 y 4.9 podemos observar las variaciones en el consumo de la cache de datos de nivel 1 y del procesador respectivamente. En ambas está representada la reducción o el aumento porcentual respecto a la ejecución en el procesador sin la incorporación del filtro.

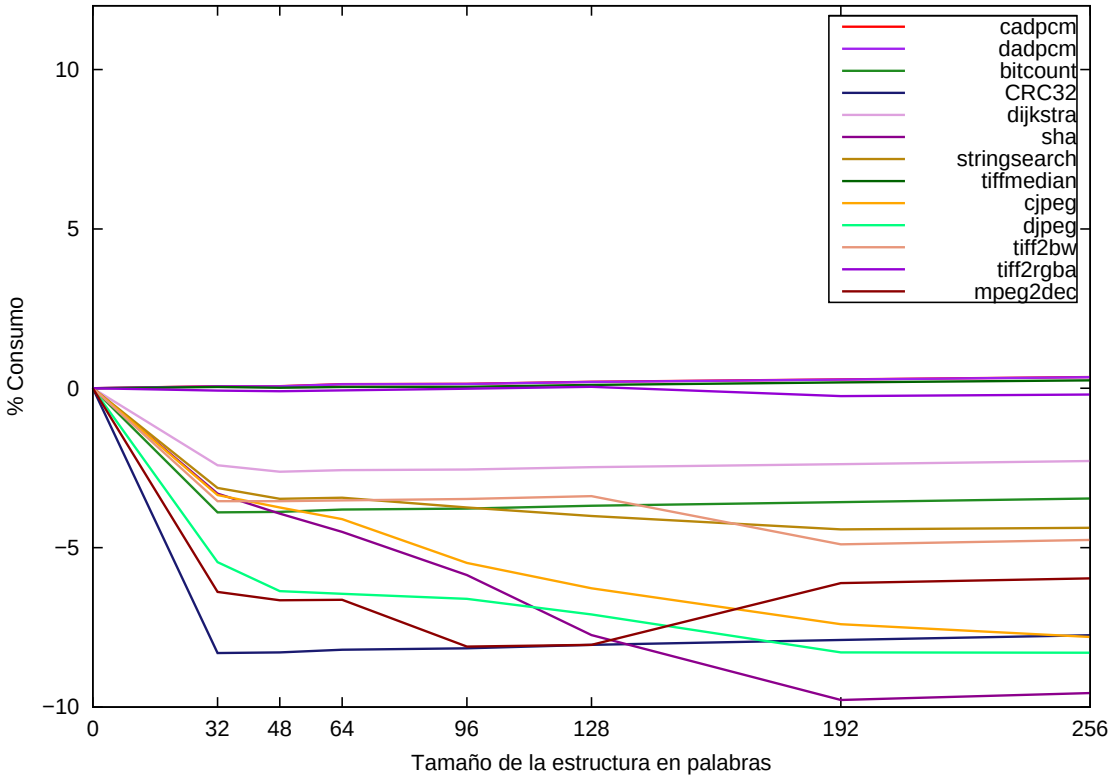

FIGURA 4.9: Variación del consumo total de la arquitectura.

Podemos ver como los programas que, como se señaló en la sección 4.6.1, hacen uso de la estructura, se consigue reducir el consumo de la cache de datos entre un 20 % y un 50 % en la mayoría de los casos y hasta en un 64.3 % en el mejor de los casos (*sha* con un tamaño de 192 palabras). El único programa que

tiene un mayor consumo en la versión modificada del procesador es tiff2rgba en el caso de que la estructura tenga un tamaño de 64 palabras. El por qué de este resultado es la aparición de el patrón de accesos comentado en la sección 4.4.

Las reducciones en el consumo de la $dl1 se ven reflejadas en una reducción de consumo total de la microarquitectura. Como muestran las figuras, el consumo total se ve reducido en un 2.8 % y un 4.3 % haciendo media de todos los benchmarks para los distintos tamaños de la estructura hasta casi un 10 % en el mejor de los casos. Esto es una reducción significativa, teniendo en cuenta que la complejidad añadida al procesador es bastante baja. Además con un tamaño de sólo 32 palabras obtenemos un ahorro del 3.46 % en media, y alrededor de un 8 % en los mejores casos.

Tal y como se esperaba, la Tabla 4.10 muestra como el tiempo de ejecución experimenta pocas variaciones, entre un -2 % y un 2 % quitando el caso patológico de *tiff2rgba* en el que si la estructura tiene un tamaño de 64 palabras, el tiempo de ejecución se ve incrementado en un 11 %. Estas pequeñas variaciones se deben a dos factores, el primero es la sobrecarga de las rutinas inyectadas por las modificaciones al valor del puntero a pila, que hace que el programa vaya más lento. El segundo factor es que si un bloque de datos que no esté en las inmediaciones del puntero a pila produce un conflicto con otro bloque de datos de la pila, como los accesos al segundo bloque no acceden a cache no se producen fallos de conflicto, lo cual acelera ligeramente la ejecución de el programa. De esta manera la jerarquía de memoria deja de cumplir la propiedad de inclusión, pero a cambio se obtienen beneficios en el tiempo de ejecución y en el consumo de energía.

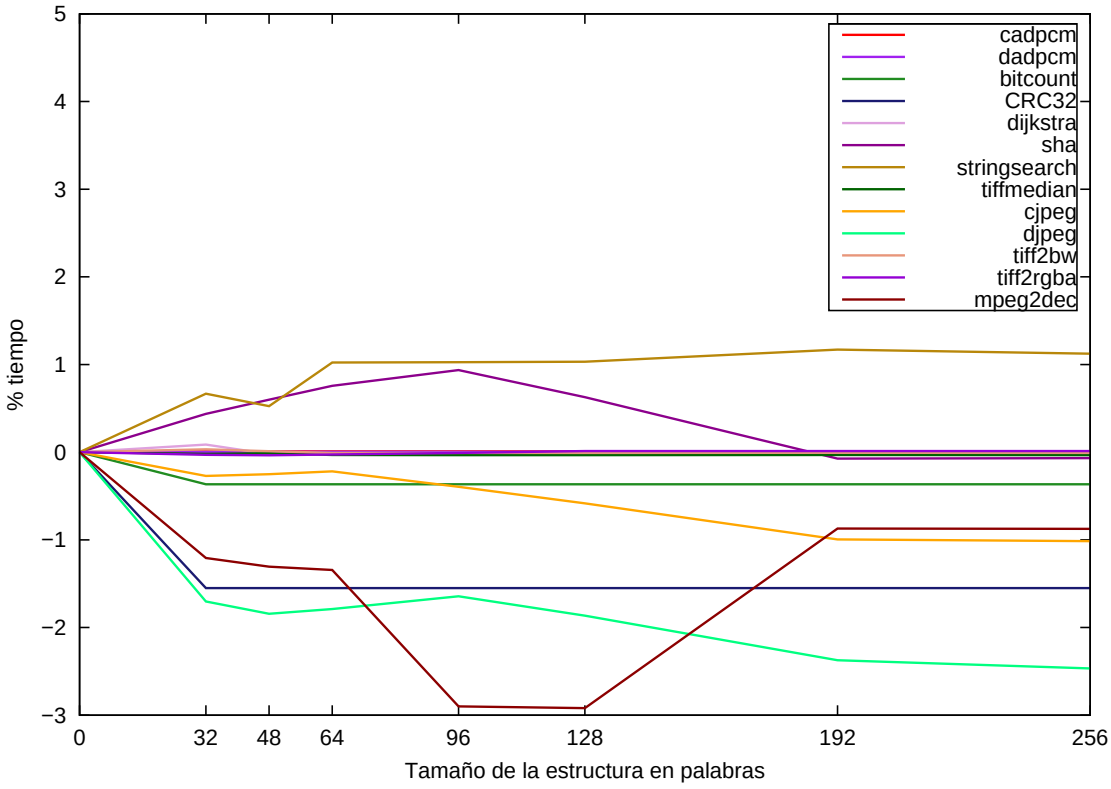

FIGURA 4.10: Variación del tiempo de ejecución.

4.7. Conclusiones

En este capítulo hemos visto cómo el puntero de pila se mueve en rangos de valores bastante reducidos. Este tipo de localidad del que carecen el resto de secciones del proceso puede ser aprovechado dotando al procesador de una pequeña estructura basada en registros que filtre los accesos a la pila.

El diseño que hemos propuesto ha demostrado ser muy efectivo, y logra suministrar la mayor parte de los datos de pila sin necesidad de acceder a la cache, reduciendo significativamente el consumo de energía de ésta última.

De este estudio también se puede concluir que el tamaño del filtro de cache no tiene por qué ser demasiado elevado, ya que en términos generales, los programas tienen un comportamiento fuertemente marcado. Es decir, o bien no obtienen beneficio del filtro, o bien obtienen un beneficio muy cercano al máximo con únicamente 32 registros, y aumentar el tamaño del filtro no conlleva mejoras significativas.

Capítulo 5

Trabajo relacionado

En este capítulo comenzamos describiendo dos técnicas que permiten al usuario disponer de más registros para reducir la cantidad de código de desalojo, primero las *ventanas de registros* y luego la *codificación diferencial*. A continuación mostramos otras técnicas enfocadas a reducir la sobrecarga por llamadas a función incluyendo más registros visibles al usuario. Para terminar mostramos el trabajo que se ha hecho en caches de pila y caches por regiones.

5.1. Ventanas de registros

Algunas de las propuestas aparecidas en los últimos años requieren ciertas modificaciones del hardware. Ravindran y otros [RSM+03] sugieren crear distintos bancos de registros a los que llaman ventanas. En cada momento una única ventana está activa, y las instrucciones sólo pueden usar los registros de la ventana activa. Hay que añadir dos nuevas instrucciones al repertorio; una para mover los datos entre distintas ventanas, y otra para cambiar de

ventana activa. De esta manera el compilador puede usar más registros. Esta solución requiere que el compilador sea modificado para poder explotar las mejoras arquitectónicas introducidas. La principal ventaja de esta propuesta es que aumenta el número de registros arquitectónicos sin tener que cambiar el formato de instrucción. Además, cada ventana es un banco separado, por lo que el hardware necesario para implementar dos ventanas de n registros es más sencillo y consume menos energía que el necesario para implementar un banco de $2n$ registros. El uso de ventanas de registros nos permite tener n registros separados en m ventanas, cada uno de los cuales es direccionado únicamente $log_2(n) - log_2(m)$ bits.

De esta manera cualquier arquitectura que incluya ventanas de registros experimenta un incremento en el número de registros por un factor igual al número de ventanas incluidas en la implementación.

Otra de las ventajas de esta propuesta es la compatibilidad binaria (hacia delante), ya que una arquitectura que no use ventanas de registros es igual que una arquitectura que implemente una única ventana de registros.

5.2. Codificación diferencial

Zhuang y Pande presentan la Codificación Diferencial [ZP05]. Esta idea dota al procesador de un banco de n registros donde cada registro tiene un índice entre 0 y $n-1$. Cada registro del banco no es direccionado por su índice, si no por la diferencia entre su índice y el índice del último registro usado. Por lo tanto podemos definir un "rango diferencial", r, que es la máxima distancia que puede existir entre dos registros direccionados secuencialmente. Este

número es menor que la cantidad de registros disponibles. De esta manera podemos direccionar n registros usando únicamente $log_2(r)$ bits. Ya que el rango diferencial es menor que el número de registros disponibles en el banco, es necesario definir dos nuevas instrucciones en el repertorio para poder permitir cualquier cadena de referencias. La primera es $setLastReference(n)$ que fija artificialmente el valor del registro que almacena el índice el último registro referenciado a n. La otra instrucción necesaria es $setLastReferenceAfter(n, t)$ que fija el valor de la última referencia a n después de la t-ésima referencia.

Esta solución también requiere la modificación del compilador, ya que la manera de direccionar registros es completamente diferente. Al contrario que en el caso las ventanas de registros, con esta modificación no existe compatibilidad binaria.

5.3. Otras técnicas

Otras técnicas hardware, como las empleadas en el procesador ADSP-219x [Ana01] y Xtensa de Tensilica [Ten02], incluyen ventanas de registros para reducir la sobrecarga de las llamadas a procedimientos y los cambios de contexto cuando se producen interrupciones en sistemas de tiempo real.

La arquitectura SPARC [SPA92] cuenta con una ventana lógica que se mueve sobre un banco de registros físicos. Las instrucciones sólo pueden usar los 8 registros globales y los registros de la ventana. Los registros de la ventana se dividen en 3 secciones: registros de entrada, registros locales, y registros de salida. Cuando la ventana se desplaza lo hace de tal manera que queda parcialmente solapada con la ventana anterior/siguiente, de manera que los

registros de salida de una ventana coinciden con los registros de entrada de la siguiente. Este solapamiento reduce la sobrecarga de las llamadas a función ya que permite poner los parámetros de una función en la región de salida. A continuación desplazamos la ventana hacia delante y hacemos la llamada a la función, el solapamiento hace que los parámetros de la función invocada estén en su región de entrada. De igual manera los valores de retorno de la función se escriben en la región de entrada, se ejecuta la instrucción de retorno y se desplaza de nuevo la ventana a su posición anterior, de manera que los resultados están en la zona de salida. El deslizamiento de la ventana no es automático, si no que se ha añadido la instrucción *save* al ISA, que es la encargada de deslizar la ventana antes de llamar a una función. La instrucción *restore* es la que se encarga de hacer el desplazamiento de la ventana en el otro sentido.

Esta implementación del banco de registros reduce la sobrecarga producida por las llamadas a función, ya que evita guardar los registros y los parámetros en la pila, y luego tener que restaurarlos, reduciendo el consumo y acelerando la ejecución.

Estas tres técnicas modifican el ISA y por lo tanto requieren la modificación del compilador para poder usar los recursos introducidos.

5.4. Cache de pila

Han sido muchos los autores que se han dado cuenta de que la región de la pila presenta una localidad distinta a la de otras regiones de datos, y por lo tanto han decidido aprovechar estas diferencias para beneficiarse. A

principios de los 80, cuando las tendencias del diseño de procesadores eran aumentar la complejidad de los procesadores para que entendieran lenguajes de alto nivel Ditzel [DM82] propone eliminar el banco de registros e incluir una *cache* de la pila para favorecer las arquitecturas memoria-memoria y ahorrar al compilador el complejo proceso de asignación de registros. Hay que recordar que desde entonces la tendencia ha sido trasladar parte de la complejidad del procesador al compilador, y esa técnica no sería aplicable a las arquitecturas contemporáneas.

Lee y otros [LSNT01] proponen un banco para los valores de la pila. Este banco consiste en una estructura basada en registros para almacenar los valores cercanos a la cima de la pila. Este banco está dotada de bits de presencia y modificación por cada registro. Detecta los accesos que usan como registro índice el *sp* en la fase de decodificación. Los accesos que se hacen usando como índice otro registro tienen que comparar su dirección efectiva con el rango de la pila almacenado en el banco. El banco contiene tantas tags como páginas distintas pueda ocupar la pila, es decir, una pila de 8Kb en un sistema con páginas de 4Kb necesita almacenar 3 tags. El objetivo de este trabajo es el rendimiento, ya que el tamaño del banco de pila es el mismo que el de una cache de datos.

Huang y otros [HRT01] y Geiger y otros [GMT05] proponen dos técnicas que consisten en separar los accesos a cache en accesos a la pila y accesos al heap en el primero, mientras que en el segundo separan entre datos frecuentemente accedididos y datos accedidos con menor frecuencia. En ambos casos el objetivo es la reducción de consumo, aunque la implementación propuesta en ambos casos usa memorias cache, solo que de menor tamaño y menor

asociatividad, por lo tanto menor consumo.

Hemsath y otros [HMS07] presentan en un informe técnico los resultados de introducir una cache de pila en el procesador StrongARM SA-110. Este trabajo también estudia la inclusión de una memoria cache, de tamaño igual que la $dl1. Cuentan además con una unidad que dinámicamente se encarga de predecir cuando será necesario desplazar bloques de la cache de pila, o cuando se producirá un fallo en la cache para planificar las transferencias con los niveles inferiores de la jerarquía lo antes posible y así enmascarar las latencias. El estudio evalua la cache en términos de rendimiento.

Estas tres últimas propuestas colocan la cache de pila de manera horizontal a la cache de nivel 1. Esto requiere que los fallos accedan a la cache de nivel 2 o a memoria RAM en caso de no haber. Esto consume más energía que en el caso de [LSNT01] y este trabajo en el que el almacenamiento extra se coloca en nivel 0, de manera que los fallos son servidos por la cache de nivel 1.

Capítulo 6

Conclusiones y principales aportaciones

A lo largo de este trabajo hemos visto cómo se pueden reducir los accesos a memoria usando mecanismos software como hardware. Por un lado hemos evaluado diversos algorimos de asignación, analizando su impacto sobre la jerarquía de memoria y, por otro lado, hemos propuesto un mecanismo de filtrado que permite reducir los accesos a memoria derivados del desalojo de registros y el paso de parámetros a funciones.

Las principales conclusiones del estudio comparativo de los algoritmos de asignación son las siguientes:

- Como cabía esperar, el principal efecto de los algorimos de asignación en la jerarquía de memoria tiene que ver con el número de accesos a la misma, ya sea leer instrucciones o bien para leer/escribir datos.

- El comportamiento de la cache no se ve afectado por los algoritmos de

asignación ya que las variables desalojadas de los registros se almacenan en la pila, y acceder a ellas rara vez produce fallo de cache.

- Excepto para el más simple, LS, se observan pocas diferencias entre los algorimos de asignación tanto en términos de consumo como de rendimiento. Por lo tanto, un sencillo algoritmo de escaneado lineal como UCB puede resultar suficiente tanto para compiladores *off-line* como *JIT*.

- El comportamiento de CGC con respecto a IRC difiere substancialmente de lo indicado por otros autores en el contexto de otro tipo de programas y computadores. Esto sugiere que los algorimos de asignación deben ser reevaluados en cada contexto particular, ya que tanto los estudios teóricos (los más frecuentes) como los que se efectuan para otras plataformas o cargas de trabajo pueden dar lugar a conclusiones erroneas.

- Incorporar información de la arquitectura y/o diseñar algorimos más específicos sin duda alguna mejorararía substancialmente la asignación de registros. Sin embargo, esto objetivo excede el ámbito del presente trabajo y se he dejado como trabajo futuro.

Aunque la idea de emplear un almacenamiento específico para la pila no es nueva, el filtro que hemos propuesto presenta las siguientes ventajas:

- Al tratarse de elemento ubicado antes de la cache, los aciertos evitan tanto el acceso a esta como a la tlb de datos. A diferencia de las *cache de pila* que se situan en paralelo con la cache de datos, este mecanismo

permite importantes ahorros de energía en este nivel de la jerarquía, incluso con tamaños de filtro de tan solo 32 palabras.

- La discriminación entre aciertos y fallos se lleva a cabo en paralelo con el cálculo de la dirección efectiva del acceso, mediante una unidad específica. Esto evita penalizaciones adicionales y permite tratar de manera homogénnea los accesos a pila tanto si emplean el registro *sp* como si emplean un registro de propósito general.

- En caso de fallo en el acceso al filtro, no se produce penalización ya que se accede al filtro en un ciclo anterior al de lectura de memoria.

- Al emplear direccionamiento basado en el desplazamiento respecto a la cabeza de pila, el uso de etiquetas de dirección es inecesario y se puede prescindir de todo lógica asociativa.

- El filtro captura una vecindad $+/-\frac{N}{2}$ del puntero de pila. El diseño, no obstante, no permite emplear una vecindad arbitraria. Un estudio más pormenorizado de los accesos a pila permitiría ajustar su tamaño para hayar una configuración óptima.

- Al igual que algunas de las propuestas anteriores, la inicialización de los datos de la pila no genera tráfico adicional en la memoria.

Algunas de las principales aportaciones de este trabajo se han recogido en los siguientes artículos:

- [GAGPT08c] R. González Alberquilla, L. Piñuel, J. I. Gómez y F. Tirado. Measurement of Register Allocation Architectural Impact. En *ACACES*, L'Aquila, Italia, Julio de 2008.

- [GAGPT08a] R. González Alberquilla, L. Piñuel, J. I. Gómez y F. Tirado. Evaluación de Algoritmos de Asignación de Registros desde la Perspectiva de los Sistemas Empotrados. En *Jornadas de Paralelismo*, Castellón, España, Septiembre de 2008.

- [GAGPT08b] R. González Alberquilla, L. Piñuel, J. I. Gómez y F. Tirado. Evaluation of Register Allocators from an Embedded System Perspective. En *Conference on Design of Circuits and Integrated Systems*, Grenoble, Francia, Noviembre de 2008.

Bibliografía

[ALE02] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 39(2):59–67, February 2002.

[Ana01] Analog Devices. *ADSP-219x/2191 DSP Hardware Reference Manual*, Jul 2001. http://www.analog.com/UploadedFiles/Associated_Docs/41255245479508219lHardRef_1_1.pdf.

[BDR07] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of Register Coalescing. In *International Symposium on Code Generation and Optimization*, pages 102–114, Washington, DC, USA, 2007.

[Cha82] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–105, New York, NY, USA, 1982.

[DM82] David R. Ditzel and H. R. McLellan. Register allocation for free: The c machine stack cache. *SIGARCH Comput. Archit. News*, 10(2):48–56, 1982.

[Fre74] R. A. Freiburghouse. Register allocation via usage counts. *Commun. ACM*, 17(11):638–642, 1974.

[GA96] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996.

[GAGPT08a] R. Gonzalez-Alberquilla, J. I. Gómez, L. Piñuel, and F. Tirado. Evaluación de Algoritmos de Asignación de Registros desde la Perspectiva de los Sistemas Empotrados. In *Jornadas de Paralelismo*, Castellón, España, September 2008.

[GAGPT08b] R. Gonzalez-Alberquilla, J. I. Gómez, L. Piñuel, and F. Tirado. Evaluation of Register Allocators from an Embedded System Perspective. In *Conference on Design of Circuits and Integrated Systems*, Grenoble, Francia, November 2008.

[GAGPT08c] R. Gonzalez-Alberquilla, J. I. Gómez, L. Piñuel, and F. Tirado. Measurement of Register Allocation Architectural Impact. In *ACACES*, L'Aquila, Italia, July 2008.

[GMT05] Michael Geiger, Sally McKee, and Gary Tyson. Beyond Basic Region Caching: Specializing Cache Structures for High Performance and Energy Conservation. In *Hipeac 2005*, 2005.

[GRE+01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[HMS07] Alex Hemsath, Robert Morton, and Jan Sjodin. Implementing a Stack Cache. Technical Report, Rice University, June 2007.

[HP06] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 4th edition, 2006.

[HRT01] M. Huang, J. Renau, and J. Torrellas. L1 Data Cache Decomposition for Energy Efficiency. In *International Symposium on Low-Power Electronics and Design*, pages 10–15, Huntington Beach, California, August 2001.

[KAMG02] N. S. Kim, T. Austin, T. Mudge, and D. Grunwald. Challenges for architectural level power modeling. pages 317–337, 2002.

[Lip68] J. S. Liptay. Structural Aspects of the System/360 Model 85 II The Cache. *IBM Journal of Research and Development*, 7(1):15–21, 1968.

[LPMS97] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO 30: Proceedings of the 30th*

annual ACM/IEEE international symposium on Microarchitecture, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society.

[LSNT01] H. Lee, M. Smelyanskiy, C.Ñewburn, and G. Tyson. Stack Value File: Custom Microarchitecture for the Stack. In *International Symposium on High-Performance Computer Architecture*, pages 5–14, Monterey, Mexico, January 2001.

[NPP07] V. Krishna. Nandivada, Fernando. M. Q. Pereira, and J. Palsberg. A Framework for End-to-End Verification and Evaluation of Register Allocators. In *Lecture Notes in Computer Science*, pages 153–169, August 2007.

[PP05] F. MQ Pereira and J. Palsberg. Register Allocation Via Coloring of Chordal Graphs. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, pages 315–329, Tsukuba, Japan, November 2005.

[PS99] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, 1999.

[RSM+03] R. A. Ravindran, R. M. Senger, E. D. Marsman, G. S. Dasika, M. R. Guthaus, S. A. Mahlke, and R. B. Brown. Increasing the number of effective registers in a low-power processor using a windowed register file. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and*

synthesis for embedded systems, pages 125–136, New York, NY, USA, 2003. ACM.

[SPA92] SPARC International Inc. *The SPARC Architecture Manual*, 1992. http://www.sparc.com/standards/V8.pdf.

[Ten02] Tensilica Inc. *Xtensa Architecture and Performance*, Sep 2002. http://www.tensilica.com/pdf/xtensa_arch_white_paper.pdf.

[ZP05] X. Zhuang and S. Pande. Differential register allocation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 168–179, New York, NY, USA, 2005. ACM.

Índice de figuras

Índice de tablas