View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Tesis Doctorals en Xarxa

Computación Difusa

Computación Difusa

Carlos Álvarez

UPC. Universidad Politécnica de Cataluña
Barcelona, Diciembre 2006

Computación Difusa

Carlos Álvarez

Director de Tesis:
Mateo Valero Cortés

Co-director de Tesis:
Jesús Corbal San Adrián

A Chani que la hizo conmigo…
… y a mis padres que me hicieron a mi.

Agradecimientos

Esta tesis, como suele suceder, no es tan solo el resultado de mi trabajo, sino que más bien soy yo el recoge los frutos del trabajo de todo un grupo de gente. Primero y ante todo debo agradecer el que hoy esté escribiendo esto a Mateo Valero y Jesus Corbal. El primero ha sido mi director de tesis y probablemente una de las mejores personas que he conocido. El segundo ha sido mi codirector y un buen amigo. Muchas gracias a los dos, de verdad.

También me gustaría agradecerle el trabajo que dedicaron a esta tesis gente que, en principio, no tenía ninguna razón para involucrarse y, sin embargo, lo hizo. Muchas gracias a Jose A. R. Fonollosa que nos prestó amablemente su software y su tiempo y a Josh Fisher que apareció de la nada para decirnos lo buena que le parecía nuestra idea y darnos su apoyo.

Quería agradecer también su ayuda a mucha gente del departamento que simplemente estuvo ahí. Nunca me han gustado las lista de nombres, así que no pondré ninguna, pero creo que saben quienes son: mis compañeros de doctorado, de asignatura, de café y de algún que otro vídeo robado. Y, naturalmente a todo el soporte logístico, aquellos que siempre están ahí para nuestras prisas de última hora: a Alex y a todos los de sistemas y a Trini y toda la gente de secretaría. Por su paciencia.

Finalmente, muchas gracias a Francisco, por ayudarme con la banda sonora, a Miguel Angel, por las tertulias y a Montse por las tutorías. Muchas gracias a todos.

Este trabajo se ha realizado con el soporte del Ministerio de Educación bajo los proyectos CYCIT de High Performance Computing: Architectures, Compilers, Operating Systems, Tools and Algorithms (TIC98-0511, TIC2001-0995 y TIN2004-07739-C02-01) y de la red HiPEAC (European Network of Excellence in High Performance Embedded Architectures and Compilers).

Resumen

Esta tesis se enmarca en el ámbito de las técnicas de mejora de la eficiencia de ejecución (disminución del consumo y aumento de la velocidad) en el diseño de procesadores orientados a la ejecución de aplicaciones multimedia. En los últimos años la proliferación de los sistemas de baja potencia (móviles, PDAs, etc.) ha generado una enorme demanda de capacidad de cálculo en estos dispositivos. Los sistemas móviles de tercera generación ya empiezan a ser capaces de ejecutar aplicaciones multimedia que hasta ahora estaban restringidas a los sistemas de propósito general de sobremesa. Aún así, estos sistemas han de seguir enfrentándose al difícil reto de mejorar sus prestaciones manteniendo limitado su consumo.

Dentro del ámbito de las aplicaciones multimedia ejecutadas en sistemas de bajo consumo, se han propuesto muchas alternativas para aumentar la velocidad de ejecución. Sin embargo, finalmente, muchos de los diseñadores se han decidido por arquitecturas VLIW debido a su capacidad de obtener un buen rendimiento con una complejidad arquitectónica baja y un consumo de potencia razonable. Otro gran grupo de arquitecturas han optado por incluir en sus arquitecturas instrucciones con paralelismo a nivel de subpalabra, similares a las que se encuentran en las extensiones MMX.

En cualquier caso, se puede ver claramente que la evolución seguida por las arquitecturas de bajo consumo se aparta de la tendencia marcada años atrás por los procesadores de alto rendimiento. En el segmento de alta velocidad los aumentos significativos del rendimiento se han obtenido habitualmente a cambio de ejecutar más instrucciones por unidad de tiempo, muchas de ellas de forma especulativa. Así, técnicas recientemente propuestas como la predicción de valores, la ejecución especulativa o la preejecución,

no son aplicables al ámbito del bajo consumo.

En esta tesis se propone un novedoso sistema de cálculo para contenidos multimedia: **el cálculo difuso**. Este sistema permite aprovechar, por un lado, la redundancia de los contenidos multimedia y, por otro, la tolerancia respecto a los errores exhibida por los destinatarios de los contenidos (los sentidos humanos, es decir, nuestra propia percepción) para obtener grandes ganancias de tiempo y energía en el procesado de dichos contenidos.

Para conseguir los objetivos propuestos, el cálculo difuso se basa en un principio muy simple: para obtener un resultado correcto para una operación no es necesario realizar un cálculo exacto. O dicho de otro modo, no importa cometer errores si no hay nadie que pueda distinguirlos. Este principio, que aplicado a otros ámbitos (bases de datos, por ejemplo) puede parecer una aberración, cobra pleno sentido cuando se aplica a los contenidos multimedia. De hecho, hace muchos años que este principio se está empleando asiduamente, solo que a otro nivel: Los algoritmos de compresión más usuales (mp3 y jpeg concretamente) se basan en obtener resultados inexactos para lograr reducir la cantidad de información que es necesario almacenar o transmitir. En esta tesis proponemos hacer exactamente lo mismo pero para conseguir aumentar la velocidad de los procesadores y reducir la energía que consumen.

La implementación de un sistema de cálculo difuso se ha abordado desde dos perspectivas distintas: el cálculo difuso de instrucciones y el cálculo difuso de regiones. El cálculo difuso de instrucciones, como su mismo nombre indica, consiste en la substitución de una instrucción determinista (por ejemplo, una multiplicación) por otra instrucción que de lugar a un resultado aproximado. Esto se ha realizado, o bien mediante unidades funcionales que operan sobre un número menor de bits, o bien mediante un sistema de memorización tolerante.

El cálculo difuso de regiones, por su parte, consiste en la substitución de un fragmento de código completo por un mecanismo hardware-software que es capaz de generar un resultado aproximado para las mismas entradas que el código original.

Para evaluar estas propuestas se ha recurrido a la implementación de un simulador de reuso. Se ha partido del simulador SimpleScalar integrado con el medidor de energías Wattch y se ha procedido a introducir en él todos los elementos que eran necesarios para realizar las medidas requeridas. Se han añadido las diferentes tablas de reuso y se han configurado los saltos necesarios para medir exactamente la energía consumida, tanto por las instrucciones (o grupos de instrucciones) reusadas como por aquellas que no lo son. Además ha sido necesario mejorar de forma importante la integración

entre Wattch y SimpleScalar ya que aunque este último es ampliamente parametrizable, el primero no utilizaba estos parámetros para modificar su funcionamiento y por tanto daba lugar a medidas erróneas cuando se usaba con una configuración no estándar.

Los resultados obtenidos muestran que las ideas propuestas en esta tesis dan lugar a importantes ahorros de energía y tiempo en el procesado de contenidos multimedia sin modificar sustancialmente (es decir, de forma apreciable) la salida de dicho procesado. Se han obtenido ahorros del 15% en tiempo y del 25% en energía para un conjunto de programas multimedia con calidades de salida muy buenas.

En resumen, a lo largo de la tesis se han realizado las siguientes aportaciones:

1. Se ha propuesto una nueva forma de procesar datos: **el cálculo difuso**, que permite ahorrar tiempo y energía en el procesado de archivos multimedia.
2. Se han propuesto diferentes implementaciones de cálculo difuso: la memorización tolerante de regiones, la memorización tolerante de instrucciones y el uso de unidades funcionales reducidas. Todos estos sistemas pueden integrarse en un procesador actual.

Paralelamente, y como parte de las herramientas necesarias para elaborar este trabajo, se ha desarrollado un simulador de reuso que permite obtener resultados de tiempo y energía para un amplio catálogo de posibles sistemas de reuso. El simulador se ha configurado para tres tipos diferentes de procesadores: muy bajo consumo (reproductores mp3, móviles), bajo consumo (agendas digitales, UMPCs) y alto rendimiento (portátiles o sobremesa). Para todos ellos se han realizado pruebas con programas seleccionados multimedia ampliamente utilizados como *jpeg* o *lame* (codificador de mp3) y representativos de un amplio espectro de aplicaciones (audio, video, imagen o reconocimiento de voz). Con todos estos elementos se han evaluado las diferentes propuestas obteniendo buenos resultados de mejora de rendimiento y ahorro de energía, demostrando así la validez de la idea original.

Índice

1 Introducción **1**

1.1 Motivación 2
1.2 Visión general de la tesis 3
1.2.1 Cálculo difuso de instrucciones. 4
1.2.2 Cálculo difuso de regiones. 5
1.2.3 Obtención de resultados. 6
1.2.4 Conclusiones obtenidas. 6
1.3 Estructura de la tesis 7

2 Sistemas multimedia **9**

2.1 Importancia de los sistemas multimedia. 10
2.1.1 Pero ¿qué es multimedia? 10
2.1.2 ¿Por qué es diferente el procesado multimedia? 11
2.2 Propuestas para el procesado multimedia. 12
2.2.1 Las instrucciones SIMD. 13
2.2.2 Nuevas instrucciones de Prefetch. 14
2.2.3 Coprocesadores dedicados. 14
2.2.4 Arquitecturas vectoriales. 15
2.2.5 Arquitecturas VLIW. 15

3 La propuesta de trabajo **17**

3.1 El cálculo difuso. 18
3.1.1 Los errores. 20
3.2 El entorno de trabajo. 24
3.2.1 Argumentación. 24

3.2.2 Modelo de arquitectura. 25
3.2.3 El entorno de simulación. 28

4 Cálculo difuso de instrucciones 31

4.1 Introducción. 32
4.1.1 Trabajo Relacionado. 32
4.1.2 Ámbito de trabajo 34
4.2 El estándar IEEE754. 35
4.3 Los programas de pruebas. 37
4.4 Unidades funcionales difusas. 37
4.4.1 Metodología. 37
4.4.2 Resultados. 41
4.5 Reuso aproximado. 45
4.5.1 Sistemas de reuso clásico aplicados a las operaciones de punto flotante en multimedia. 45
4.5.2 Nuestra propuesta: el reuso tolerante. 50
4.6 Unidades funcionales difusas y reuso. 61
4.7 Conclusiones. 66

5 Cálculo difuso de regiones 69

5.1 Introducción. 70
5.2 Trabajo relacionado. 70
5.3 Los programas estudiados. 71
5.4 Reuso de regiones para multimedia. 72
5.5 Hardware para el reuso de regiones. 73
5.5.1 Resultados del reuso de regiones. 79
5.5.2 Conclusiones sobre el reuso de regiones. 88
5.6 El reuso tolerante de regiones. 89
5.6.1 Hardware para el reuso tolerante. 89
5.6.2 Modificaciones en el ISA. 90
5.6.3 La tolerancia, los aciertos y el error. 91
5.6.4 Resultados del reuso tolerante de regiones. 98
5.6.5 Ajuste dinámico de la tolerancia. 103
5.7 Conclusiones. 116

6 Conclusiones y extensiones futuras 121

6.1 Objetivos y motivaciones. 122
6.2 Aportaciones y conclusiones. 122
6.3 Extensiones futuras. 124

Bibliografía 127

A Uso del simulador SimpleReuse **135**
A.1 Introducción 136
A.2 El simulador SimpleReuse. 136
A.3 Uso del simulador. 137
A.3.1 Reuso de regiones. 138
A.3.2 Reuso de instrucciones. 141
A.4 Conclusiones y extensiones futuras 144
A.5 Código de filtrar.c 145
A.6 Ejemplos de ficheros de configuración. 148
A.6.1 Configuración para procesador de ancho 1 en orden. . 148
A.6.2 Configuración para procesador de ancho 2 en orden. . 151
A.6.3 Configuración para procesador de ancho 4 fuera de orden. 153

Lista de Figuras **157**

Lista de Tablas **161**

1

Introducción

Resumen

En este capítulo se explica la motivación y razón de ser de esta tesis fundamentando las bases que han llevado a enfocar este trabajo al procesado multimedia. A continuación se muestra una visión general de la misma detallando sus aspectos más importantes y principales aprotaciones. En el siguiente punto se introduce el concepto de cálculo difuso y su posible aplicación tanto a instrucciones individuales como a grupos (o regiones) de estas. Finalmente se estructura el resto del escrito.

1.1 Motivación

A lo largo de los últimos años hemos vivido un crecimiento exponencial de los sistemas multimedia. Móviles, agendas electrónicas, reproductores de música y vídeo y un sinfín de nuevos aparatos invaden nuestras vidas. Todos ellos se caracterizan por componerse de un procesador relativamente sencillo (y a poder ser de bajo consumo) y una serie de programas de procesado multimedia principalmente.

¿Por qué programas específicamente de procesado multimedia? Bien, quizás la acepción más exacta no sea esta, pero en el mundo actual se ha dado en llamar multimedia a todos aquellos sistemas (electrónicos principalmente) que tienen como origen o destino la comunicación con los usuarios en un lenguaje afín a estos últimos. Así, es multimedia un reproductor de música, un grabador de voz, un visor de vídeos y una cámara de fotos, pero también lo es un reconocedor de escritura, un reconocedor de voz o un sintetizador.

Ha sido principalmente esta característica de interacción en un lenguaje afín la que ha propiciado el gran auge de lo multimedia. Los sistemas multimedia son fáciles de manejar porque se comunican de una forma sencilla para nosotros (la misma que ya usábamos para comunicarnos con otras personas) de forma que la gran mayoría de la población puede acceder rápidamente a estos dispositivos y usarlos sin necesidad de tener ningún conocimiento técnico. Esta ventaja, por otro lado, también es la razón de que el ciclo no se agote. Una vez que hemos descubierto lo fácil que es tomar, manipular y enviar una foto, queremos hacer lo mismo con los vídeos. Y además, queremos llevarlos encima para poder enseñarselos a los amigos. Y lo mismo con nuestra música favorita y próximamente con las películas. Por si fuera poco, también nos gustaría poder olvidarnos de recargar los aparatos cada pocos días.

Estas observaciones se pueden extrapolar a los diferentes segmentos que pueblan el mercado: desde los UMPCs [UMP06, Pen06] y PVPs [PVP06] de alto rendimiento, pasando por los dispositivos intermedios tales como videoconsolas, PDAs o GPSs [PSP, PDA06], hasta llegar al mercado realmente portable (teléfonos [Mob] y reproductores de música portátiles [PMP06]), todos tienen en común el procesado multimedia *de los mismos ficheros* con diferentes niveles de complejidad.

En este contexto resulta evidente que los procesadores siempre se quedan cortos en prestaciones. Por mucho que se haya avanzado en el desarrollo de sistemas de bolsillo capaces de realizar tareas impensables hace solo unos años para los sistemas de sobremesa, siguen siendo insuficientes... y consumiendo demasiado.

1.2 Visión general de la tesis

Esta tesis se ha realizado teniendo como objetivo este contexto de una necesidad cada vez más creciente de procesadores multimedia portátiles que sean capaces de aumentar el rendimiento mientras mantienen acotado el consumo. Hace tiempo que ha resultado evidente que no es posible que este tipo de procesadores sigan el mismo camino que los procesadores de altas prestaciones. Las recientes aportaciones a este tipo de procesadores, como la predicción de valores o la inclusión de colas de miles de instrucciones, incrementan el rendimiento a base de ejecutar cada vez más instrucciones por unidad de tiempo. De estas instrucciones, un porcentaje cada vez más significativo es descartado a cambio de que el número de instrucciones ejecutadas útiles aumente. El principal inconveniente de esta aproximación es que el consumo aumenta más que el número de instrucciones ejecutadas, de forma que el rendimiento global (teniendo en cuenta la energía) tiende a empeorar aunque la velocidad de proceso aumente.

En este sentido los procesadores multimedia han tomado dos caminos principales: por un lado han tendido hacia la explotación de arquitecturas tipo VLIW ya que estas presentan una alta relación entre capacidad de proceso y potencia consumida, mientras que por otro, han tendido a incluir instrucciones con paralelismo de subpalabra, tipo MMX.

En esta tesis se propone una aproximación diferente al problema de la velocidad y la potencia: **el cálculo difuso**. Esta novedosa propuesta consiste en incluir en los procesadores un hardware capaz de realizar cálculos aproximados a las operaciones más comunes del procesado multimedia, tanto a nivel de instrucciones como de grupos de instrucciones. Este hardware debe, además, ser más rápido que el hardware original usado para este tipo de procesado y, en el mejor caso, consumir menos energía para realizar el cálculo. Como estamos hablando de sistemas multimedia, es importante hacer notar que la salida tiene como objetivo interactuar con los sentidos humanos, de forma que una diferencia no perceptible por los sentidos humanos no es importante[1]. Así pues, a efectos prácticos, nuestra propuesta permite ahorrar energía y tiempo en los sistemas de bajo consumo. Es importante observar que el principio de intercambiar precisión por otro factor no es nuevo, ya que se encuentra implementado en gran cantidad de sistemas compresores de datos de amplia difusión (JPEG, MP3, MPEG2...). Lo que es totalmente novedoso es el echo de aplicar el intercambio a la energía e implementarlo en hardware.

[1] Esto no pasa, evidentemente, con otro tipo de aplicaciones como, por ejemplo, el cálculo de nóminas.

Para estudiar la viabilidad de este nuevo tipo de sistema de cálculo a la hora de aplicarlo a sistemas reales se ha estudiado si existía la posibilidad de implementar el hardware necesario. Además se ha estudiado si dicho hardware obtenía mejores resultados que el sistema sin el nuevo hardware (no olvidemos que deseamos obtener avances tanto en tiempo de proceso como en consumo de energía por lo cual los sistemas propuestos se encuentran muy limitados, en la práctica, en cuanto a tamaño y complejidad).

1.2.1 Cálculo difuso de instrucciones.

La forma más sencilla de cálculo difuso es el cálculo difuso de instrucciones. Existen muchos tipos de instrucciones, pero resulta evidente que una técnica como la propuesta debe restringirse a operaciones que:

- Operen solo con los datos del programa, ya que los valores usados para el control no son tolerantes a errores (son utilizados por el procesador que no se caracteriza por su imaginación).
- Consuman suficiente energía como para dejar un margen suficiente de ganancia a un sistema alternativo.

Ambos puntos son bastante restrictivos. El primer punto implica diferenciar las operaciones de datos de las de control. Hay que tener en cuenta que, en este sentido, por datos queremos expresar exclusivamente el sonido o la imagen del contenido multimedia, no lo que habitualmente se entiende como datos del programa. Es importante ser conscientes de que los procesadores, al menos de momento, no son capaces de tolerar errores, por ejemplo, en el índice de un bucle. Además tampoco son capaces de diferenciar a priori si una instrucción afecta a dicho índice o a, por ejemplo, el color de un punto en pantalla.

La segunda restricción también es difícil de solventar. Ya ha habido diferentes trabajos que demuestran que, por ejemplo, la mayoría de instrucciones reusables de un programa son más rápidas y baratas de calcular que de reutilizar ya que es necesario un hardware adicional para poder llevar a cabo el reuso. Hemos de tener muy en cuenta esta restricción a la hora de escoger las instrucciones objeto de estudio en esta tesis y la forma de poder ahorrar tiempo en ellas ya que un nuevo hardware implica, por un lado, utilizar más superficie, y por otro, mayor gasto de energía aunque no se use, debido a las inevitables pérdidas de corriente en los transistores.

Teniendo en cuenta estas restricciones en esta tesis se han propuesto dos métodos distintos de cálculo difuso orientados ambos al mismo tipo de

instrucciones: las instrucciones de coma flotante. Este tipo de instrucciones cumple ambos requisitos ya que, por un lado, son instrucciones que suelen operar solo con los datos (es difícil que se programe un bucle con un índice decimal) y, además, son instrucciones costosas de computar, tanto en tiempo como en energía.

Las propuestas realizadas para este tipo de instrucciones incluyen:

- Un sistema de memorización tolerante, es decir, un sistema de reuso mediante tablas que reusa una entrada de la tabla aunque los valores de entrada a la operación no sean exactamente los mismos que en la instrucción original memorizada.
- El uso de unidades funcionales que operan sobre una menor cantidad de bits.

Aunque las ideas de reusar instrucciones o de modificar las unidades funcionales no son nuevas, si lo es el planteamiento que toman en esta tesis ya que se aplican de forma inexacta. Resultado de este trabajo han sido los artículos [ACSV02, ACS$^+$03, ACV05] que muestran como aplicar el reuso tolerante de instrucciones a los programas multimedia de punto flotante. Actualmente, además, estamos preparando un artículo con nuestra propuesta de un nuevo tipo de datos para punto flotante.

1.2.2 Cálculo difuso de regiones.

El siguiente punto que se evaluó fue la posibilidad de realizar calculo difuso de regiones completas de código. En este caso se trata de encontrar alternativas de cálculo capaces de recibir las mismas entradas que la región a substituir y obtener salidas similares, pero a un menor coste computacional. Debido a que las regiones de código contienen más de una instrucción, es más fácil obtener ganancias energéticas, ya que tenemos más margen de maniobra, pero, en contra, tenemos mayor dificultad en obtener alternativas lo suficientemente flexibles como para adaptarse a la gran variedad de algoritmos que puede encerrar una región de código. Además, deberemos escoger, como en el caso de las instrucciones, regiones que operen estrictamente sobre la información multimedia, así que para estas aplicaciones necesitaremos la ayuda del compilador (y por lo tanto la técnica propuesta no será puramente hardware) para determinar las regiones susceptibles de ser substituidas.

Para poder aproximar el resultado de regiones completas de código se ha recurrido a la memorización de regiones que cumplieran los siguientes requisitos:

- Fueran regiones pertenecientes a los núcleos del procesado de forma que se ejecutaran dinámicamente mucho a lo largo de todo el procesado.
- Tuvieran una cantidad limitada de valores de entrada y salida del programa, de forma que se pudieran memorizar con tablas no excesivamente anchas.
- Tuvieran un solo punto de entrada y salida.

Nuestras propuestas de cálculo difuso de regiones se han publicado en [ACSV01]. Siguiendo exactamente nuestras ideas ha sido publicado también [CH05], con un análisis sencillo de nuestras ideas de reuso de regiones aplicadas a MPEG2. Actualmente estamos preparando, asimismo, un último artículo que recoge los resultados de reuso de regiones presentados en esta tesis.

1.2.3 Obtención de resultados.

Para realizar la evaluación de la viabilidad de las propuestas se utilizó en primer lugar un sistema de instrumentación. Los códigos a evaluar se ejecutaban monitorizados y se substituía el código original por el nuevo código generado mediante rutinas programadas al efecto. Con este esquema pudo evaluarse la validez del enunciado en cuanto a resultados, es decir, se comprobó que la salida tenía una calidad suficiente y que existía suficiente código substituible como para que hubiese un ahorro significativo de tiempo y energía.

Una vez realizada la validación se pasó a un entorno de simulación para evaluar en detalle el hardware propuesto para las diferentes alternativas. Se implementó un simulador que incorporaba dicho hardware y que realizaba medidas de tiempo y energía sobre los programas de prueba. En este entorno se realizaron las medidas de tiempo de ejecución y consumo de energía del sistema. Además se comprobó de nuevo la calidad de las salidas de datos corroborandos los resultados ya obtenidos.

1.2.4 Conclusiones obtenidas.

Las principales conclusiones obtenidas con este trabajo han sido:

- La propuesta de cálculo difuso es viable para el procesado de contenidos multimedia.

- Se puede implementar cálculo difuso hardware a nivel de instrucciones del procesador.
- Se puede implementar el principio del cálculo difuso a nivel de regiones mediante un sistema combinado de software y hardware.
- En cualquiera de ambas aproximaciones se ha demostrado que un sistema difuso es capaz de obtener importantes beneficios en tiempo de ejecución y ahorro de energía, tanto frente al programa original como frente a sistemas de reuso clásicos.

1.3 Estructura de la tesis

Esta tesis está organizada de la siguiente forma: el capítulo 2 describe la importancia del procesado multimedia y las principales aportaciones realizadas en este campo por otros trabajos previos.

El capítulo 3 describe los principios básicos del cálculo difuso y sus fundamentos teóricos. También se describen las herramientas usadas para evaluar nuestra propuesta.

El capítulo 4 explica como se aplica el cálculo difuso al procesado de instrucciones. Se proponen dos vías de aplicación distintas, el uso de unidades funcionales difusas y el reuso tolerante de instrucciones y se evalúan ambas propuestas.

El capítulo 5 se realiza una propuesta de implementación para el cálculo difuso de regiones. Dicha propuesta se complementa con un sistema dinámico de cálculo de la tolerancia y se evalúan las diversas alternativas comparándolas con un sistema clásico de reuso de regiones.

Finalmente, el capítulo 6 presenta un resumen de resultados y conclusiones junto con las posibles extensiones futuras de este trabajo.

2

Sistemas multimedia

Resumen

En este capítulo empezaremos intentando definir el concepto de programa multimedia. A continuación describiremos la importancia de este tipo de programas y las características que los hacen distintos a del resto de las aplicaciones existentes, la primera de las cuales es que son programas destinados a interactuar con el ser humano. Se describen además otras propuestas recientes para el procesado de contenidos multimedia como las instrucciones SIMD, los procesadores vectoriales o los procesadores VLIW. Finalmente se resumen sus principales características y deficiencias.

2.1 Importancia de los sistemas multimedia.

Es indiscutible que los sistemas multimedia gozan de una gran popularidad. Basta con mirar a nuestro alrededor para darnos cuenta de que todo el mundo usa sistemas multimedia. Es más, si examinamos nuestros bolsillos, bolsos, mochilas o carteras es muy probable que descubramos uno, o varios, aparatos de este tipo.

2.1.1 Pero ¿qué es multimedia?

En un principio, la palabra multimedia, etimológicamente, quiere decir varios medios. Es decir, sería la utilización coherente de diversos medios (de comunicación) para presentar una única información. Así pues, una película es multimedia (ya que incluye imagen y sonido), pero, estrictamente hablando, una imagen no lo sería, ni tampoco un mensaje sonoro (una canción o una grabación).

Sin embargo, en el mundo de la informática, bajo el término multimedia se han agrupado toda una serie de diversos programas que tratan con la información en diversas vertientes. En [RAJ99] se define “media processing” como aquel procesado que trata con información digital multimedia, pero esto no es tampoco demasiado preciso. En realidad, actualmente se usa el término multimedia para englobar los sistemas que ayudan a la interacción entre el hombre y el computador. Así pues, se llama multimedia a cualquier programa que realice tareas en los ámbitos de la imagen, del sonido, del vídeo, de los gráficos 3D, de la voz, etc. Y estas tareas pueden ser de grabación, reproducción, procesado, encriptado, codificación y un sinfín más de posibilidades.

Lo que ha convertido “lo multimedia” en un genero tan atractivo, es el hecho (ya observado por [DD97]) de que los usuarios enseguida comprendieron las enormes posibilidades que les brindaban estos sistemas. De repente comprender y usar los computadores era mucho más sencillo gracias a las nuevas formas de presentar la información, y por tanto todos podíamos hacerlo. Este factor incrementó mucho la productividad de estos sistemas y, por qué no decirlo, la diversión que eran capaces de aportarnos. Es un segmento que crece exponencialmente desde hace una década y que, de momento, parece no tener fin (al fin y al cabo, hace unos años era raro que el coche nos hablara, y ahora empieza a ser normal hablarle nosotros a él...).

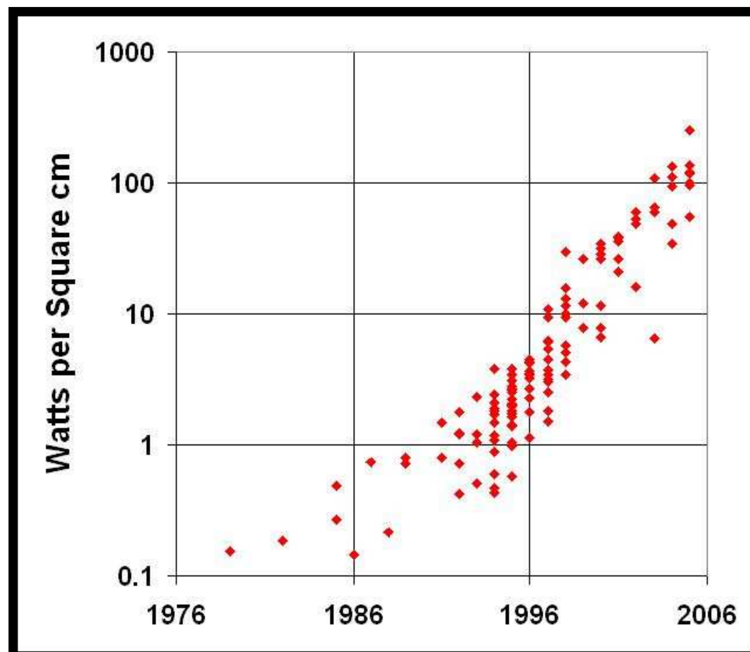


Figura 2.1: Gráfico de consumos por cm^2 de procesadores reales.

2.1.2 ¿Por qué es diferente el procesado multimedia?

El mundo del procesado multimedia es extremadamente dinámico. Constantemente surgen nuevas demandas en forma de nuevos algoritmos y nuevas utilidades que los sistemas actuales se ven en apuros para poder procesar. Actualmente se espera que los procesadores sean capaces de trabajar con videoconferencias, grabar y procesar vídeos, juegos en 3D, reconocimiento y sintetización de voz, etc. El ritmo de crecimiento de las aplicaciones es imparable.

Por si todo lo anterior fuera poco, el tipo de procesadores que se utiliza para el procesado multimedia no es el mismo que el usado para los sistemas de sobremesa. El procesador multimedia habitualmente se espera que sea portátil y esté imbuido en un teléfono, en una agenda electrónica (PDA), en una cámara o en otros lugares donde no solo existe un problema de tamaño, sino también de cantidad de energía disponible. No solo queremos hacer procesado multimedia sino que queremos hacerlo con un par de pilas en la playa y, a poder ser, que duren una semana, ya que sino el sistema resulta incómodo.

Todas estas restricciones hacen que los procesadores multimedia no puedan seguir el camino que marcaron hace años los procesadores de sobremesa. En

este tipo de procesadores, las últimas innovaciones siguen el camino de incorporar sistemas cada vez más complejos para aumentar el rendimiento. La predicción de valores [LS96], la ejecución especulativa [SM98, AD98], cada vez más agresiva, o la preejecución [RS01] son técnicas que incrementan el rendimiento a cambio de consumir cada vez más energía. En la gráfica 2.1[1] se puede ver el consumo de energía de diferentes procesadores y se puede observar que, de seguir al ritmo actual, el consumo dentro de poco subirá a niveles insostenibles. Evidentemente, si hablamos de procesadores que deben alimentarse mediante pilas o pequeñas baterías, este efecto se vuelve aún más evidente.

¿Cómo ser capaces de mantener el ritmo actual de crecimiento en capacidad de proceso sin aumentar el consumo? Se suele argumentar que un sistema consciente del consumo es todo aquel que incrementa el rendimiento en mayor medida que el consumo según la ecuación $f = v^2$. La base de esta línea de pensamiento es que, si conseguimos ir mucho más rápido, aunque se consuma un poco más de potencia, el resultado neto es beneficioso, ya que siempre podemos reducir la frecuencia de trabajo y consumir menos. Sin embargo, esto tiene dos inconvenientes. El primero y más significativo de ellos es el área ocupada. Un procesador de sobremesa actual ocupa unos 4 cm^2, un área imposible de imbuir en la mayoría de los teléfonos móviles actuales. El segundo inconveniente es que, en el mercado de sobremesa, esta aproximación es beneficiosa ya que tenemos margen en ambas vertientes del problema: es decir, queremos ir más rápido y, además, podemos permitirnos gastar un poco más ya que siempre podemos mejorar algo la refrigeración. Así, una técnica que se mantenga por debajo de la formula ya genera resultados positivos. En el mercado concreto en el que nosotros vamos a trabajar dicho margen es más estrecho: queremos procesar más rápido pero, gastando lo mismo o menos, nunca más, ya que la duración de las baterías debe aumentar o mantenerse, no disminuir. Si tenemos en cuenta ambos factores (consumo y rendimiento) a la vez, muchas técnicas de probada eficacia en el terreno de las altas prestaciones resultan en un bajo beneficio conjunto, de forma que, en muchos casos, no merece la pena la complejidad que introducen en el diseño.

2.2 Propuestas para el procesado multimedia.

A continuación vamos a presentar algunas de las propuestas más importantes realizadas para el procesado de información multimedia en los últimos años. Hemos intentado presentar las ideas más exitosas y que ofrecen un

[1]Fuente: Peter M. Kogge, in International Conference on High Performance Computing, Networking and Storage, SC06, Tampa, Florida, 2006.

mayor rendimiento y no procesadores concretos, aunque hemos incluidos en casi todas una lista de los procesadores donde se han aplicado. Es importante comentar que aunque algunas de estas propuestas (instrucciones SIMD, prefetch) son de propósito general se han incluido aquí ya que ofrecen un rendimiento excepcional cuando se aplican a procesado multimedia.

2.2.1 Las instrucciones SIMD.

Una de las arquitecturas más adecuadas para el procesado multimedia ha sido la inclusión, en procesadores de propósito general, de instrucciones SIMD (Single Instruction Multiple Data) [Pen06, Sem99]. Estas instrucciones aprovechan el paralelismo a nivel de datos (DLP) inherente a los programas multimedia ya que permiten codificar diversas operaciones en paralelo. Además resultan especialmente adecuadas para multimedia ya que los bucles de los núcleos de las aplicaciones multimedia suelen ser cortos.

El gran problema de estas operaciones es su falta de escalabilidad ya que, ni se puede hacer crecer indefinidamente el registro sobre el que opera la instrucción (y que es el encargado de almacenar los diversos datos), ni los códigos multimedia poseen una gran cantidad de datos paralelos en el último nivel de sus bucles, ya que estos suelen recorrer cantidades cortas de datos: 4, 8 o 16 típicamente. Las reducciones tambiéns son otro de los caballos de batalla de este tipo de extensiones [CSEV99, JCV01].

Otro problema de estas instrucciones es que los algoritmos multimedia suelen estar optimizados para ejecutarse en sistemas superescalares puros. La transformada DCT, por ejemplo, tiene como base una matriz de tamaño 8×8, A, que permite obtener una matriz de salida C a partir de: $C = A \cdot B \cdot A^T$, donde A^T indica la matriz traspuesta de A y B es el bloque de entrada. Esto implicaría en principio 1024 multiplicaciones para cada bloque de entrada B. Sin embargo, ha habido numerosa investigación en el tema ya que es una operación muy común, y actualmente se conocen algoritmos mucho más eficientes para obtener el resultado. Como ejemplo, el estándar JPEG solo utiliza 192 multiplicaciones en su algoritmo DCT, pero este algoritmo no es vectorizable.

Los ejemplos de sistemas con instrucciones SIMD son numerosos: MMX de Intel [PW96, MMX98], VIS de Sun [TONH96], MDMX de Mips [MDM97]. En el mercado de los procesadores DSP (Digital Signal Processor) [TI99] también hay distintos casos, como Trimedia de Philips [Sem99] y TigerSHARC de Analog Devices [Dev]. Todos estos sistemas se basan en la idea de que los datos multimedia son pequeños (muchos de ellos de 8 a 16 bits) y enteros. Sin embargo, la necesidad de realizar cálculos para gráficos 3D

y otro tipo de aplicaciones, llevó a varios fabricantes a crear instrucciones SIMD para datos en coma flotante de simple precisión. Ejemplos como Altivec de Motorola [Mot98, NJ99, DDHS00], 3DNow! de AMD [3DN99] y SSE de INTEL [htt00] incluyen registros especiales de 128 bits e instrucciones SIMD de coma flotante de simple precisión de 32 bits.

2.2.2 Nuevas instrucciones de Prefetch.

Este tipo de instrucciones también se pueden encontrar en las extensiones Altivec [Mot98, DDHS00], 3DNow! [3DN99] y SSE [htt00] y permiten la precarga de datos aprovechando la naturaleza secuencial de los algoritmos multimedia (que acostumbran a realizar las mismas operaciones de forma consecutiva sobre todos los datos de entrada). Muchas de estas instrucciones permiten, además, no polucionar la memoria caché de nivel 1 ya que precargan directamente desde la caché de nivel 2.

2.2.3 Coprocesadores dedicados.

Otro de los caminos comunes que se han encontrado para solucionar las necesidades específicas de capacidad de procesado multimedia es el uso de coprocesadores dedicados. Aunque hay diversos campos donde se han utilizado, el más representativo es, sin duda, el de los sistemas de ayuda a la visualización 3D. Así, aunque los principales fabricantes hayan incluido en sus arquitecturas extensiones SIMD para el cálculo en punto flotante (como se acaba de explicar), los sistemas de sobremesa incluyen aceleradoras gráficas como, por ejemplo la NVidia GeForce [Nvi01], la 3DFX VooDoo3 [Int00] o la ATI Rage Radeon 8500 [ATI01] para sistemas PC o la Neon de Compaq [MMG+99] que usaban los sistemas Alpha.

Todos estos sistemas incorporan hardware específico que permite a los sistemas de sobremesa alcanzar la capacidad de proceso que requieren las aplicaciones gráficas de última generación. Esta aproximación, sin embargo, por sus mismas características no es posible en sistemas de bajo consumo ya que este tipo de coprocesadores consumiría más que el propio procesador del sistema. Sin embargo, si que ha habido ejemplos de coprocesadores en este mercado, como por ejemplo el coprocesador vectorial del ARM, que por sus mismas características podría entrar dentro de la siguiente sección.

2.2.4 Arquitecturas vectoriales.

Una solución típica para conseguir una gran capacidad de cálculo en sistemas portátiles que permite, a la vez, mantener bajo el consumo, es el uso de arquitecturas vectoriales. Ejemplos de este tipo de procesadores pueden ser el Torrent T0 [oCaBI95] o el V-IRAM [oCaB02].

Estos procesadores, sin embargo se enfrentan a diversos problemas que los limitan. Por un lado todos los que se han enumerado para sistemas SIMD (poca escalabilidad y optimización de los algoritmos) que les afectan en mayor medida aún que a estos. Por otro lado está el problema de la ley de Amdahl que afecta especialmente a los sistemas vectoriales. Los algoritmos multimedia no responden bien a los sistemas vectoriales debido a que habitualmente tenemos numerosos bucles anidados con unos pocos recorridos cada uno.

Para solucionar estos problemas, se han propuesto arquitecturas mixtas superescalares y vectoriales que combinan lo mejor de ambos mundos [QCEV99, QCEV01], incluso con extensiones de varias dimensiones [CVE99, CEV99, CEV01, CEV02]. Estas extensiones consiguen aumentar de forma significativa el rendimiento de los procesadores para aplicaciones multimedia, mucho más que aumentando el ancho del procesador. Otra solución alternativa similar ha sido ubicar la arquitectura vectorial en un coprocesador adyacente (como en el caso de ARM [PMP06]). Sin embargo, en estos casos estamos hablando de añadir, como mínimo, una nueva vía de ejecución al sistema base (superescalar) lo cual es una buena alternativa para los procesadores de alto rendimiento (como puede ser el caso del Tarantula [EAE+02]), pero no para los de bajo consumo.

2.2.5 Arquitecturas VLIW.

Debido a sus características, los procesadores VLIW [Sem99, TI99] han sido quizás, la elección mayoritaria para los sistemas multimedia de bajo consumo. La cualidad de estos procesadores de mover la complejidad hacia el compilador y mantener el procesador simple y efectivo ha sido su mejor baza.

Ejemplos relevantes de este tipo de procesadores son ManArray de BOPS [BOP99] o Trimedia de Philips [Sem99]. Su mayor cualidad es la capacidad de ejecutar varias instrucciones en paralelo sin ningún coste de hardware de selección ya que la tarea de organizar el trabajo la lleva a cabo el compilador. De esta forma, el procesador puede centrarse en simplemente ejecutar las instrucciones, es decir, solo hace trabajo real. Recientemente ha habido

propuestas de ampliar este tipo de procesadores con extensiones vectoriales [SV05].

Si hubiera que poner alguna pega a este tipo de procesadores sería, precisamente, su punto fuerte: el compilador. Este tipo de procesadores dependen para su rendimiento totalmente del compilador. Dado que la tarea de éste muchas veces no es todo lo eficiente que sería de desear, la asistencia humana se vuelve imprescindible y, por tanto, el tiempo de desarrollo de las aplicaciones para estos sistemas se hace demasiado alto para el mercado cuando no hablamos de tareas muy específicas.

3

La propuesta de trabajo

Resumen

En este capítulo se introduce el concepto de cálculo difuso que se ha estudiado a lo largo de la tesis. Se comentan sus ventajas y deficiencias así como los sistemas utilizados para medir los resultados de los experimentos realizados mediante estas nuevas técnicas. Se explican, además, el entorno usado para el desarrollo de las pruebas, las carácterísticas de configuración del sistema de simulación y las diferentes herramientas utilizadas para realizar las medidas.

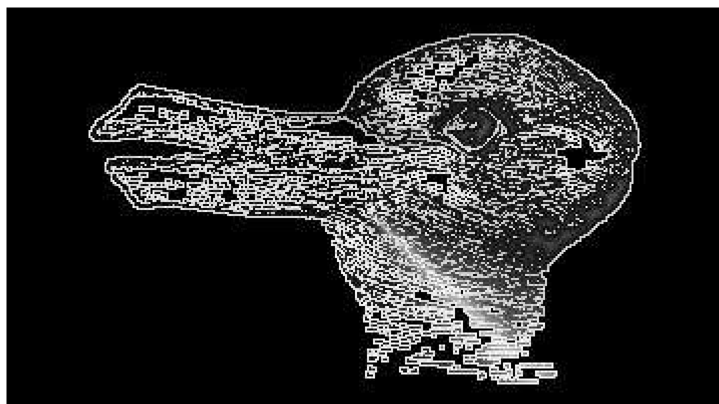


Figura 3.1: La percepción es traicionera.

3.1 El cálculo difuso.

Esta tesis se basa en la propuesta y estudio de un nuevo sistema de cálculo para aplicaciones multimedia: el cálculo difuso. Este sistema consiste, principalmente, en aprovechar el hecho de que el objetivo final de las aplicaciones multimedia es la comunicación con las personas y, por lo tanto, la salida de dichas aplicaciones tiene como destino ser percibida por los sentidos humanos.

A diferencia de lo que ocurre con los sistemas informáticos, los sentidos humanos tienen una alta tolerancia a los errores. Esta tolerancia se debe a dos características bien diferenciadas. Por un lado existen los limites perceptivos, es decir, somos incapaces de percibir ciertos tonos de luz y de oír ciertos sonidos debido a la propia configuración de nuestros oídos y ojos. Estos límites varían de persona a persona, pero aún así se suelen mover dentro de un margen bastante conocido. Así pues, es inútil un sistema de megafonía que procese ruidos por encima de los 50 KHz, o un sistema de visión que emita señal en el espectro infrarrojo. Toda esta información es superflua para nosotros, dado que no somos capaces de captarla, y puede por tanto eliminarse sin problemas del sistema.

Pero, además de nuestras limitaciones, existe otro factor aún más importante que es la tendencia del cerebro humano a ajustar la percepción a la cognición, es decir, tendemos a ajustar lo que percibimos a aquello que ya conocemos. Fruto de esta característica son los típicos juegos perceptivos que nos permiten ver números en una agrupación de puntos de color o percibir, a la vez, en el mismo dibujo, un pato o una liebre como se puede ver en la figura 3.1. Si la figura 3.1 tuviese como objeto informarnos en vez de divertirnos sería inútil ya que no lo conseguiría de forma fiable. Una imagen

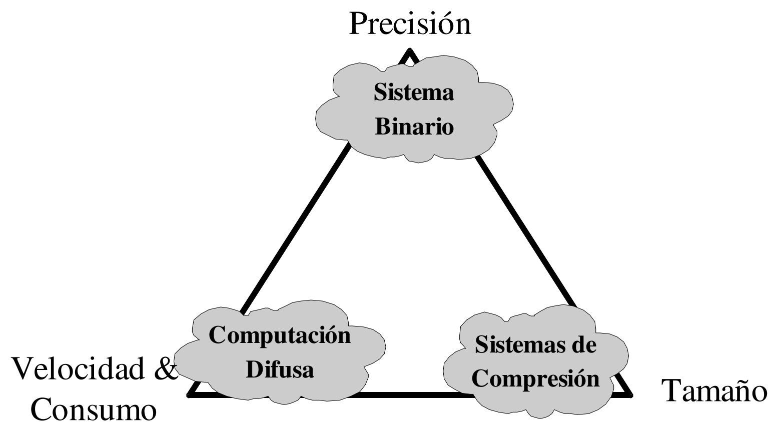


Figura 3.2: Base de la propuesta de la tesis.

quizás más simple y con menos información sería para nosotros más asimilable a uno solo de los animales del dibujo y por lo tanto para nosotros sería más útil. Es decir, el cerebro es capaz de rellenar los huecos si colocamos estos estratégicamente.

Es gracias a estas características que funcionan algoritmos ampliamente difundidos hoy en día como son JPEG, MP3 o DIVX. Estos algoritmos codifican las señales (imagen, sonido y vídeo, respectivamente) con un gran ratio de compresión debido a que son algoritmos que incorporan perdidas en las señales, es decir, no guardan la imagen original sino una copia aproximada. El quid de estos sistemas está en la forma en que consiguen la copia aproximada: no se limitan a distribuir las perdidas a lo largo de la señal sino que las concentran en aquellas características que nos resultan más fáciles de readaptar. Por poner un ejemplo, el algoritmo JPEG codifica con 8 veces más información la intensidad luminosa que el color ya que este último nos es muy fácil de ajustar a la realidad (hace años, con las televisiones en blanco y negro, la gente sabía perfectamente que color era un determinado gris).

En esta tesis se propone aprovechar estas propiedades de las señales multimedia y de la percepción humana para aumentar el rendimiento de los procesadores, queriendo decir por aumentar el rendimiento, conseguir que realicen el procesado más rápidamente y gastando menos energía en el proceso. En la figura 3.2 se pretende ilustrar este intercambio. Hasta ahora los algoritmos de compresión de datos se movían tan solo a lo largo de la línea que permite intercambiar precisión por tamaño. Nosotros proponemos añadir una nueva dimensión al intercambio e intercambiar precisión por velocidad y energía. En este sentido es importante comentar que esta nueva dimensión es ortogonal a la anterior, es decir, el hecho de ganar velocidad y

energía no implica en ningún caso comprimir menos las señales o incorporar mayores perdidas, es una ganancia adicional que obtiene el hardware.

Otra observación importante tiene que ver con las propiedades del intercambio. En los sistemas de compresión podemos esperar compresiones altas a cambio de pequeñas perdidas de precisión (en el caso de JPEG, por ejemplo, compresiones 25 a 1 con una calidad de imagen alta), pero a medida que las perdidas aumentan, el algoritmo satura y así, perdidas moderadas solo aumentan un poco la compresión (nuevamente en JPEG, si bajamos la calidad a baja, la compresión solo aumenta a 35 a 1). En nuestra propuesta deberemos esperar el mismo tipo de comportamiento. Pequeñas pérdidas iniciales nos permitirán obtener buenas ganancias en velocidad y energía, pero más perdidas adicionales tan solo ocasionarán aumentos marginales de los beneficios.

3.1.1 Los errores.

Un problema común en el mundo del procesado de señales es la medida de los errores introducidos en un conjunto de datos que representan una imagen o un sonido o un vídeo. Resulta muy difícil cuantificar este tipo de errores y los sistemas clásicos, como la distancia, se demuestran insuficientes.

Para entender más este problema analicemos la figura 3.3. En los apartados (a) y (c) tenemos las figuras originales, mientras que en los apartados (b) y (d) tenemos las mismas figuras pero con diferentes errores introducidos. Si se observan las figuras con detenimiento y se pregunta a cualquier persona cual de ellas se ve peor, es razonablemente seguro suponer que todo el mundo dirá que la (d) es una figura de mucha peor calidad que la (b). Sin embargo, ya podemos adelantar que la figura (b) contiene más errores, es decir, si miramos las diferencias entre los valores de señal originales y los mostrados, estas son más y mayores, pero se ven menos.

Supongamos que utilizamos para analizar estas imágenes dos medidas clásicas de error, como pueden ser la distancia media (o error medio, EM):

$$EM = \frac{1}{N}\sum_{i=0}^{N}(X_i - X_i')$$

o el error cuadrático medio (ECM):

$$ECM = \sqrt{\frac{1}{N}\sum_{i=0}^{N}(X_i - X_i')^2}$$

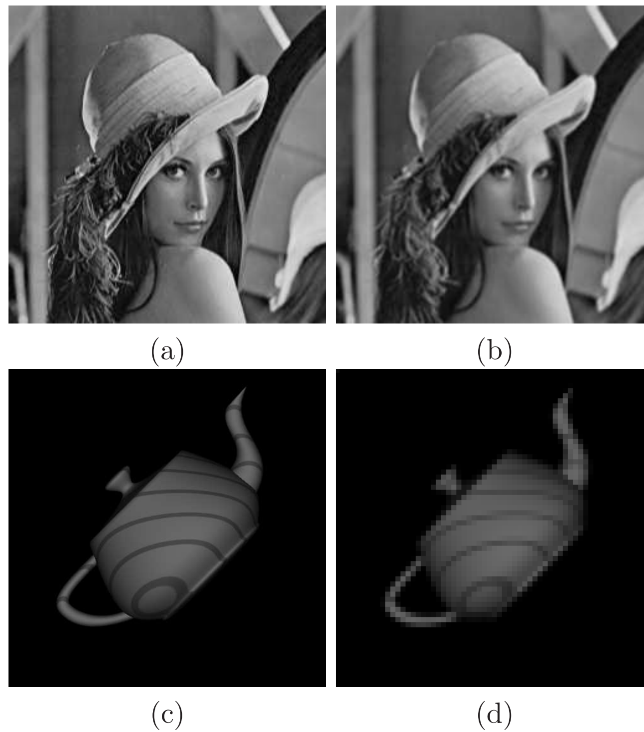


Figura 3.3: Dos imágenes para comparar, originales (a, c) y con errores (b, d).

donde X_i es el valor de un punto de la señal original, X_i' es el valor del mismo punto de la señal con errores y N es el número de puntos totales de la señal.

Si aplicamos ambas medidas a la figura 3.3 vemos que los resultados para la figura (b) son de $EM = 0$ y $ECM = 65$, mientras que para la figura (d) son $EM = 0$ y $ECM = 31$. En ambos casos un resultado cuanto más bajo mejor, así que se puede comprobar que los resultados no reflejan en absoluto la realidad. Está claro que ambas figuras contienen errores, pero el EM no lo refleja ya que los errores se anulan entre si. Sin embargo, el ECM tampoco refleja la realidad ya que indica que hay errores muchos mayores en la figura (b) que en la (d) siendo evidente para nuestra percepción que esto no es así. Esto se debe a una trampa: al analizar dos figuras diferentes, la cantidad de error introducida es distinta, pero influye más en la figura con menor cantidad de "datos visibles" (menor potencia de señal) que es la tetera.

Para compensar este efecto se toma como medida estándar la relación señal a ruido (Signal Noise Ratio, SNR):

$$SNR = 10 \log_{10} \frac{P_S}{P_N}$$

donde

$$PS = \sum_{i=0}^{N} X_i^2$$

y

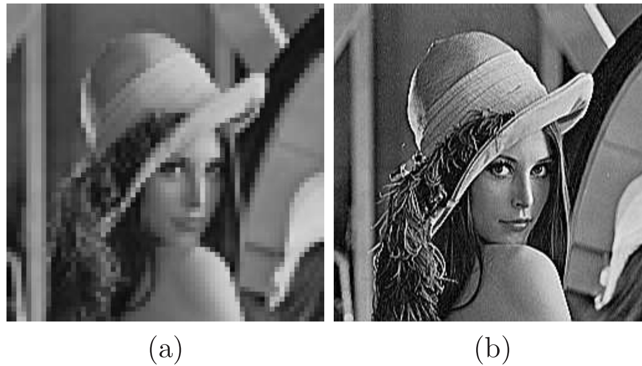
$$PN = \sum_{i=0}^{N} X'^2_i$$

Que como se puede ver introduce en la fórmula la información sobre la potencia de la señal original (PS) y por tanto no se ve afectada por el tamaño de la muestra. Para las figuras de ejemplo, $SNR_{(b)} = 22.8$ y $SNR_{(d)} = 15.8$ (el ideal es infinito y cuanto más bajo peor) lo cual concuerda mejor con nuestra percepción. La salida de esta medida se mide en decibelios (dB). Como regla de aplicación general se suele considerar que una SNR superior a 30 dB tiene una calidad muy buena (de forma que apenas se puede distinguir entre SNRs de 30 y 35 dB), mientras que SNRs entre 25 y 30 dB suelen referirse a calidades buenas pero con diferencias perceptibles y menos de 25 decibelios ya se consideran calidades regulares. Si la señal tiene menos de 15 dB de SNR entonces la calidad se considera pobre o mala y por tanto, por debajo de lo aceptable. En cualquier caso, la SNR es siempre una medida de calidad relativa, es decir, la misma imagen con 25 dB o con 30 dB será de mayor calidad, generalmente, en el segundo caso, pero si una imagen tiene una SNR de 30 dB y un sonido una SNR de 15 dB, es muy difícil saber, relativamente, cual de los dos tiene una calidad mayor (y, en todo caso, la percepción puede varias entre diferentes personas).

Existe todavía otra medida que se usa mucho en procesado de señal es la relación señal a ruido de pico ($PSNR$):

$$PSNR = 20 \log_{10} \left(\frac{\max_{i=0}^{N} (X_i)}{ECM} \right)$$

Pero como se puede observar en la fórmula, esta medida es matemáticamente equivalente al ECM y por tanto introduce los mismos problemas que este, aunque con la ventaja de que los resultados también se miden en dB.



(a) (b)

Figura 3.4: Dos nuevas imágenes de la chica de la pamela, ambas con errores.

Bien, así pues ¿la SNR es la mejor medida posible? Pues desgraciadamente si y no. Es en efecto una de las mejores medidas posibles matemáticamente (y desde luego, la más ampliamente usada), pero tiene límites y, conociendo las características de nuestra percepción, es posible hacer "trampas". Fijémonos en la figura 3.4. En ella tenemos dos nuevas versiones con errores de la imagen (a) de la figura 3.3. Si observamos ambas, parece evidente a nuestra vista que la imagen (a) es de mucho peor calidad que la imagen (b). Sin embargo, si realizamos una medida de la SNR para ambas, obtenemos que $SNR_{(a)} = 16.4$ y $SNR_{(b)} = 15.2$. Sorprendentemente la imagen (b) sale peor parada de la medida. ¿Cómo es esto posible? Pues debido a que los errores introducidos en la imagen (b) se han buscado de forma que realzan los contornos, lo que a nuestra visión le resulta más agradable y mucho más fácil de "arreglar".

¿Como se puede reflejar este funcionamiento de nuestra percepción en una medida de error? Bien, existe un sistema muy ampliamente utilizado que consiste en la simple observación. Es decir, se coge a un grupo de testigos, se les enseñan distintas imágenes y se les pide que califiquen las mejores y las peores. Hemos usado este sistema como un indicador en algunos puntos de esta tesis, pero, lamentablemente, las muestras de testigos no eran demasiado amplias y en algunos casos tampoco se puede afirmar rotundamente que fueran exactamente imparciales, así que estas opiniones deberán tomarse como un mero indicador orientativo.

En la tabla3.1 se puede ver el resultado para las imágenes de las figuras 3.3 y 3.4 de cada una de las medidas de error explicadas. Como se puede observar, la medida que más se aproxima a la calidad subjetiva real, aún no siendo perfecta (último caso) es la SNR.

Imagen	EM	ECM	SNR	PSNR	Calidad subjetiva
Figura 3.3(b)	0	65	22.8	30.0	Regular
Figura 3.3(d)	0	31	15.8	33.2	Mala
Figura 3.4(a)	0	285	16.4	23.6	Mala
Figura 3.4(b)	-1	375	15.2	22.4	Buena

Tabla 3.1: Diferentes medidas de calidad para las figuras 3.3 y 3.4

3.2 El entorno de trabajo.

En este apartado se describen la arquitectura, el entorno y los programas y sistemas de medida utilizados en esta tesis. Asimismo se argumentan las razones que hay detrás de las elecciones realizadas.

3.2.1 Argumentación.

El objetivo de esta tesis es proponer un nuevo sistema de cálculo, el cálculo difuso, y estudiar su viabilidad y posibilidades. Dadas las características de este nuevo sistema y su ámbito de aplicación (mejorar la velocidad y reducir la energía consumida por los sistemas multimedia), el objetivo principal ha sido estudiar el impacto en sistemas de bajo consumo, ya que estos son los que precisan de técnicas más especializadas para mejorar su rendimiento.

Hemos partido de la base que los sistemas de bajo consumo del futuro van a ser, principalmente, sistemas RISC superescalares en orden. Los motivos para usar esta base en lugar de otros sistemas han sido:

- Los sistemas fuera de orden incrementan modestamente el rendimiento si los miramos desde le punto de vista de la potencia: es decir, su rendimiento aumenta pero a costa de incrementar mucho más su consumo. Así pues es difícil que los sistemas de bajo consumo se decidan a incorporar esta solución a sus diseños.

- Los sistemas no superescalares han demostrado no ser capaces de competir sin la ayuda de compiladores manuales y estos salen demasiado caros. Probablemente esta no sea la mejor opción desde el punto de vista teórico, pero desde el punto de vista práctico, un procesador superescalar es, directamente, capaz de ejecutar gran cantidad del código ya existente. La tendencia actual es a añadir extensiones vectoriales o SIMD (tipo MMX) a un sistema superescalar.

Existe una gran pega a esta decisión, que se podría resumir como los procesadores VLIW (Very Long Instruction Word). Efectivamente este tipo de procesadores son una buena elección en muchos sentidos para el procesado multimedia en sistemas de bajo consumo. Sin embargo hemos decidido no usarlos en el análisis ya que se debe entender que, en cualquier caso, la ambición de esta tesis es defender la propuesta del cálculo difuso y mostrar, a través de alguna aplicación concreta, la validez de sus premisas, y no tanto desarrollar todas las facetas de la propuesta para todos los paradigmas de procesado existentes.

Sin embargo, un pequeño análisis de las características de estos procesadores nos revela que estas técnicas, efectivamente son aplicables a VLIW, aunque con sutiles modificaciones que, como ya se ha dicho, no son nuestro principal objetivo. A grandes rasgos:

- Las técnicas de unidades aritméticas difusas se podrían aplicar directamente a las operaciones sobre datos multimedia. Bastaría, al igual que en el caso estudiado, con tener unidades aritméticas configurables mediante un bit que indicase su funcionamiento “normal” o “difuso”.
- Las técnicas de reuso tolerante de instrucciones, dado que se encuentran integradas en el funcionamiento de la unidad aritmética también serían implementables. En este caso, sin embargo, debido a la rigidez de tiempos de los procesadores VLIW, la tabla de reuso debería implementarse en paralelo con las unidad aritmética y la ganancia en tiempo se reduciría.
- En el caso del reuso tolerante de regiones el sistema podría incorporar el nuevo ISA sin problemas. Es más, probablemente el problema de acceder a la tabla a través de varias instrucciones se solucionaría gracias al uso de la línea de instrucciones de los procesadores VLIW. Su implementación sería prácticamente directa al igual que en el primer caso.

3.2.2 Modelo de arquitectura.

Para poder demostrar la validez del paradigma de cálculo propuesto se ha decidido usar como objetivo tres arquitecturas distintas, cada una representativa de un ámbito de mercado concreto, de forma que el resultado cubriese un amplio espectro de posibilidades:

1. En primero lugar se ha utilizado una arquitectura realmente de bajo consumo, escalar en orden y sin predicción de saltos. Esta arquitectura

pretende representar los sistemas de bajo consumo actuales y, quizás, los sistemas de muy bajo consumo del futuro.

2. El segundo modelo que se ha utilizado ha sido una arquitectura superescalar en orden de ancho 2. Esta arquitectura es el futuro de los sistemas de bajo consumo. Actualmente se puede equiparar a procesadores tipo el SH4 de Hitachi que, aun siendo de consumo moderado, son demasiado potentes para ser considerados de bajo consumo y ser instalados en, por ejemplo, un móvil. En un par de generaciones técnicas, probablemente estos chips ya sean los usados para este tipo de dispositivos.

3. Finalmente la técnica se ha evaluado en un procesador superescalar, fuera de orden, de ancho 4. Aunque este tipo de procesadores no son el objetivo principal de la técnica propuesta ya que en ellos el consumo es un problema secundario, se ha querido evaluar si aún con un procesador de alto rendimiento se pueden obtener mejoras significativas en velocidad y ahorro de energía.

El ISA.

Se ha usado como ISA de referencia uno basado en la arquitectura Alpha. Esta arquitectura es RISC y por tanto se adapta bastante bien a nuestro modelo de referencia. Como principal inconveniente de este ISA está el de que es una arquitectura de 64 bits, bastante por encima de lo que es normal en un procesador de bajo consumo, pero los códigos que se han compilado no utilizan esta capacidad de 64 bits y el consumo del sistema ha sido escalado a 32 bits de forma que los resultados sean coherentes con las premisas de trabajo.

Descripción de la microarquitectura.

Nuestros procesadores modelo contienen un pipeline básico de 6 etapas: fetch, decode y rename, issue, execute, write-back y commit, aunque, dependiendo del tipo de instrucción, el número de etapas puede variar. La figura 3.5 muestra el diagrama de bloques de la arquitectura fuera de orden. El diagrama de los modelos de procesador más simples simulados es, básicamente, el mismo pero sin los elementos que no contiene dicho procesador (por ejemplo, el modelo en orden de ancho 1 no tiene memoria cache de nivel 2). En la tabla 3.2 se pueden ver las características exactas de cada procesador evaluado.

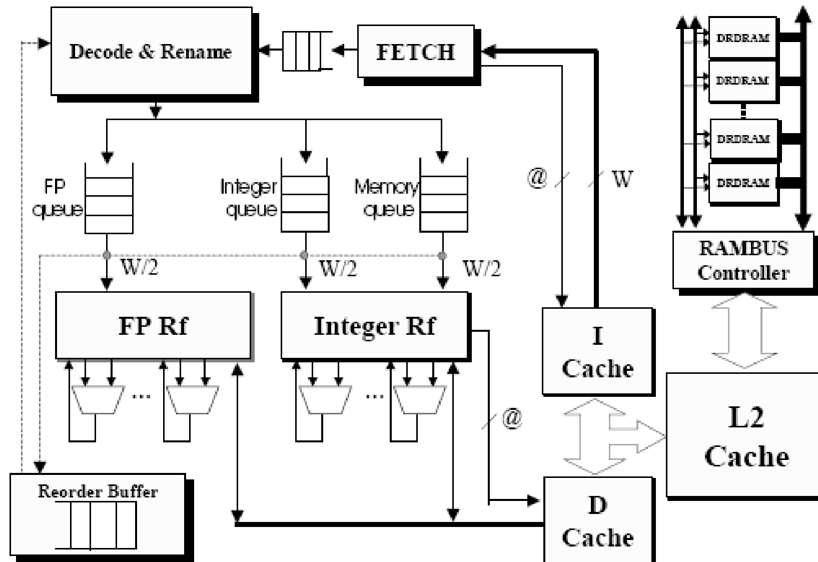


Figura 3.5: Diagrama de bloques del procesador fuera de orden.

El dato que más nos puede llamar la atención de la tabla 3.2 es, sin duda, la capacidad de cálculo en punto flotante del procesador de ancho 1. Para un procesador, teóricamente del menor consumo posible, una unidad de cálculo en como flotante de este tamaño es una utopía y probablemente una pérdida de espacio y consumo. Sin embargo nos hemos decidido por esta aproximación debido a que hemos realizado muchas pruebas con programas que *necesitan* aritmética en coma flotante. Si un procesador no posee un hardware específico para ello debe simular las instrucciones en coma flotante por software y, en este caso, nuestros resultados hubiesen sido mucho mejores, pero seguramente injustos. Este procesador pretende ser nuestra referencia en cuanto a sistemas realmente simples como reproductores de MP3, teléfonos móviles de gama baja y similares.

El procesador de ancho 2 es el que consideramos más interesante. Ha sido escogido basándonos en el procesador SH4 de Hitachi [ANUN98]. Este procesador está dirigido a las aplicaciones multimedia de consumo masivo como los videojuegos (Sega Dreamcast [Ho99]) y las agendas electrónicas. Como se puede ver su núcleo central consiste en un procesador de ancho 2 superescalar con ejecución en orden y tres ramas de ejecución distintas: entera, de saltos y de coma flotante. El SH4 a 200 MHz tiene un consumo de tan solo 1.2 Watios.

Característica	Procesadores		
Ancho del procesador	1	2	4
Ejecución	En orden	En orden	Fuera de orden
Physical Registers	32	32	32
Ancho de Fetch por ciclo	1	2	4
Ancho de Decode por ciclo	1	2	4
Ancho de Issue por ciclo	1	2	4
Ancho de Commit por ciclo	1	2	4
Unidades de Punto Flotante	1	1	4
Unidades enteras	1	2	4
Predictor de saltos	No tomado	2 Niveles	Combinado (bimodal + 2 niveles)
Latencias de las operaciones de punto flotante (latencia de issue)			
Suma (Float)	2 (1)	2 (1)	2 (1)
Suma (Double)	4 (1)	4 (1)	4 (1)
Multiplicación (Float)	4 (1)	4 (1)	4 (1)
Multiplicación (Double)	8 (6)	8 (6)	8 (6)
División (Float)	12 (12)	12 (12)	12 (12)
División (Double)	24 (24)	24 (24)	24 (24)
Jerarquía de memoria			
L1 Dcache Size	16 K	16 K	16 K
L1 Dcache Assoc.	4-way	4-way	4-way
L1 Icache Size	16 K	16 K	16 K
L1 Icache Assoc.	1-way	1-way	1-way
DTLB Size (full assoc)	32	64	128
ITLB Size (full assoc)	32	64	64
L2 Cache	none	64 K 2-way	256 K 4-way
Proceso de fabricación			
Feature Size	.25um	.25um	.25um
Vdd	1.8 V	1.8 V	2.25 V
MHz	200	200	733

Tabla 3.2: Configuración de los procesadores de referencia.

Finalmente nuestro último procesador elegido ha sido un procesador fuera de orden de ancho 4. Aunque el modelo es prácticamente idéntico a un procesador Alpha 21264, su utilidad no ha sido tanto modelar un procesador concreto sino comprobar que nuestras propuesta tienen aplicación también para procesadores de sobremesa y equipos de alto rendimiento.

Un último detalle que llama la atención es que tanto el proceso tecnológico como la frecuencia de trabajo se encuentran por debajo de lo que actualmente se considera tecnología punta. La elección se ha realizado así debido a los problemas de escalado del entorno de simulación. Sin embargo, y dado que los resultados son comparativos entre diferentes ejecuciones del mismo procesador no por ello pierden su utilidad y son extrapolables a especificaciones más agresivas.

3.2.3 El entorno de simulación.

Para realizar las medidas necesarias se han utilizado dos métodos: la ejecución anotada y la simulación. El primer método consiste en ejecutar el

programa con ayuda de un programa de instrumentación. Mediante este programa se obtienen las características del ejecutable que se desean medir (por ejemplo, instrucciones de un determinado tipo ejecutadas y sus valores de entrada). A continuación los datos obtenidos se pueden filtrar mediante un programa al efecto para obtener los datos deseados (en nuestro caso, por ejemplo, la repetición de instrucciones con los mismos datos).

Las ventajas de este método son, sobretodo, la exactitud y la velocidad ya que el programa realmente se ejecuta sobre la máquina para la que se ha compilado. Como desventajas, resulta muy difícil medir el tiempo que gasta cada instrucción o la potencia consumida por una tarea concreta, sobretodo si estamos ejecutando los programas sobre un servidor con muchos usuarios.

El segundo método, la simulación, consiste en el uso de un programa simulador que imita el comportamiento del procesador a estudiar. Este programa lee las instrucciones binarias y simula su resultado haciendolas fluir a través del pipeline de instrucciones programado, anotando los consumos de todas ellas. Es un sistema mucho más lento y costoso, e incluye el riesgo de que el simulador se equivoque, pero nos permite saber exactamente cuanto tiempo pierde un programa debido a un cierto tipo de instrucciones y cuanto consume cada una (al menos en teoría). Y lo que es más importante, nos permite incorporar al simulador nuevas funcionalidades que no podríamos probar en el procesador real ya que deberíamos fabricar uno nuevo.

En esta tesis se ha utilizado ATOM como sistema de instrumentado de los programas compilados y SimpleScalar junto con Wattch como simulador. Es importante comentar que aunque se ha partido de la versión pública de SimpleScalar, esta ha tenido que ser ampliamente modificada para que se adaptase a las necesidades de esta tesis. Además la integración con Wattch ha tenido que ser muy mejorada ya que los cambios de configuración de SimpleScalar no se veían reflejados en las medidas de potencia. Finalmente se han incorporado al simulador resultante los nuevos elementos físicos propuestos en esta tesis de forma que se ha podido obtener una medida global de los efectos de nuestra propuesta. El simulador resultante de este trabajo se ha denominado SimpleReuse.

4

Cálculo difuso de instrucciones

Resumen

En este capítulo se realiza un estudio de la aplicación de la idea del cálculo difuso a la ejecución de instrucciones. Para ello se explican los criterios seguidos para decidir que tipos de instrucciones son susceptibles de seguir este modelo. A continuación se evalúan diferentes métodos para implementarlo como el reuso tolerante o la implementación de nuevas unidades funcionales. Además se presentan los resultados de dichas evaluaciones y las conclusiones alcanzadas, entre las que caben destacar los importantes ahorros en tiempo y energía a los que dan lugar estas nuevas técnicas. Finalmente se presenta un estudio donde se implementan todas estas técnicas de forma conjunta.

4.1 Introducción.

El cálculo difuso de instrucciones consiste dar un resultado aproximado a una instrucción aritmética del procesador. Esta aproximación al cálculo difuso tiene numerosas ventajas:

- Es un sistema sencillo, ya que una instrucción ya es una unidad atómica del programa.
- El error introducido es pequeño ya que está limitado a un solo elemento de cálculo.
- Apenas necesita introducir modificaciones en el flujo de datos del procesador, basta con modificar la ALU o añadir una nueva.

Sin embargo, son también numerosos los inconvenientes a los que se enfrenta:

- No todas las instrucciones del mismo tipo son susceptibles de ser calculadas de forma difusa. Solo aquellas que afectan a los datos del programa. Realizar de forma difusa instrucciones de control puede llevar a resultados muy indeseados.
- Una sola instrucción, en general, consume pocos recursos del sistema para realizarse, de forma que las ganancias esperadas por instrucción son pequeñas.
- Es difícil implementar mecanismos de control de errores en el sistema que consuman menos que la ganancia que obtenemos con el cálculo difuso (que, como ya se ha dicho, se espera que sea pequeña por instrucción).

4.1.1 Trabajo Relacionado.

Como se puede ver a raíz de lo mostrado en la lista de ventajas e inconvenientes, los problemas a los que se enfrenta en la práctica el cálculo difuso de instrucciones son principalmente dos: determinar que instrucciones calcular de forma difusa y conseguir ganar algo de potencia y tiempo de ejecución en estas.

De todas las posibles implementaciones del cálculo difuso en este capítulo estudiaremos dos de ellas: el uso de unidades funcionales modificadas (de

forma que proporcionen una menor precisión a cambio de un menor consumo) y lo que hemos llamado reuso tolerante.

La primera técnica es totalmente nueva, quizás no en la implementación (el uso de unidades funcionales mas "pequeñas" ya se ha dado anteriormente) sino en la forma de hacerlo, ya que no nos basamos en el espacio desaprovechado en los registros físicos, como se hace en otras técnicas [BM99, GCO+04, GCP+05], sino que reduciremos la precisión.

La segunda técnica el reuso de instrucciones si que ha tenido mucho trabajo previo. En este sentido, nuestra propuesta no se basa en una técnica nueva sino que la aplica de una forma totalmente innovadora para conseguir resultados mucho mejores a los habituales.

Las técnicas de reuso de instrucciones han sido ya ampliamente estudiadas en la literatura en muchos contextos diferentes. Técnicas de memorización se usaron desde hace mucho tiempo para evitar calcular dos veces funciones con los mismos parámetros y muchos compiladores las usan para agilizar la optimización de programas ya que las comprobaciones de las mismas dependencias se llevan a cabo numerosas veces.

En [Har80] y [Har82] Harbison propone una arquitectura orientada a pila, la "Tree Machine", que usa un mecanismo hardware, la caché de valores, para eliminar subexpresiones comunes e invariantes de los bucles.

En [SS97] se introdujo el reuso de instrucciones y surgió a partir de la observación de que muchas instrucciones pueden evitarse si ya han sido ejecutadas con los mismos valores de entrada. El artículo muestra que una fracción significativa del número total de instrucciones dinámicas (hasta el 50% en algunos casos) puede ser reusada mediante, tipicamente, tablas de reuso que mantienen una copia de las entradas y salidas de distintas instrucciones.

Desafortunadamente, tal y como se muestra en [SBS00], el reuso de instrucciones solo es rentable cuando se reusan simultaneamente diversas instrucciones ya que de otra forma solo se consigue reducir la latencia de las instrucciones reusadas (suponiendo que el tiempo de acceso a la tabla sea menor que el tiempo de cálculo de la instrucción). Así pues, si queremos reusar instrucciones individuales deberemos restringirnos a aquellas que sean largas de computar.

El el contexto del reuso de instrucciones, Ranganathan et al. [RAJ00] evaluaron el impacto de aplicar caches reconfigurables para el reuso de instrucciones a diferentes programas multimedia. Las mejoras de rendimiento fueron del 4% al 20%. Más trabajo relacionado con el tema se puede en-

contrar en [CFR98] y en [AFL97] donde se muestra que la memorización de instrucciones pueden conseguir reducir el tiempo de computación y la energía consumida en operaciones de larga latencia como las multiplicaciones y las divisiones.

Otro factor a tener en cuenta en el reuso de instrucciones es el factor engañoso que introducen las instrucciones denominadas triviales. Dichas instrucciones fueron introducidas en primer lugar por Richardson en [Ric93] y consisten en instrucciones cuyo resultado es inmediato (por ejemplo, una multiplicación cuando uno de los operandos es 0) y que por tanto pueden evitarse (es decir, no es necesario ni computarlas ni memorizarlas). Si un sistema de memorización no incluye un detector de operaciones triviales tiene muchas más posibilidades de funcionar correctamente ya que acertará muchas veces en estas.

En [SBS00] se demuestra que para poder aplicar las técnicas de reuso de instrucciones es necesario buscar aquellas que tengan una alta complejidad de cálculo y, preferentemente, que presenten un alto grado de consumo.

4.1.2 Ámbito de trabajo

En esta tesis el objeto principal de estudio han sido las instrucciones de punto flotante. Estas instrucciones resultan particularmente adecuadas para el cálculo difuso dado que:

- Toda instrucción de punto flotante introduce implícitamente un error en su cálculo. Es muy fácil, pues, encontrar sistemas que, simplemente, aumenten este error.

- Las instrucciones de punto flotante son caras de realizar en el sentido computacional: consumen mucho tiempo y energía del procesador.

- Las instrucciones de punto flotante no están asociadas al control de los programas, sino a los datos. Esto permite a un sistema ciego (y por tanto automático) calcular “mal”, solamente las instrucciones referidas a los datos.

El gráfico de la figura 4.1 muestra las diferentes formas en las que se ha intentado aplicar el cálculo difuso de instrucciones a las instrucciones de punto flotante. Como se puede ver hay dos aproximaciones principales:

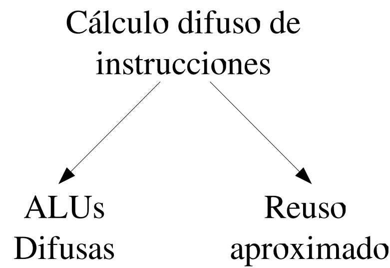


Figura 4.1: Esquema del reuso difuso de instrucciones.

- Por un lado se ha intentado ver la viabilidad de utilizar unidades funcionales más cortas para realizar los cálculos y obtener así resultados de menor precisión, pero con un gasto menor.
- Por otro lado, se han introducido tablas de reuso que permiten reutilizar resultados de instrucciones previas con entradas de datos parecidas. Al reusar instrucciones con entradas similares se consigue incrementar el número de instancias de instrucciones reusadas pero a cambio de que el resultado sea aproximado.

4.2 El estándar IEEE754.

Las codificaciones en coma flotante son las más usadas en el mundo de la computación para la representación de números reales. Esto es así debido a que sus ventajas sobre la coma fija son numerosas, siendo quizás la principal, el gran rango de los números representables. Sin embargo, durante muchos años la representación en coma flotante era uno de los principales problemas de incompatibilidad de programas debido a que cada fabricante de hardware implementaba su propia versión.

En 1987 el IEEE especificó un estándar (el 754 de 1985 según la numeración actual [75485][1]) que sería la base de prácticamente toda la computación en coma flotante actual. En este apartado veremos las características más relevantes de este estándar y su influencia en las propuestas de esta tesis.

Para empezar hay que dejar claro que el estándar define como necesario un tipo de datos, denominado de simple precisión, de 32 bits y propone, además 3 tipos de datos opcionales adicionales: de simple precisión extendida (de al menos 43 bits), de doble precisión (de 64 bits) y de doble precisión

[1] Contra las apariencias, no hay error en las fechas, la norma es de 1985 pero se completó en 1987.

Parámetro	Formato			
	Simple	Simple Ext.	Doble	Doble Ext.
m	24	$\geq$32	53	$\geq$64
e_{max}	+127	$\geq$1023	+1023	>16383
e_{min}	-126	$\leq$-1022	-1022	$\leq$-16382
Tamaño del exponente	8	$\leq$11	11	$\geq$15
Tamaño del formato	32	$\geq$43	64	$\geq$79

Tabla 4.1: Formatos definidos en el IEEE 754.

extendida (de al menos 79 bits). En la práctica la mayoría de procesadores implementan las operaciones de precisión simple y doble y nada más (de hecho numerosos procesadores emulan la doble precisión mediante el hardware de simple precisión). La gran mayoría de programadores, además, desconocen la existencia de las precisiones extendidas y se limitan a utilizar las dos más conocidas (aun cuando el hardware pueda usar la precisión extendida para hacer más fiable la no-extendida[2]).

Todos los formatos definidos en el estándar IEEE 754 almacenan números de la forma: $\pm m \times 2^e$, empleando un bit para el signo. Los exponentes se codifican en exceso y las mantisas en binario natural con bit oculto, primer bit significativo y coma a la derecha. En la tabla 4.1 se puede ver el tamaño de cada parte para cada uno de los formatos.

Actualmente el estándar IEEE 754 y su ampliación (la norma 854) están en proceso de revisión. Las líneas principales de esta revisión son[75406]:

- Añadir un nuevo formato de 128 bits que muchos procesadores ya ofrecen de facto, pero mayoritariamente a través de excepciones (software).
- Añadir la estandarización de la aritmética decimal.
- Clarificar numerosos aspectos del tratamiento de excepciones.

En cualquier caso la revisión no afectará a los resultados expuestos por esta tesis excepto por el hecho de que el nuevo formato de 128 bits permitiría muchas más ganancias si fuese implementado siguiendo nuestras propuestas.

[2]Para más detalles acerca de este problema ver [Gol91]

Programa	Descripción	Datos	Características
Epic	Compresión basada en Wavelets	pamela.pgm	Mapa bits mostrando una chica
Texgen	MESA 3D: API de gráficos 3D	teapot.ppm	Textura de una tetera de Utah
SpeechRec	Reconocimiento del habla	numbers.wav	Secuencia de mil números del 0 al 9
Lame	Codificador MP3	fugue.wav	Banda sonora de Star Wars

Tabla 4.2: Programas de prueba utilizados.

4.3 Los programas de pruebas.

En este apartado de la tesis se han usado como programas de pruebas una combinación de programas del mediabench [LPMS97] y de programas específicos de multimedia. Nuestra metodología para seleccionar los programas adecuados ha sido la de buscar programas representativos de los grandes grupos de procesado multimedia existente: imagen estática, vídeo, codificación de audio y reconocimiento de voz. Dentro de estos grupos se han buscado, por un lado, programas lo más representativos posible y, por otro, programas que contuvieran el tipo de características que nos interesaba analizar (es decir, operaciones en coma flotante) en su codificación original. Es importante resaltar que ninguno de los programas ha sido modificado en forma alguna para realizar las pruebas, simplemente se han compilado con un compilador estándar con las opciones habituales (incluidas las de optimización).

La tabla 4.2 muestra los benchmarks utilizados en estas pruebas. Epic es un sistema de compresión de imágenes estáticas de nueva generación [AS90]. Está basado en Wavelets y alcanza unos niveles de compresión muy superiores a JPEG (a cambio, precisa utilizar aritmética en coma flotante). Texgen es un programa de prueba que forma parte de la librería MESA[Pau97]. Genera una tetera con textura. SpeechRec es un programa reconocedor del habla. Es un programa experimental que marcará la línea de posibles desarrollos futuros que nos proporcionó el departamento de Señal y Comunicaciones de la UPC[Fon06]. Finalmente, Lame es uno de los numerosos programas de codificación de sonido en formato MP3[Pro06].

4.4 Unidades funcionales difusas.

4.4.1 Metodología.

La creación de unidades funcionales difusas es especialmente sencilla utilizando el sistema de punto flotante definido por el estándar IEEE 754. En

dicho estándar se especifica que los valores de punto flotante se almacenan divididos en tres partes: signo, exponente y mantisa. Así pues, una operación de multiplicación requiere, por ejemplo, multiplicar los signos, sumar los exponentes y multiplicar las mantisas (y, además un posible último paso de alineación y recodificación para mantener el formato de coma a la derecha del primer bit, que es significativo y oculto). La figura 4.2 muestra el proceso esquematizado.

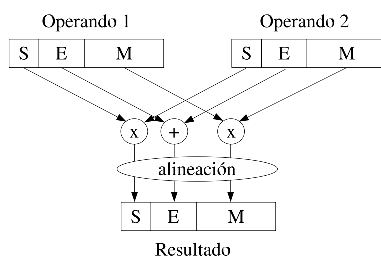


Figura 4.2: Multiplicación en formato IEEE 754.

Como se puede ver a partir de la figura 4.2 en el proceso solo está involucrada una operación real de multiplicación (ya que multiplicar los signos es inmediato) que es la de multiplicar las mantisas. Esta multiplicación es la operación más costosa del proceso ya que requiere un multiplicador capaz de admitir dos números binarios de N bits y obtener un resultado de, al menos, $N + 1$ bits (para redondeo, el resto será descartado). ¿Que pasaría si descartáramos algún bit extra en la multiplicación? El resultado seguiría siendo válido pero menos preciso. A cambio, el multiplicador utilizaría menos energía en el proceso de cálculo (ya que es más pequeño).

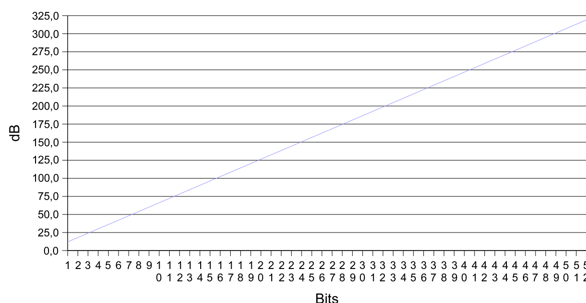


Figura 4.3: SNR mínima según los bits de mantisa de un número real.

Evidentemente, a cambio del ahorro de energía obtenido, también incrementaríamos el error. En la figura 4.3 se puede ver la SNR mínima que obtenemos según el número de bits correctos que contenga la mantisa de un número real. Como se puede ver es una relación totalmente lineal que se puede calcular a partir de la siguiente fórmula:

$$SNR = 10 \log \left(\frac{P_s}{P_n} \right)$$

Donde P_s es la potencia del valor correcto y P_n la potencia del error introducido, es decir,

$$P_s = \sum x^2$$

y

$$P_n = \sum (x - x')^2$$

Si suponemos que x es un valor real de doble precisión y x' es el mismo valor, pero con un error en el último bit de la mantisa, podemos desarrollar las fórmulas anteriores como:

$$x = \pm m \times 2^e$$

y

$$x' = \pm m' \times 2^e$$

Siendo m y m' los valores de sus respectivas mantisas, e su exponente y su signo, cualquiera[3]. Por tanto,

$$m = 1 \times 2^0 + m_0 \times 2^{-1} + m_1 \times 2^{-2} + ... + m_{52} \times 2^{-53}$$

y, si solo tenemos un error en el último bit,

$$m' = 1 \times 2^0 + m_0 \times 2^{-1} + m_1 \times 2^{-2} + ... + m'_{52} \times 2^{-53}$$

es decir,

$$P_n = (x - x')^2 = (m \times 2^e - m' \times 2^e)^2 = ((m - m') \times 2^e)^2 = (m - m')^2 \times 2^{2e} =$$
$$= (m_5 2 \times 2^{-53} - m'_{52} \times 2^{-53})^2 \times 2^{2e} = 2^{-106} \times 2^{2e} = 2^{2 \times (-53 + e)}$$

Si ahora aproximamos la P_s por su valor mínimo:

[3] Lo importante es que tanto el signo como el exponente, sean los que sean, son iguales.

$$P_s \geq 1 \times 2^e$$

obtenemos:

$$SNR \geq 10 \log \left(\frac{2^{2e}}{2^{2\times(-53+e)}} \right) = 10 \log \left(\frac{1}{2^{-106}} \right) =$$

$$= 1060 \log 2 \approx 319.1\ dB$$

Es decir, un error en el último bit de la mantisa de doble precisión, deriva en, como mínimo, unos 320 dB de relación señal a ruido, más que suficiente para nuestros sentidos. Evidentemente, a medida que quitamos bits, la relación señal a ruido baja (tal como muestra la figura4.3) y, además, el efecto acumulativo de las operaciones puede dar lugar a efectos muy interesantes y altamente indeseables [Gol91]. Aun con todo esto en mente, a la vista de los resultados anteriores se pueden extraer dos importantes conclusiones:

- El error introducido con la multiplicación (o la división) al disminuir la precisión de las mantisas en una cantidad determinada de bits, es independiente de los datos de entrada cuando operamos en punto flotante.

- Si tenemos en cuenta el error detectable habitualmente por nuestros sentidos (unos 30 dB) la precisión de los formatos de doble precisión es más que suficiente para realizar las operaciones de multimedia.

Finalmente, es importante hacer notar que, si partiendo del estándar de doble precisión del IEEE754, eliminamos 29 bits de precisión de la mantisa, no estamos utilizando el estándar de simple precisión, ya que el rango dinámico continuaría siendo el de doble precisión y, por tanto, podríamos representar muchos más números que en simple precisión.

Así pues se procedió a modificar el simulador SimpleScalar para introducir la posibilidad de "acortar" las unidades funcionales. Asimismo, el simulador Wattch fue modificado para tener en cuenta dichos "recortes" en sus cálculos de potencia.

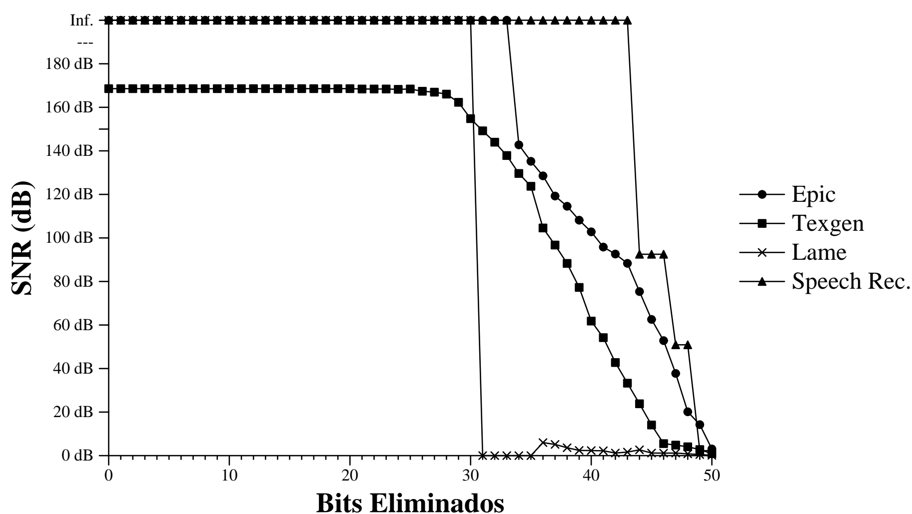


Figura 4.4: SNR al introducir unidades funcionales difusas.

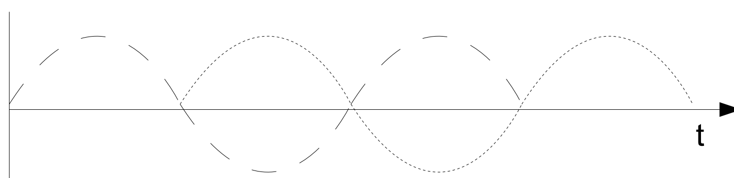


Figura 4.5: Dos sonidos idénticos desfasados: SNR=0 dB.

4.4.2 Resultados.

En primer lugar se ha evaluado hasta que punto se pueden eliminar bits de la operación de multiplicación y división sin afectar a los datos de salida. Para ello se ha medido la SNR resultante entre la salida original de los programas de pruebas y la salida obtenida mediante nuestro sistema con unidades difusas en función de los bits eliminados de las unidades funcionales. Dichos resultados se pueden ver en la gráfica4.4. Hay algunas SNR que no se han podido representar debido a que los programas dejaban de funcionar correctamente antes de dicho punto.

Tal y como se puede apreciar en la figura 4.4, los resultados varían mucho de unos programas de pruebas a otros. Si partimos de que deseamos obtener una SNR mínima de unos 30 dB (lo cual se considera una calidad excelente), podemos eliminar desde 48 bits en la aplicación de reconocimiento de voz, o 47 en el compresor de imágenes Epic, hasta tan solo 30 en el codificador de mp3 Lame. Este resultado, sin embargo, es engañoso ya que la baja SNR obtenida en el codificador mp3 se debe a un pequeño desfase de las señales de sonido que resulta imperceptible para el oído humano. No olvidemos que

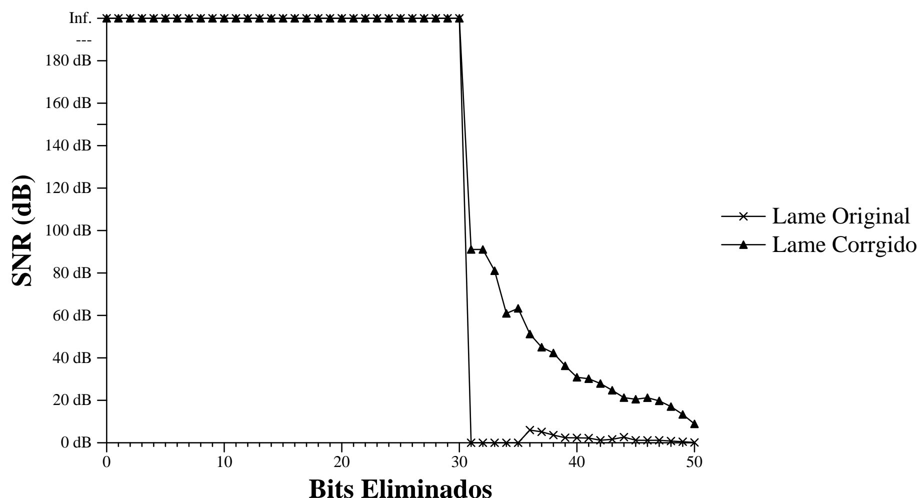


Figura 4.6: SNR del codificador Lame con unidades funcionales difusas, corrigiendo los desfases.

los sonidos se perciben como señales de una determinada frecuencia y que es posible tener dos señales de exactamente la misma frecuencia (y que por tanto suenen exactamente igual) desfasadas 180º y que, por tanto, su SNR sea de 0 dB (ver figura 4.5). Como solución a este problema hemos procedido a medir la SNR mediante un software específico para señales de sonido que corrige el desfase introducido. Si aplicamos esta corrección obtenemos la figura de SNR que se puede ver en la figura 4.6. A esta misma conclusión podemos llegar aplicando el criterio subjetivo (es decir, simplemente escuchando las señales) pero se ha preferido incluir esta nueva medida específica a efectos teóricos.

Así pues, usando los resultados de las figuras 4.4 y 4.6, podemos ver que se pueden llegar a eliminar hasta 42 bits de precisión de las unidades de coma flotante de doble precisión y obtener, aún así, resultados de una calidad buena para los sentidos humanos. Una precisión que es necesario hacer en este punto es que los programas de pruebas no funcionan (es decir, el programa da una salida totalmente incorrecta, no da ninguna salida en absoluto o produce un error) si en lugar de utilizar en los cálculos variables de doble precisión usamos variables de simple precisión.

Este hecho es importante ya que implica que educar a los programadores para substituir el tipo "double" por el tipo "float" en los programas no es una medida de ahorro, ya que la doble precisión es necesaria, no tanto por la precisión, como por el rango dinámico de los valores que puede almacenar. Tomemos como ejemplo el reconocimiento de voz. Para realizarlo se utilizan cálculos que implican una cadena de multiplicaciones en punto

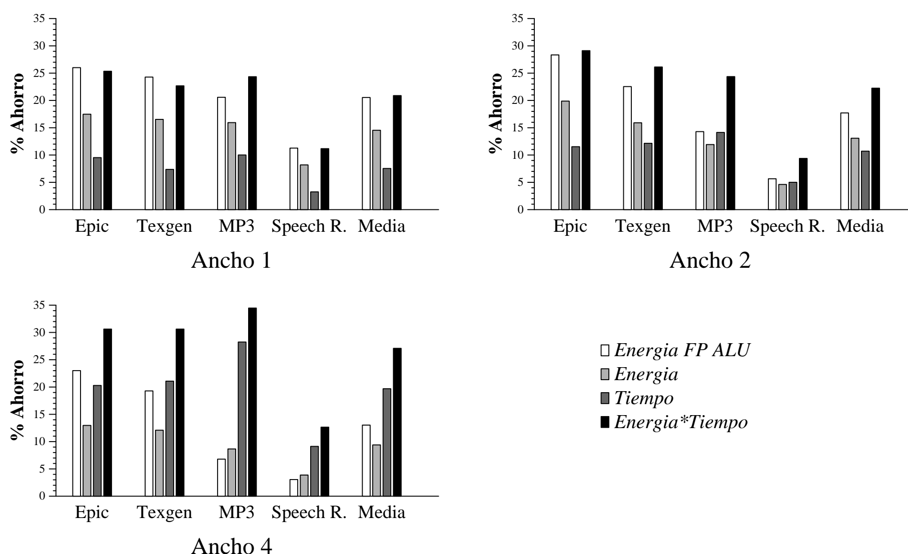


Figura 4.7: Resultados de utilizar unidades funcionales difusas en diferentes procesadores.

flotante (ya que el reconocimiento se realiza a través de cadenas de Markov). Si suponemos que estos valores se encuentran entre 0 y 1 y que un valor típico, por ejemplo, es 0'1 obtendremos que al cabo de 10 multiplicaciones (si el valor inicial es 1) la salida es de 0'0000000001, es decir, tiene 1 bit de precisión (igual que el valor original) pero necesita ya al menos 3 o 4 bits de exponente. Si la cadena en lugar de 10 multiplicaciones tiene 200 el resultado seguirá teniendo un solo bit de precisión, pero ya no será representable en simple precisión (obtendríamos un underflow aún con el uso de números denormales).

Este efecto, efectivamente, se puede compensar con un adecuado y cuidadoso examen del algoritmo y un código hecho a medida (recordemos las técnicas que se usaban para la implementación en coma fija), pero la realidad es que nadie utiliza ya este recurso debido a su alto coste de desarrollo. Se usa el tipo de datos "double" que garantiza que no tendremos que lidiar con estos problemas.

Una primera propuesta realizada a raíz de este trabajo es, pues, reducir la precisión de las unidades de doble precisión (para procesado de algoritmos multimedia) de doble a simple, es decir, en 29 bits. Esta primera vertiente implica que las operaciones de multiplicación de doble precisión se realicen en parte mediante la ALU de simple precisión (multiplicación de mantisas) y en parte mediante la de doble (suma de exponentes). Evidentemente, dado que la mayoría de procesadores ya disponen tan solo de una única ALU, esto simplemente consistiría en ajustar convenientemente el flujo de datos. Los

resultados para los distintos tipos de procesadores simulados pueden verse en las gráficas de la figura 4.7. Para cada procesador y programa de pruebas se muestran, respectivamente, los ahorros de: energía en la unidad de punto flotante, energía en el conjunto de la aplicación, tiempo de ejecución en el conjunto de la aplicación y producto energía*tiempo. Como puede verse los ahorros alcanzados son significativos, superando para todos los casos el 20% de media en el producto energía*tiempo.

Entre los resultados más significativos de los que se muestran en las gráficas de la figura 4.7 se encuentra la gran diferencia cualitativa entre la incidencia de este tipo de medida en los procesadores de bajo consumo y los de altas prestaciones. Así, mientras que en el modelo de procesador de ancho 1 conseguimos un gran ahorro de energía (de hasta casi el 15% de media), en el modelo de ancho 4, fuera de orden, el ahorro baja hasta aproximadamente un 9%. Sin embargo, las reducciones en tiempo de ejecución presentan un comportamiento inverso, siendo muy altas en el procesador más agresivo (cerca del 20% de media) y mucho más moderadas (aunque igualmente significativas) en el procesador más sencillo (de poco más de un 7%). Este efecto se debe a que el procesador fuera de orden es incapaz de esconder la latencia de la unidad de doble precisión (no olvidemos que estas operaciones, en estos códigos, se encuentran en el nucleo de las aplicaciones, típicamente dentro de bucles cortos donde es difícil encontrar muchas operaciones alternativas en paralelo). Así, una reducción del tiempo de ejecución en estas operaciones que son el cuello de botella, debido a la ley de Amdahl, es mucho más significativa en un procesador rápido que en un más lento. En la energía nos encontramos con el efecto contrario, ya que esta unidad de punto flotante es, proporcionalmente, más significativa en un procesador sencillo que en uno complejo, y, así pues, los ahorros de energía son mayores en los dos modelos de procesador más sencillos.

Como se puede ver en la última columna de las gráficas de la figura 4.7, estos dos efectos se complementan para obtener unos ahorros sorprendentemente significativos en todos los procesadores cuando nos referimos al producto energía*tiempo. Aquí las ganancias obtenidas se mueven, de media, entre el 20 y el 30%.

Una segunda propuesta alternativa que surge a partir de estos resultados es la de diseñar un nuevo tipo de datos. Este tipo de datos que podríamos calificar de "simple precisión, doble rango" (SPDR), correspondería a datos de 32 bits (para poder almacenarlo en registros de simple precisión) con 11 bits de exponente, 1 de signo y, consecuentemente, 20 de mantisa. Este tipo de datos presenta prácticamente las mismas ganancias que las expuestas anteriormente, pero la filosofía de trabajo es totalmente distinta. Mientras que la propuesta anterior implica utilizar parte de la unidad de simple

precisión para realizar los cálculos de doble precisión y, por tanto, es una herramienta hardware que debería activar el compilador o el procesador bajo ciertas circunstancias, esta segunda propuesta implica hacer conscientes a los programadores de que disponen de un nuevo tipo de datos que les proporcionará una gran libertad de rangos de representación y una precisión razonable a cambio de mayores velocidades de proceso. Nuestra primera propuesta no implica ningún cambio en los programas, mientras que la segunda si, se ha de definir un nuevo tipo de datos y se ha de implementar como estándar en todos los procesadores (y es, por tanto, mucho más difícil de llevar a cabo). Sin embargo, en caso de llevarse a la práctica, estamos convencidos de que sería muy útil, ya que el sistema dejaría de ser ciego y los datos SPDR podrían calcularse con un gasto de energía menor y más rápidamente en todos los programas que los utilizasen (y no solo en aquellos que podemos calificar de multimedia).

4.5 Reuso aproximado.

La siguiente aproximación que se plantea al cálculo difuso de instrucciones se basa en el reuso aproximado de instrucciones. Antes de evaluar la posibilidad de realizar reuso aproximado, sin embargo, se decidió evaluar la posibilidad de realizar reuso clásico para las instrucciones de punto flotante.

4.5.1 Sistemas de reuso clásico aplicados a las operaciones de punto flotante en multimedia.

Esquema de memorización clásica para sistemas de bajo consumo.

Lo primero que estudiamos fue la viabilidad de los sistemas de memorización clásica para multimedia aplicados a las operaciones de punto flotante. Para ello utilizaremos un esquema con tabla de reuso (TDR) clásico como el que se puede ver en la figura 4.8.

Los sistemas de reuso que se pueden encontrar habitualmente consisten en una tabla de reuso a la cual se accede mediante algún tipo de indexación a partir de la operación memorizada y los operandos de entrada. Dicha tabla almacena, para cada tipo de operación memorizada, los operandos de entrada y el resultado correspondiente. Así pues, si los operandos con los que accedemos están en la tabla (tenemos un acierto), la instrucción no necesita ser procesada y podemos obtener directamente el resultado. En cambio, si los operandos no se encuentran memorizados (fallamos) deberemos realizar

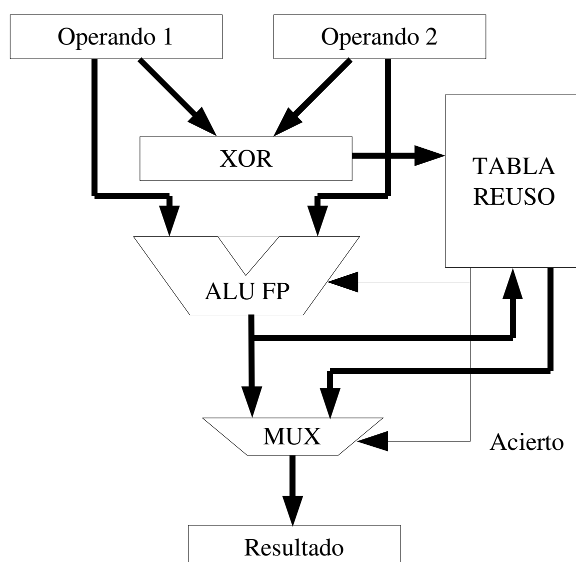


Figura 4.8: Esquema de una tabla de reuso secuencial.

la operación normalmente y, al finalizar, proceder a actualizar la tabla con los nuevos valores.

El esquema de la figura 4.8 muestra el esquema para un acceso secuencial a la tabla, donde primero comprobamos si acertamos o fallamos el acceso y a continuación, en caso de fallo, realizamos la operación. Existe otra posibilidad, que es acceder a la tabla en paralelo, es decir, a la vez que verificamos si tenemos los operandos memorizados, vamos empezando a realizar la operación. Ambos esquemas presentan ventajas e inconvenientes. En el sistema secuencial, un acierto en la tabla implica no gastar nada de energía en realizar la operación (solo gastaremos el consumo de acceder a la tabla), pero un fallo dará como resultado al menos un ciclo más de latencia (que habremos perdido mirando si ya habíamos memorizado la entrada). En el esquema en paralelo, en cambio, un fallo no implica más tiempo de proceso, pero un acierto gasta siempre un poco más de energía ya que la operación ya había empezado. Conceptualmente se puede considerar que, si nuestra mayor preocupación es la energía, el esquema secuencial es el más adecuado. Si, por el contrario, lo que deseamos es velocidad, es mejor utilizar el esquema paralelo.

Instrucciones triviales.

En el sistema de memorización clásica estudiado hemos incluido, además, un sistema de detección de instrucciones triviales. La idea de dicho esquema es

Operación	Condición de los operandos	Resultado
Suma / Resta	Uno de ellos igual a 0	El otro operando
Multiplicación	Uno de ellos igual a +/-0	+/-Cero
Multiplicación	Uno de ellos igual a +1/-1	+/- El otro operando
División	El divisor igual a +/-1	+/- El dividendo

Tabla 4.3: Instrucciones Triviales.

Operaciones:	Suma, Resta, *Multiplicación y División*
Cantidad de tablas:	*1*, 2 y 4
Tamaños:	1,5 KB; *6 KB; 24 KB* y 96 KB
Asociatividad:	1, *2* y 4
Indexado:	*XOR bms*[4]; Bms superpuestos; etc.

Tabla 4.4: Variables evaluadas para las tablas de reuso.

detectar previamente aquellas instrucciones cuyos operandos son tales que el resultado de la operación es inmediato y no implica realizar ningún cálculo. En la tabla 4.3 se puede ver un resumen de las condiciones que hacen una instrucción trivial.

Así pues, las instrucciones detectadas como triviales ni se computan ni se almacenan en la tabla de reuso, sino que directamente dan lugar a la salida correcta. De esta forma el ahorro de energía del sistema clásico es mayor, ya que estas instrucciones, además de acertarse siempre que se dan las condiciones, no ocupan espacio en la tabla y consumen un mínimo (tan solo lo necesario para detectarlas).

Viabilidad de la memorización clásica.

La memorización clásica ha sido evaluada para muchos parámetros diferentes. La tabla 4.4 muestra todas las posibilidades medidas y la tabla 4.5 los parámetros que se han encontrado como idóneos.

Características de las tablas	
Numero de tablas hardware	1
Operaciones memorizadas	Multiplicación y División
Indexado	XOR de los bits menos significativos de las mantisas
Tamaños	6 KBytes (Bajo Coste) y 24 KBytes (Agresiva)
Asociatividad	2

Tabla 4.5: Características óptimas de las tablas de reuso.

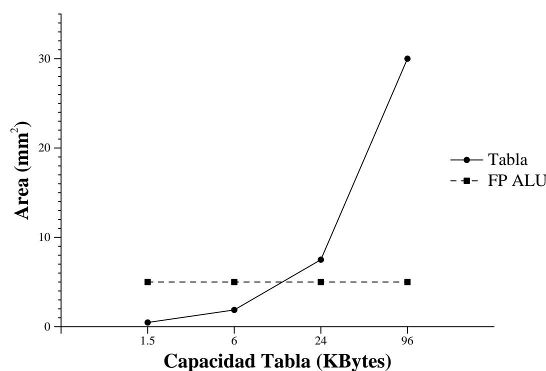


Figura 4.9: Tamaño de la tabla de reuso frente a la ALU de FP.

Se ha decidido memorizar tan solo las operaciones de multiplicación y división debido a su mayor latencia y consumo de energía. Las operaciones de suma y resta pueden no ser rentables en este sentido[5]. Se decidió asimismo utilizar una única tabla de reuso con un bit extra que permitiera almacenar el tipo de operación. Esto se debe a que la cantidad de divisiones que típicamente se pueden encontrar en un programa es baja, pero aún así son lo bastante costosas como para merecer la pena almacenarlas. Se podría haber optado también por dos tablas de distinto tamaño, pero el incremento de complejidad no justificaba las pocas ganancias obtenidas frente a usar una única tabla. Sobre este problema, hay que tener en cuenta que si el número de divisiones de un programa es inusualmente bajo, esto es debido a que una de las optimizaciones estándar en los compiladores es substituir la división por la multiplicación por el recíproco, siempre que sea posible, ya que esta segunda operación es mucho más rápida.

El sistema de indexado de las tablas también fue objeto de diversas medidas. Entre las opciones estaba realizar una XOR de los bits menos significativos de los operandos, concatenar los bits de un operando con los de otro, intercalarlos, o realizar las mismas operaciones con los bits más significativos en lugar de los menos. Un análisis detallado de estos casos para instrucciones enteras (con resultados idénticos a los obtenidos por nosotros) puede encontrarse en [CF00b].

Finalmente el análisis de los tamaños de las tablas nos llevó a descartar los casos extremos. En las figuras 4.9 y 4.10 puede verse una comparativa entre el tamaño (en mm^2) y el consumo (en vatios) de la unidad de punto flotante y el de las tablas de reuso. A partir de las figuras se puede ver que entre las dos tablas más pequeñas no hay diferencias significativas, mientras que la tabla mayor ocupa más espacio que la propia unidad de punto flotante

[5]Este punto también lo discutieron Citron et al. en [CFR98, CF00a, CF00b].

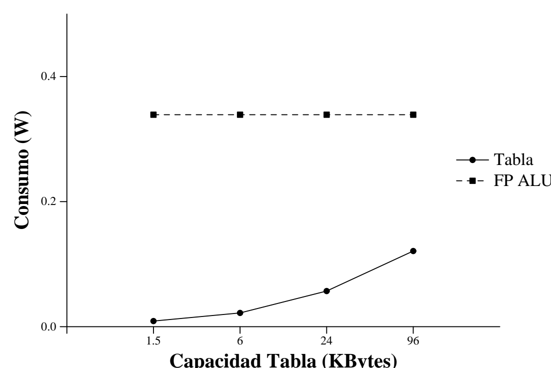


Figura 4.10: Consumo de la tabla de reuso frente a la ALU de FP.

y, además, consume casi la mitad de potencia que esta. Como todo cálculo implicará al menos un acceso a la tabla, esta última tabla es demasiado grande y consume demasiado. Así pues se ha decidido utilizar para el estudio las dos tablas intermedias. La de 6 KB es de un tamaño muy razonable aún para un procesador pequeño, y la más agresiva de 24 KB puede dar mejores resultados a costa de ocupar más espacio. Como cada entrada de la tabla debe almacenar 3 números en punto flotante de doble precisión, ocupa 24 octetos, de forma que la tabla de 6KB tiene 256 entradas y la de 24 KB, 1024 entradas.

Resultados de la memorización clásica

En la figura 4.11 se pueden ver los resultados del sistema de memorización clásica para las operaciones de punto flotante con los distintos tamaños de tabla analizados. Las diferentes gráficas de la figura muestran, respectivamente, los ahorros en energía, tiempo y energía*tiempo, para todo el programa analizado, de los diferentes métodos frente al programa ejecutado en un procesador sin tablas de reuso. Los resultados se muestran para el procesador de ancho 2 descrito en el capítulo anterior.

En las gráficas se puede observar, en primer lugar, que las ganancias son pequeñas. Dos de los programas apenas presentan ahorros de energía y prácticamente ninguno presenta ganancias en tiempo de ejecución. Tan solo uno de los programas (Texgen) presenta ahorros significativos en la gráfica energía*tiempo.

Además resulta significativo observar que el mecanismo detector de operaciones triviales presenta prácticamente unos resultados igual de buenos que la tabla de 24 KB, siendo mucho más económico en espacio y complejidad. Así pues, aunque una tabla de reuso clásico puede ser conveniente

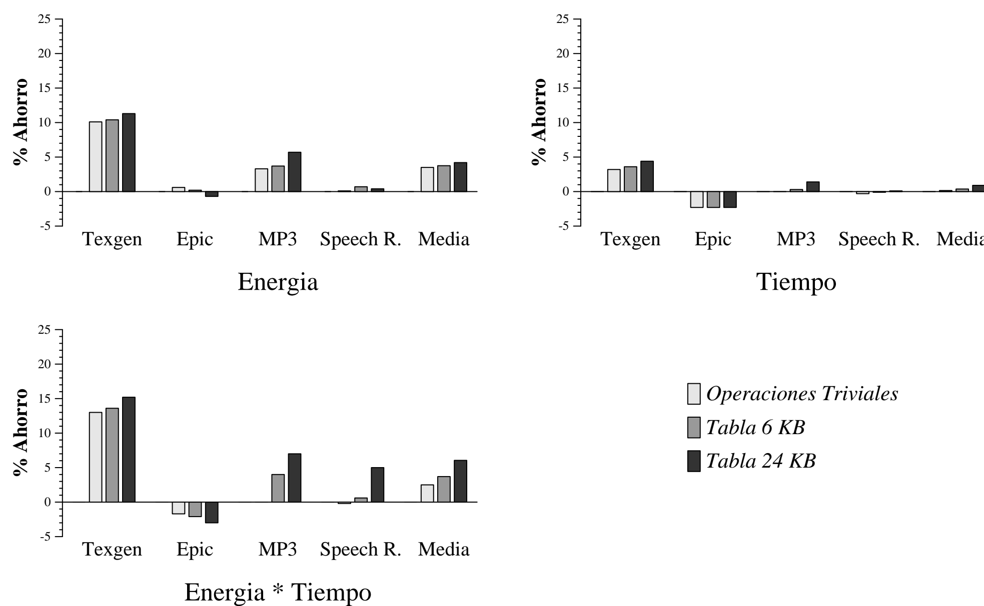


Figura 4.11: Resultados de la memorización clásica.

Programa	Operaciones Triviales	Tabla de 6KB	Tabla de 24KB
Epic	2,00	2,00	2,00
Texgen	30,89	30,91	30,96
MP3	18,99	19,00	19,01
R. Voz	4,50	4,50	4,50

Tabla 4.6: Porcentajes de aciertos del reuso clásico.

en algunos diseños, nuestra conclusión de este análisis preliminar es que un sistema consciente de la potencia debería implementar siempre un detector de operaciones triviales, aún para las operaciones de punto flotante.

La explicación de este comportamiento se encuentra en la tabla 4.6. Como se puede ver, la mayoría de aciertos obtenidos en el sistema de reuso clásico de operaciones de tipo flotante se deben a las operaciones triviales. Las tablas de reuso apenas son capaces de capturar algún reuso de cada 10000 instrucciones.

4.5.2 Nuestra propuesta: el reuso tolerante.

Hemos llamado reuso tolerante a nuestra propuesta de aplicación del calculo difuso al reuso. La idea de este tipo de reuso de basa en reusar instancias de la misma operación, no solo cuando sus operandos son idénticos sino

también cuando estos son parecidos.

En la memorización clásica, por ejemplo, si tenemos dos instancias de multiplicación, una con los valores de entrada 5'000 y 5'001 y otra con los valores 5'000 y 5'000, los resultados serían 25'005 y 25'000 (los mismos que en un sistema sin memorización). El hecho de añadir una tabla de reuso a la unidad de cálculo implica, en este ejemplo concreto, que además del cálculo se han de realizar 4 accesos a la tabla: dos para comprobar que los conjuntos de valores de entrada (5'000,5'001) y (5'000,5'000) no están almacenados, y dos más para actualizar la tabla con los resultados 25'005 y 25'000. Como consecuencia de ello, el resultado final será que a partir de este momento, nuevas instancias de la multiplicación con esos valores de entrada acertarán en la tabla (pero, por ejemplo, las entradas 5'001 y 5'001, no lo harán y volverán a fallar).

En nuestro esquema el primer acceso (5'000 y 5'001) también accederá y fallará en la tabla de reuso. A continuación la unidad de cálculo de punto flotante calculará el resultado normalmente (25'005) y actualizaremos la tabla con los valores 5'00, 5'00 y 25'005. Lo importante de este sistema es que no almacenamos los datos de entrada de forma exacta, sino aproximada (5'00 en vez de 5'000 o 5'001). La cantidad exacta de precisión que ignoramos depende del nivel de tolerancia que detallaremos más adelante. El resultado de esta imprecisión en los datos de entrada es que cuando tenemos la segunda instancia de la multiplicación con entradas 5'000 y 5'000, la tabla acertará y directamente obtendremos el resultado, ahorrando tiempo y energía. Como contrapartida, el resultado obtenido será 25'005 en vez de 25'000 que sería más correcto. A partir de este momento, cualquier acceso a la tabla con los valores (5'000,5'000) o (5'000,5'001) acertaría nuevamente en la tabla, pero accesos con valores similares (como el (5'001,5'001) comentado anteriormente) también acertarían. Además de ello, otra ventaja sobre el método clásico es que para acertar solo necesitaremos ocupar una entrada de la tabla y no dos, y por tanto podremos almacenar (y acertar) un rango mayor de valores.

Estructura hardware para el reuso tolerante.

¿Cómo se consigue ignorar una cierta precisión en los datos de entrada? La figura 4.12 muestra la estructura hardware de este sistema y, como se puede ver, una implementación simple es ignorar los últimos N bits de la mantisa de los operandos. El número de bits ignorados, N, es lo que denominamos el nivel de tolerancia. Este sistema tiene, además, la ventaja de que nos permite reducir aún más los costes respecto a la memorización clásica, ya que en un sistema donde toleramos N bits podemos reducir las tablas de

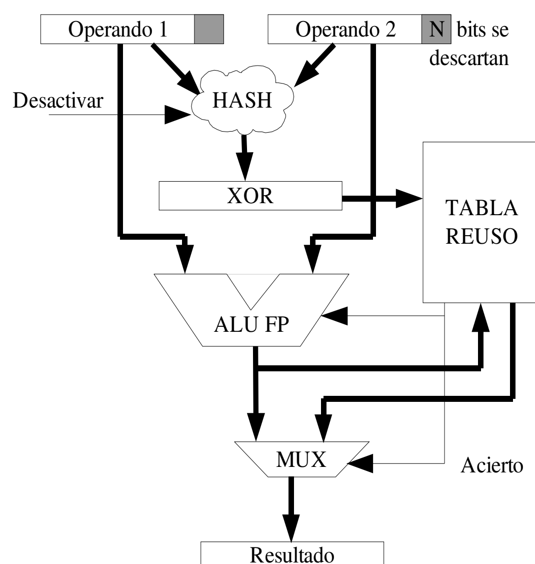


Figura 4.12: Estructura del sistema hardware de reuso tolerante.

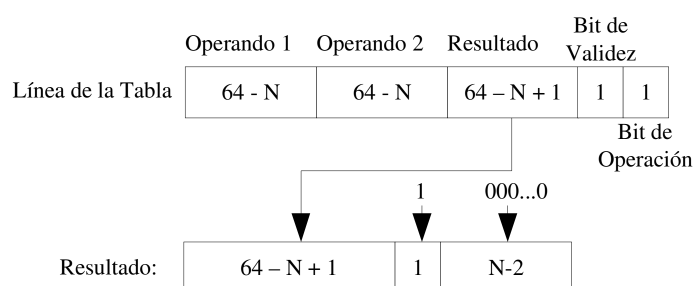


Figura 4.13: Mecanismo de relleno del resultado almacenado en la tabla tolerante.

reuso en $2 \times N$ bits, de forma que nuestro sistema es aún más eficiente (tablas menores consumirán menos o, con tablas de igual tamaño dispondremos de más entradas).

En el esquema mencionado se han de añadir, además, 2 entradas extras: una señal de desactivación (D) que permite que el reuso tolerante no funcione con algoritmos de tipo no tolerante (por ejemplo, en aplicaciones numéricas) y un registro donde se almacene el nivel de tolerancia N. Este nivel de tolerancia debería ser ajustable según la aplicación ejecutada, y debería poder decidirlo la propia aplicación (se puede determinar previamente mediante profiling), o bien el propio usuario mediante algún tipo de sistema de ajuste (calidad vs. duración de la batería).

Finalmente un punto más que se decidió mejorar después de las primeras

pruebas fue el tamaño de la salida almacenada en la tabla. Si, tal y como realizamos la tolerancia de las entradas, el resultado que guardamos no es exacto para los datos para los que lo vamos a usar, ¿por qué es necesario guardar el resultado exacto de la entrada original? Para averiguar esto se procedió a comprobar el error introducido en la señal de salida conforme se disminuían los bits de resultado almacenados. La conclusión de este experimento fue que para conseguir unos resultados con un error similar al introducido por reusar entradas con N bits de tolerancia era suficiente con almacenar los resultados con un bit de precisión más que las entradas. En la figura 4.13 se puede ver como el resultado almacenado se completa antes de escribirlo en el registro de destino de la operación: básicamente se rellenan con ceros todos los bits desechados, menos el primero que se pone a uno, de forma que obtenemos la media del rango de valores descartados. Este sistema permite reducir el error del resultado a practicamente el mismo que tendríamos con un sistema que guardase los resultados exactos. La ventaja de realizarlo de esta forma es que nos permite reducir aún más lo que ocupa cada entrada en las tablas de reuso tolerante, hasta una cantidad de bits igual a $(3 \times 64 - 3 \times N + 3)$, incluyendo los bits de validez y de tipo de instrucción memorizada (multiplicación y división).

Nivel de tolerancia.

El punto más delicado en nuestro sistema de reuso tolerante es evaluar el nivel de tolerancia que admiten los programas que vamos a utilizar. Ya se ha visto en la sección anterior, donde se evaluó la posibilidad de utilizar unidades funcionales de una precisión menor, que llega un punto en el que los programas directamente no funcionan. Para realizar esta evaluación se decidió medir la SNR introducida con diferentes grados de tolerancia (es decir, con una diferente cantidad de bits N desechados de cada operando almacenado en las tablas) para cada uno de los programas de prueba.

Las barras de los gráficos de la figura 4.14 muestran el error introducido para diferentes valores de N para cada uno de los 4 programas de prueba. Una SNR infinita implica ausencia de errores, mientras que una SNR de menos de 20 dB implica calidades bajas de señal. Antes de entrar a analizar los datos, sin embargo, la primera sorpresa que obtuvimos es que los programas aceptan reuso tolerante con niveles de tolerancia mayores que los resultados obtenidos para las unidades funcionales de menor precisión.

La respuesta al porqué de este comportamiento hemos de buscarla en la diferente forma en que funcionan ambas aproximaciones. El sistema con unidades funcionales de menor precisión introduce un determinado error en cada operación, independientemente de las veces que se realiza o de si es

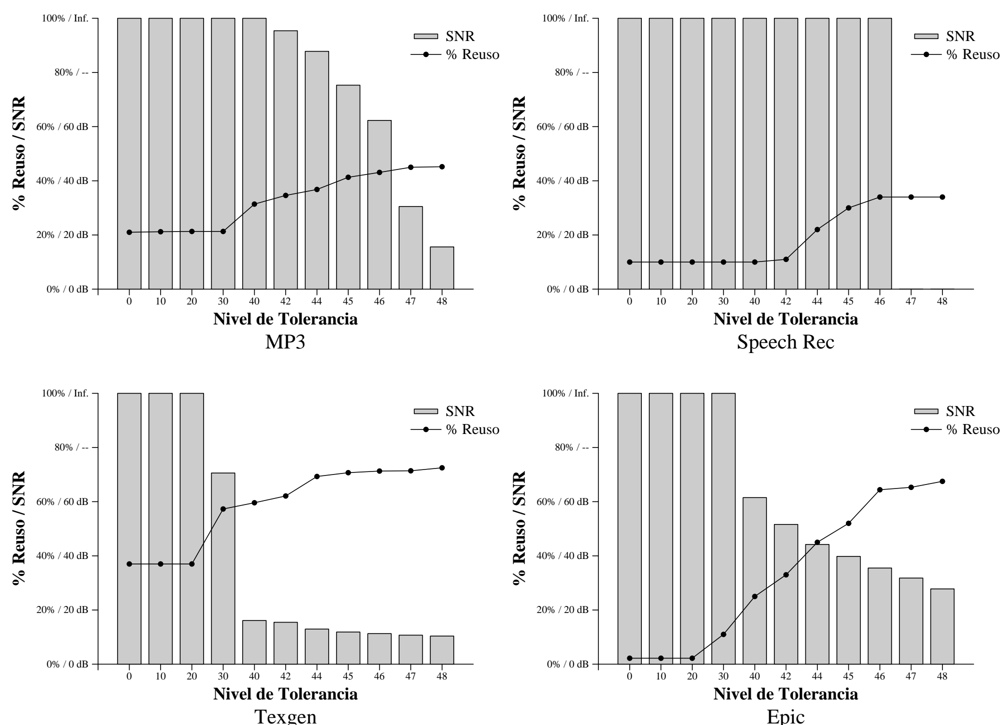


Figura 4.14: Error introducido vs. aciertos en las tablas de reuso.

la primera vez que se usa para, por ejemplo, ajustar un parámetro global de la transformación de señal que se va a realizar. El sistema de reuso aproximado, en cambio, la primera vez que se realiza una operación da un resultado exacto, ya que el acceso a la tabla falla y, por tanto, el cálculo debe realizarse y su resultado es correcto para todos los bits. Si tenemos en cuenta porcentajes de acierto realmente altos en las tablas, podemos hablar de un 70% de hits, y eso implica que, aún en el mejor de los casos, un 30% de las operaciones serán correctas y no tendrán más error que el normal de cualquier operación de coma flotante.

Para el resto de las operaciones el resultado se reusa de la primera operación introduciendo un error aunque los datos de entrada sean idénticos (no olvidemos que el sistema también desecha bits de la mantisa del resultado para reducir el tamaño de las tablas de reuso), pero el resultado almacenado no es la media de todos los resultados posibles para esa precisión (lo sería si almacenásemos el resultado medio entre la mínima y la máxima entradas con esa tolerancia) sino el resultado de una entrada que presumiblemente será más próxima a la actual que la media. La figura 4.15 muestra esta idea. El sistema de reuso tolerante almacena, de toda la esfera de entradas que se toleran hacia la misma posición de la tabla, no el valor medio (punto central negro) sino el resultado de una en concreto (punto de

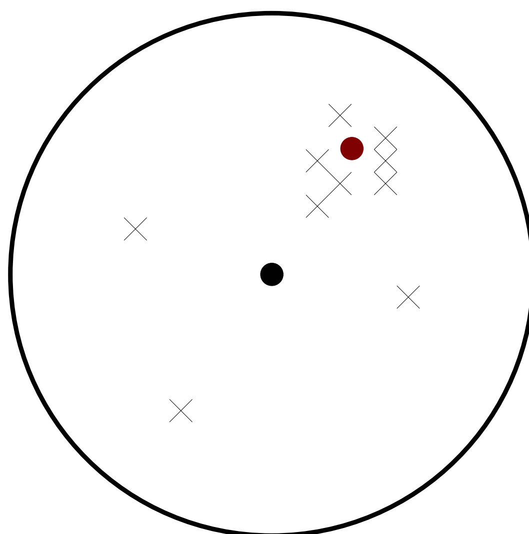


Figura 4.15: Distribución de los puntos reusados.

color rojo) que será más cercana a la mayoría de instancias de reuso (cruces) que la media, reduciendo así el error frente al sistema anterior.

La figura 4.14 muestra, además, para cada nivel de tolerancia, el porcentaje de aciertos que se alcanza con una tabla de 512 conjuntos de 2 entradas. Es importante poder comparar la evolución del porcentaje de aciertos (linea continua) con la evolución del error ya que se puede observar un efecto muy interesante que es que los aciertos crecen *antes* de que el error decrezca de manera significativa. Esta zona de tolerancias (desde unos 40 hasta unos 46 bits) es la que nos interesa como zona de trabajo, ya que en ella es donde obtendremos ganancias frente a la memorización clásica sin degradar excesivamente la señal.

En la tabla 4.7 se muestra el nivel de tolerancia que se puede aceptar desde un punto de vista subjetivo de la calidad de la señal. El caso más curioso es sin duda el del algoritmo reconocedor de voz ya que este cambia abruptamente desde un sistema perfecto a un sistema donde no realiza reconocimiento ninguno. Este tipo de comportamiento es el ideal ya que podemos ajustar con toda tranquilidad el nivel de tolerancia al máximo posible obteniendo grandes beneficios en cuanto a aciertos en la tabla de reuso (del 10 al 34%).

Otro resultado sorprendente (y muy conveniente) de la tabla 4.7 es la homogeneidad de los resultados: los niveles de tolerancia para calidades buenas o muy buenas son prácticamente los mismos independientemente de la aplicación. Esto implica que se puede optar por un tamaño reducido de

Programa	Calidad Subjetiva	Nivel de tolerancia
Epic	Muy Buena	0 - 45
	Buena	46
	Regular	47
	Mala	48 -
Texgen	Muy Buena	0-37
	Buena	37 - 43
	Regular	44 - 47
	Mala	48 -
MP3	Muy Buena	0 - 44
	Buena	45 - 46
	Regular	47 - 49
	Mala	50 -
Speech Rec	Muy Buena	0-46
	Mala	47 -

Tabla 4.7: Calidades subjetivas y niveles de tolerancia para diferentes aplicaciones.

la tabla de reuso (con capacidad, por ejemplo, para niveles de tolerancia a partir de 43) y utilizarla para todas las aplicaciones. En todos los experimentos restantes de este capítulo se ha considerado como calidad mínima aceptable la "buena" y se ha utilizado el mayor nivel de tolerancia posible que permitía mantener esta calidad.

Resultados del reuso tolerante en un procesador de bajo consumo.

En la figura 4.16 se pueden ver los resultados de ahorro de energía obtenidos mediante el sistema de reuso tolerante para la unidad de punto flotante en la

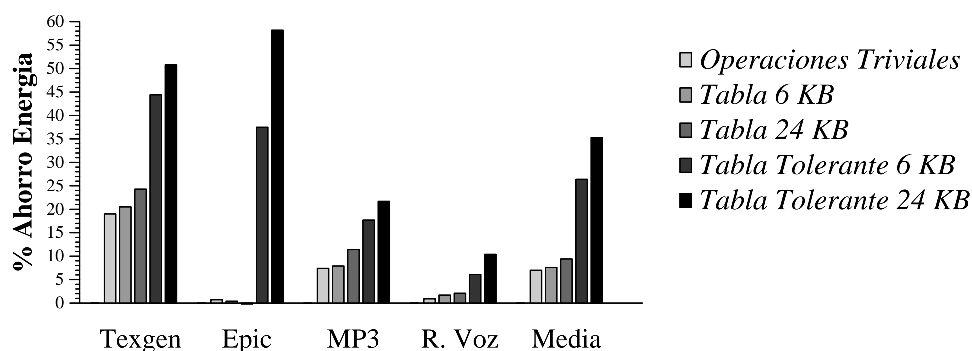


Figura 4.16: Ahorros de energía en la unidad de coma flotante.

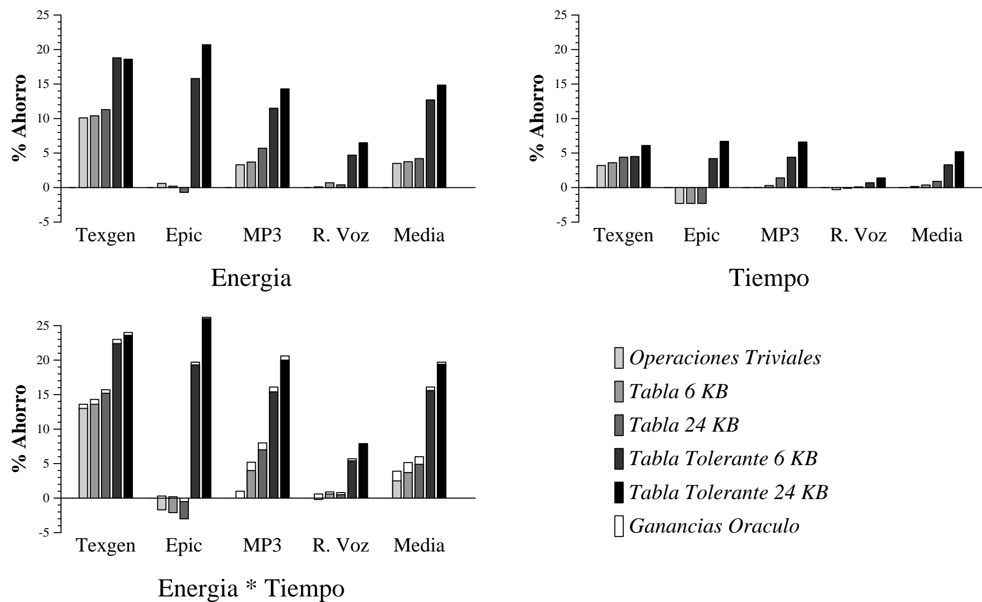


Figura 4.17: Ahorros de tiempo y energía obtenidos mediante reuso tolerante.

ejecución de cada uno de nuestros programas de pruebas. Las tres primeras barras corresponden a los ahorros obtenidos mediante el sistema de reuso clásico, para el sistema con detección de operaciones triviales, y tablas de 6 y 24 KB respectivamente. Las dos últimas barras muestran los resultados con nuestro sistema de reuso tolerante y tablas de un tamaño equivalente a las usadas en el reuso clásico. Como se puede ver, los resultados mejoran sustancialmente utilizando el reuso tolerante. Aun cuando comparamos los resultados de la tabla de 6 KB con la tabla reuso clásico de 24 KB los resultados aumentan de un 9'4% a un 26'4%, es decir, 3 veces más ahorro con 4 veces menos espacio.

Los resultados de ahorro de energía y tiempo para el programa completo pueden verse en la figura 4.17. Lógicamente en este caso los ahorros son menores que en la figura 4.16 ya que existen numerosas partes del programa donde no se ahorra, pero sin embargo, los resultados siguen siendo importantes debido a la enorme incidencia que tiene el subsistema de coma flotante en el gasto de energía. No hay que olvidar que todo el procesado de la imagen tiene lugar en coma flotante y por tanto su incidencia es decisiva. La figura 4.17 es una ampliación de la figura 4.11 donde se han añadido los resultados para las tablas de reuso tolerante de 6 y 24 KB. Como se puede ver, los ahorros de energía mejoran de forma notable, pero además, el sistema de reuso tolerante, al contrario que el clásico, presenta ahorros de tiempo de ejecución en todos los programas de pruebas, es decir, su resul-

tado es siempre positivo. Esta diferencia cualitativa puede ser vital para los sistemas de bajo consumo que habitualmente no funcionan con demasiado margen de capacidad de cálculo en aplicaciones que requieren resultados en tiempo real (como el vídeo o el audio).

En la figura 4.17 se puede observar, además, una barra blanca superpuesta a los resultados obtenidos. Esta barra muestra los resultados obtenidos mediante el uso de un oráculo ideal que solo accede a la tabla de reuso con aquellos valores que ya sabe que van a acertar. De esta forma en los valores que fallaremos no introducimos el ciclo extra que se usa para determinar si acertamos o no en la tabla. Esta barra es, pues, el máximo de ahorro que podríamos obtener mediante la tabla de reuso. Como se puede ver por la gráfica, en todos los casos, el ciclo extra de latencia en los fallos de la tabla es un precio pequeño (por lo que no vale la pena usar algún tipo de sistema extra para predecir posibles aciertos o fallos por instrucción estática, por ejemplo) pero, también aquí, el sistema de reuso tolerante es superior al clásico ya que solo se aleja, en media, un 0'5% del ideal, frente al 1% que se aleja el reuso clásico.

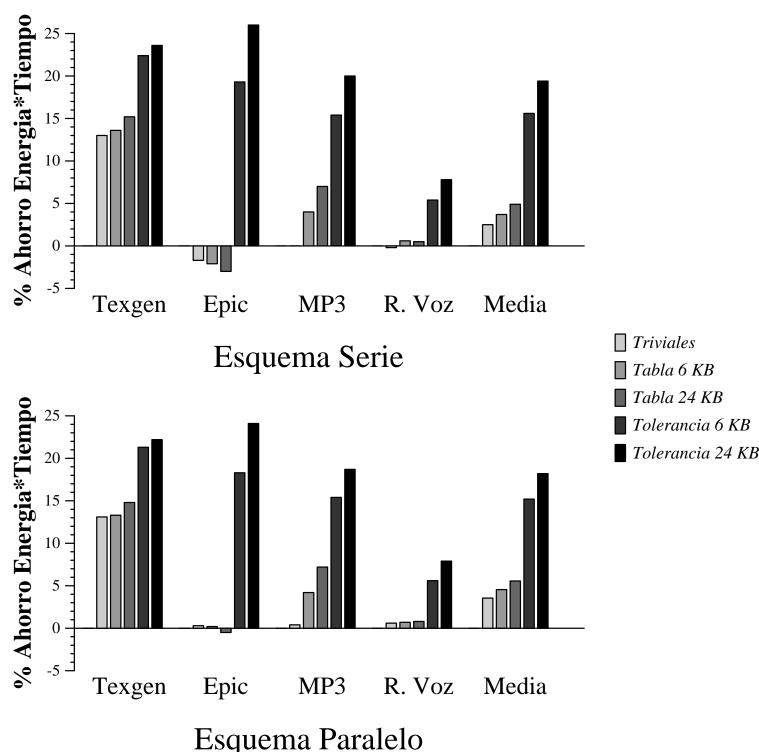


Figura 4.18: Configuración en Serie vs. configuración en Paralelo.

Finalmente, la figura 4.18 muestra los resultados obtenidos en la medida de Tiempo*Energía para la configuración en serie (que hemos usado para

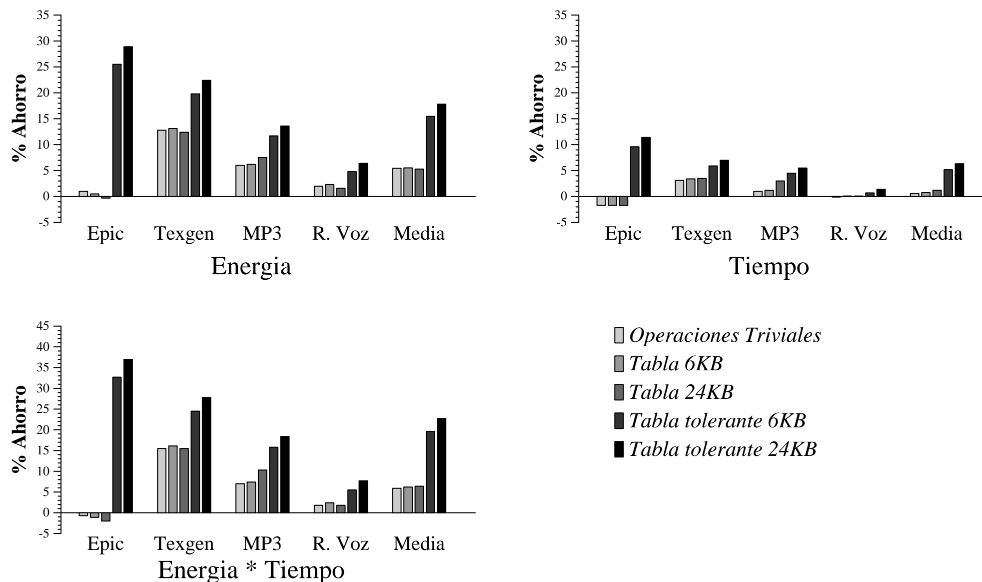


Figura 4.19: Procesador de ancho 1.

todas las configuraciones anteriores) frente a la configuración en paralelo. El propósito de este experimento es verificar que nuestra decisión de implementar la tabla hardware en secuencial ha sido correcta. Como se puede ver, las diferencias son pequeñas en cualquier caso (ya se ha visto que un sistema sin accesos erroneos a la tabla de reuso no alcanza ganancias mucho mayores), pero como regla general, se puede decir que para sistemas con un porcentaje de aciertos bajos (reuso clásico u operaciones triviales) el sistema en paralelo presenta resultados ligeramente mejores, mientras que para un porcentaje de aciertos alto (reuso tolerante) es el sistema serie el que mejora en algo los resultados. Esta diferencia se puede ver intuitivamente si pensamos en el caso del detector de operaciones triviales. Si lo implementamos en serie con el sistema, de forma efectiva estaremos aumentando en uno la latencia de la operación para todas aquellas instrucciones que no sean triviales (es decir, la gran mayoría) y este efecto es muy importante. En cambio, implementado en paralelo tendremos un sistema que apenas gasta energía extra y que en un porcentaje de casos significativo reduce en mucho la latencia de la operación.

Resultados del reuso tolerante en diferentes procesadores.

Finalmente hemos estudiado el resultado de implementar el reuso tolerante en los tres tipos de procesadores estudiados. Las gráficas de las figuras 4.19, 4.20 y 4.21 muestran los resultados obtenidos. Los resultados de la figura

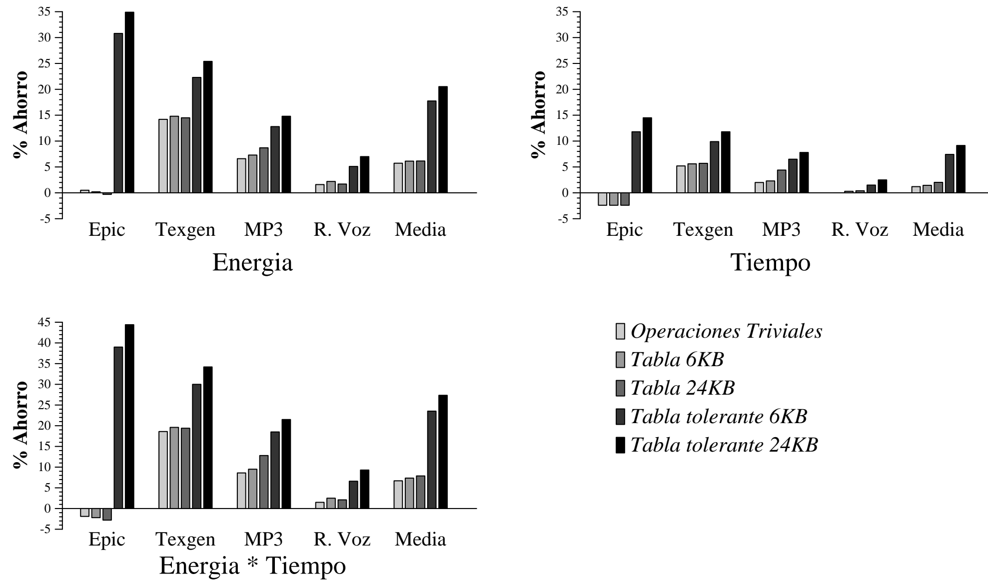


Figura 4.20: Procesador de ancho 2.

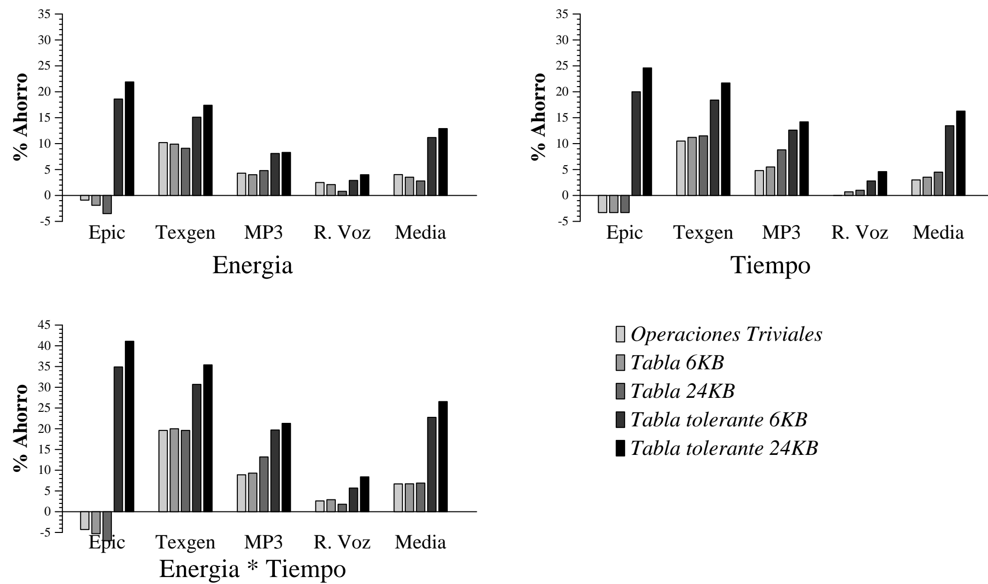


Figura 4.21: Procesador de ancho 4 fuera de orden.

4.20 no coinciden de forma exacta con los mostrados en la figura 4.17 ya que en este caso no se ha utilizado una tolerancia fija adecuada para todos los resultados sino que se ha utilizado la tolerancia máxima posible para cada programa de forma independiente (y por tanto los resultados son ligeramente mejores en estas gráficas.

Lo primero que se puede observar en las figuras 4.19, 4.20 y 4.21 es que, contrariamente a lo que cabría esperar en principio, los mejores resultados no se encuentran en el procesador más simple, sino en el intermedio. Este efecto se debe, como ya hemos comentado anteriormente, a que las ganancias en tiempo de procesado son más significativas en el procesador más agresivo, precisamente debido a que su tiempo de ejecución total es mucho menor. Sin duda, el hecho de que el procesador más simple tenga una unidad de punto flotante tan potente también ayuda a mantener más limitadas las ganancias del sistema de bajo consumo, ganancias que serían mayores en el caso de una implementación más real.

Otra conclusión que se puede extraer a la vista de las gráficas es que la configuración en serie no debe emplearse si se desea implementar un mecanismo simple de detección de operaciones triviales ya que este sistema es capaz de ralentizar incluso al procesador más agresivo. En un sistema que se limitase a implementar un detector de operaciones triviales, este debería implementarse en paralelo con el multiplicador (y debería, por tanto, ser capaz de apagar el multiplicador en caso de acierto).

4.6 Unidades funcionales difusas y reuso.

Nuestra última propuesta consiste en combinar las dos técnicas presentadas anteriormente para conseguir mayores ahorros conjuntos. En este sentido, la alternativa ideal desde nuestro punto de vista es combinar el uso de un nuevo tipo de datos especial para multimedia (SPDR) con nuestra propuesta de reuso tolerante. Esta combinación es especialmente atractiva ya que un dato del tipo SPDR es un dato que ya se ha declarado con una intención específica y por tanto es un candidato ideal para ser reusado mediante nuestra propuesta.

Las figuras 4.22, 4.23 y 4.24 muestran los resultados de ahorro en tiempo, energía y tiempo×energía para cada uno de los procesadores estudiados. Cada grupo de barras muestra los resultados individuales para cada programa y la media obtenida con todas las técnicas mencionadas. Los dos primeros grupos muestran los resultados de utilizar solamente operaciones triviales, con el detector usado en serie y en paralelo, respectivamente, con

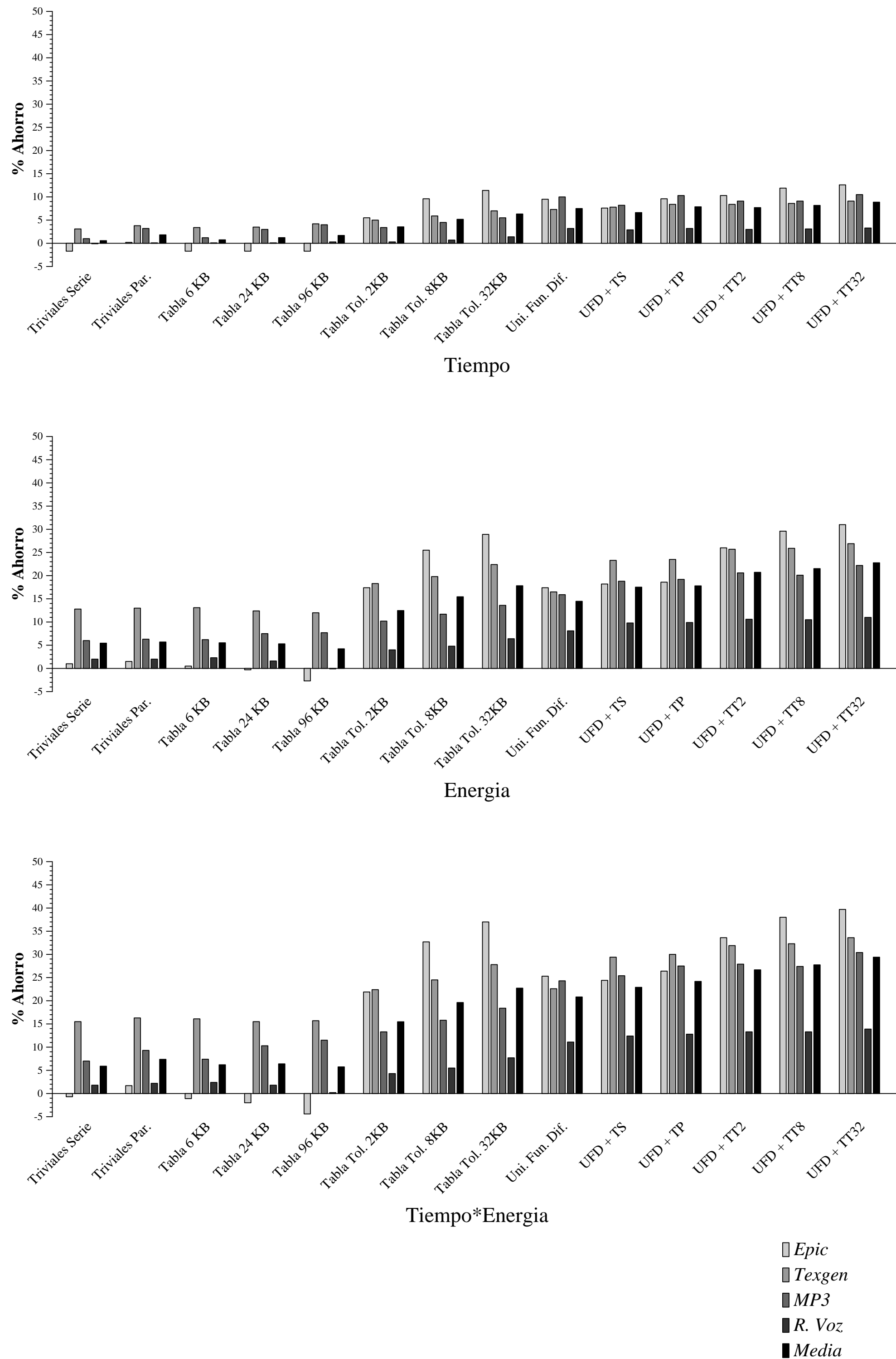
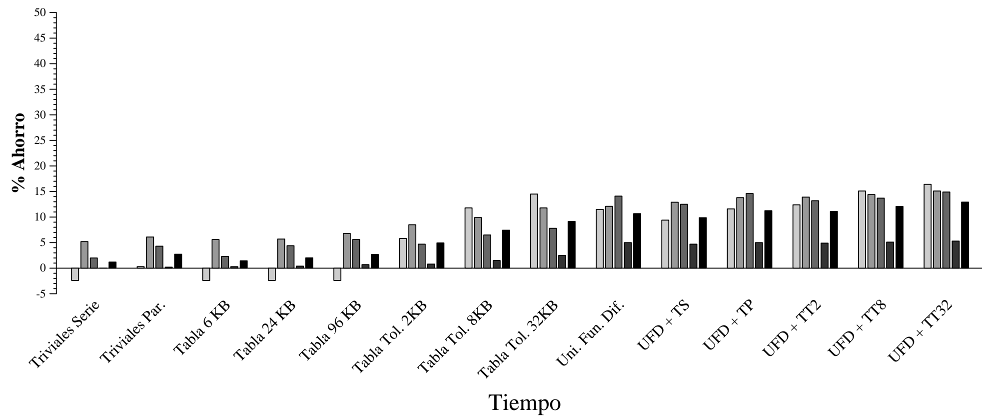
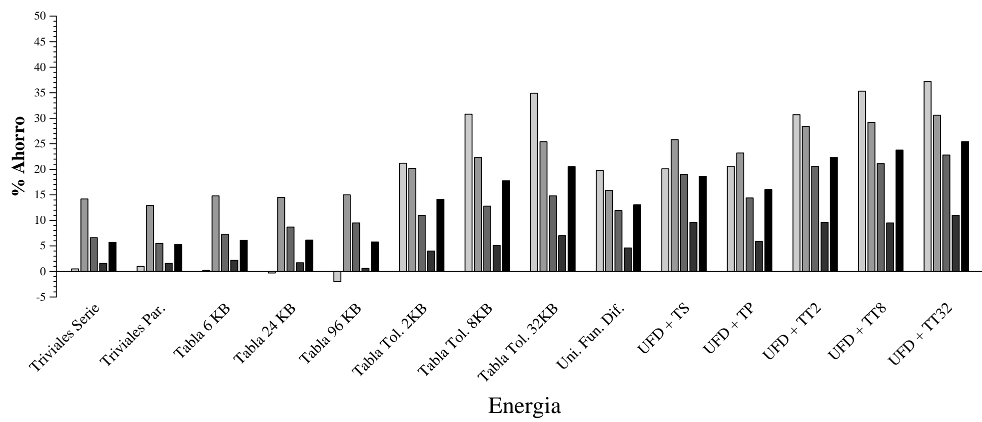
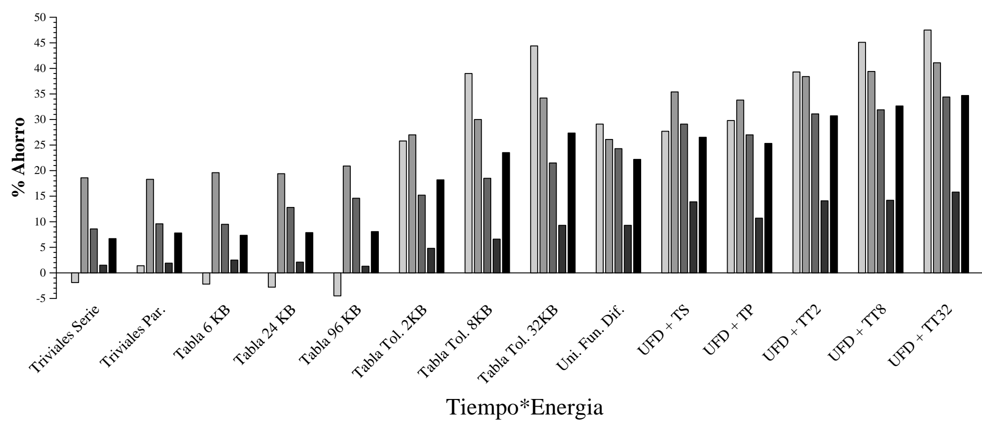


Figura 4.22: Ganancias en el procesador de ancho 1.

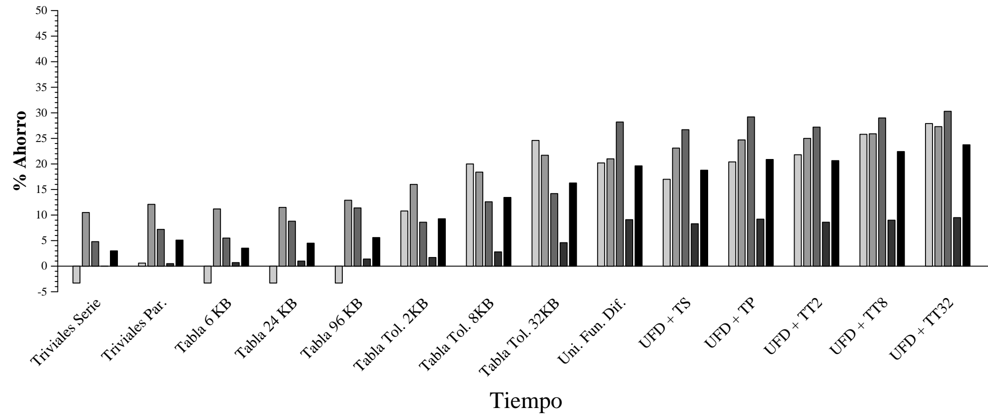


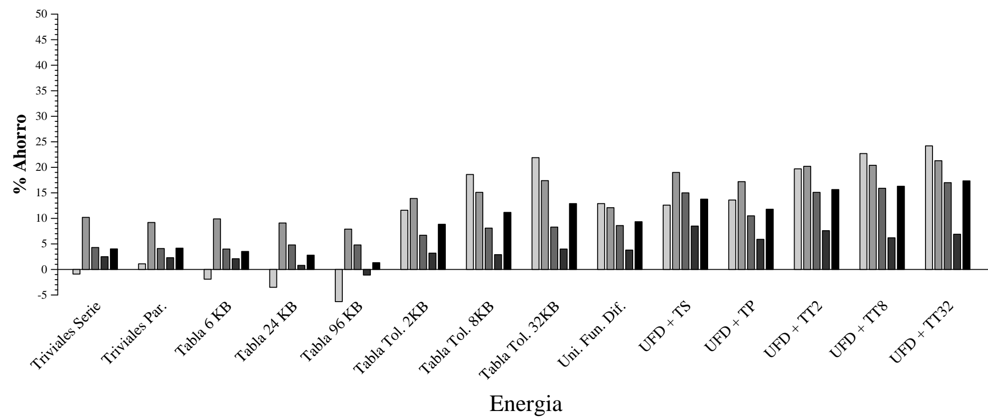




Epic
Texgen
MP3
R. Voz
Media

Figura 4.23: Ganancias en el procesador de ancho 2.





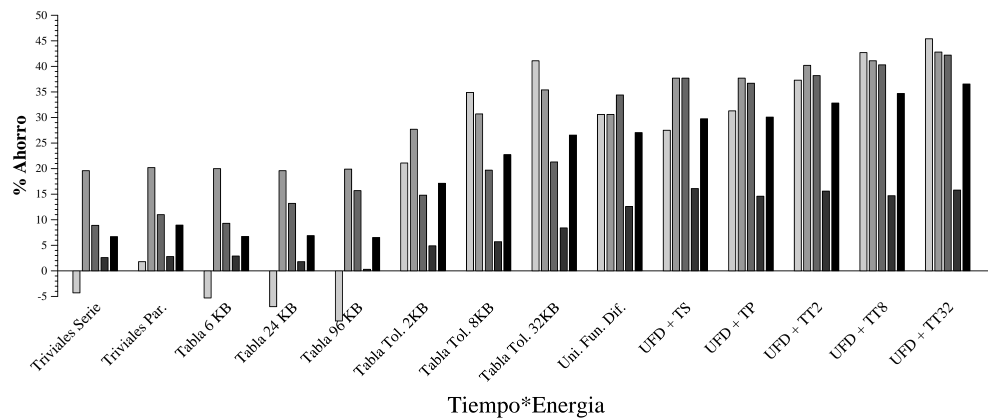


Figura 4.24: Ganancias en el procesador fuera de orden de ancho 4.

el sistema de cálculo propiamente dicho. Como puede verse, el sistema paralelo es mucho más eficiente que el sistema serie, principalmente debido al ciclo extra que introduce el sistema en serie en caso de que la operación no sea trivial y que tiene la capacidad de retrasar todo el programa.

Los siguientes 3 grupos muestran los resultados obtenidos mediante el reuso clásico con tablas de 256, 1024 y 4096 entradas (6, 24 y 96 KB), respectivamente. Como se puede ver en las gráficas, el sistema de reuso clásico no es eficiente para la memorización de instrucciones en punto flotante. La baja tasa de aciertos conseguida provoca que las ganancias obtenidas por estos se vean compensadas por el ciclo extra de latencia y el consumo de las tablas, de forma que el resultado llega a ser negativo desde el punto de vista del rendimiento y de la eficiencia energética.

Los tres grupos siguientes de barras muestran los resultados obtenidos mediante tablas de reuso tolerante de, también, 256, 1024 y 4096 entradas respectivamente. Es importante notar que, debido a que no necesitamos memorizar con toda precisión los valores de entrada ni el resultado, en este caso las tablas ocupan mucho menos espacio que en el caso del reuso clásico (2, 8 y 24 KB respectivamente). Además, en este caso los resultados son siempre positivos, más cuanto más grande es la tabla de reuso ya que el número de aciertos en la tabla crece lo suficiente con el tamaño de la tabla como para compensar su gasto en energía. Para sistemas de bajo consumo nuestra recomendación, sin embargo, sería la tabla de 8 KB que, con un tamaño acotado, proporciona unas buenas ganancias.

Los siguientes tres resultados muestran las ganancias del sistema de unidades funcionales difusas. Como hemos dicho antes, hemos supuesto que se ha optado por el sistema que incorpora un nuevo tipo de datos de 32 bits con el rango de la doble precisión (11 bits) de cara a aunarlo a continuación con el reuso tolerante, pero los resultados prácticamente no difieren de los de la otra posibilidad propuesta (una unidad que utilice el sistema de simple precisión para calcular los resultados de doble y que, por lo tanto, solo sea 29 bits más corta y no 32). El primer grupo de resultados es el sistema con las unidades funcionales difusas y los dos siguientes muestran los resultados si se incorpora además la detección de operaciones triviales en serie y en paralelo con la nueva unidad. En este caso, como se puede ver, no existe una solución óptima, ya que ambas configuraciones dan resultados muy similares. Sin embargo, dado que nuestra propuesta es que siempre se incorpore en los sistemas de cálculo conscientes del consumo un detector de operaciones triviales, proponemos la configuración en paralelo, ya que el detector puede ser común para todas la operaciones y todos los tipos de la unidad de punto flotante.

Finalmente, los tres últimos grupos de resultados muestran la combi-

nación de utilizar unidades funcionales difusas y reuso tolerante con los mismo tamaños de tablas ya mencionados (2, 8 y 32 KB). El hecho de juntar ambas aproximaciones da lugar a unos resultados sensiblemente mejores a los obtenidos por ambos métodos por separado, llegando a reducciones en el factor tiempo×energía del 36,5% de media en el conjunto de los programas de prueba para el procesador fuera de orden y a cerca del 30% en el procesador de ancho 1. El hecho curioso de que las ganancias totales sean más significativas en el procesador más agresivo se debe, como ya se ha comentado, a que debido a Amdahl, la incidencia de las operaciones de punto flotante es mucho más significativa cuando todo el programa se ejecuta muy rápido que cuando el resto del programa es muy lento. Las ganancias en energía, sin embargo, son mejores en los procesadores más simples (verdadero objetivo de nuestras propuestas).

4.7 Conclusiones.

En este capítulo se presentan dos diferentes métodos de implementación del cálculo difuso de instrucciones: el uso de unidades funcionales de menor precisión y el sistema de reuso tolerante.

Las unidades funcionales de menor precisión permiten constatar que el cálculo difuso puede realizarse y que el error introducido por este sistema puede ser, tal y como nos interesa, irrelevante. Se han introducido además, dos propuestas de aplicación, de similar incidencia en la ejecución de los programas, pero de diferente filosofía: utilizar la unidad de simple precisión para realizar los cálculos de doble por un lado; e introducir un nuevo tipo de datos con el rango de doble precisión y la precisión de la simple precisión por otro (en realidad la precisión es un poco menor debido a los 3 bits de más, necesarios para almacenar el exponente). La primera propuesta permite, de manera muy simple, obtener grandes ganancias en la ejecución de los programas multimedia actuales en cualquier tipo de procesadores. La segunda, en cambio, apuesta por introducir un nuevo tipo de datos que permita un mayor control de los programadores sobre el comportamiento de sus programas y, por tanto, a pesar de ser desde nuestro punto de vista la aproximación ideal, depende de estos para ser plenamente eficaz y su proceso de implantación es mucho más lento.

El reuso tolerante, en cambio, permite ampliar enormemente las posibilidades del reuso clásico, haciendo que, aplicado a las instrucciones de punto flotante, pase de ser un sistema con ganancias marginales a ser un sistema con un gran potencial, plenamente capaz de ser implementado en un procesador comercial. Un sistema de resuso tolerante permite alcanzar

reducciones del 25% en el producto energía$\times$tiempo en varios de los programas de pruebas sin degradaciones significativas (apreciables) en la calidad del resultado y empleando tablas de reuso de tamaños perfectamente implementables (8 Kbytes).

Finalmente hemos combinado las técnicas propuestas para ver que se pueden alcanzar ganancias cercanas al 50% en el producto tiempo$\times$energía en programas intensivos desde el punto de vista del cálculo en coma flotante. Incluso la inclusión de una modesta tabla de reuso tolerante de 2KB es capaz de aumentar las ganancias del sistema con unidades funcionales difusas en más de un 5%.

5

Cálculo difuso de regiones

Resumen

En este capítulo se explica como se implementa el cálculo difuso de regiones mediante el reuso tolerante de regiones. En primer lugar se parte del reuso clásico de regiones y se muestra como este es insuficiente para las características de los algoritmos multimedia. A continuación se muestra como ampliar las capacidades del sistema mediante el reuso tolerante y se evalúan sus resultados. Finalmente se propone un hardware de ajuste dinámico de la tolerancia capaz de ajustar de forma automática la tolerancia y se miden sus diferencias con respecto al reuso clásico y al reuso tolerante ajustado de forma manual. Las conclusiones finales explican las ventajas de este nuevo enfoque tanto frente a las técnicas anteriores como frente a las desarrolladas en un principio en esta tesis.

5.1 Introducción.

En el capítulo anterior hemos visto como se pueden aplicar los criterios de cálculo difuso a instrucciones individuales del procesador. Sin embargo esta aproximación presenta algunos inconvenientes. Quizás los más llamativos sean la poca cantidad de energía que podemos ahorrar en cada reuso y la falta de control que tenemos sobre todo el proceso.

Como suele suceder, lo que es, por un lado, una gran ventaja para el sistema (transparencia al programador, sencillez de implementación) también es su mayor inconveniente (poco control sobre el error introducido, incapacidad de saber si se opera realmente sobre los datos).

En este capítulo realizaremos una aproximación diametralmente opuesta al cálculo difuso: intentaremos aplicarlo a grandes regiones de código que se encuentren en los núcleos de los programas multimedia. Esto implicará que nuestro sistema necesitará la ayuda de un programador o, como mínimo, de un compilador altamente desarrollado.

Como contrapartida a dicha necesidad, el calculo difuso de regiones operará siempre sobre los datos del programa (ya que siempre seleccionaremos regiones adecuadas) y además podrá alcanzar mayores ahorros de energía, ya que una región equivaldrá a varias instrucciones.

La primera propuesta de implementación, estudiada en este capítulo, de cálculo difuso de regiones es el reuso tolerante de regiones. Esta propuesta, similar al reuso tolerante de instrucciones, tiene la ventaja de basarse en el reuso clásico de regiones, un sistema que por sus propias características se adapta a regiones que contienen cálculos de muy diversa índole sin necesidad de utilizar hardware reconfigurable.

5.2 Trabajo relacionado.

El reuso de regiones es un intento de explotar la localidad de valores que se da para ciertos conjuntos de instrucciones de un programa[1]. El ámbito de aplicación de dicho reuso puede ser muy diversa: bloques básicos, trazas o incluso funciones completas pueden ser candidatas a ser reusadas.

Es por esto que hay un gran número de artículos que se centran en el tema del reuso "grueso". Como ya se ha comentado, hace tiempo que se

[1] Dichos conjuntos de instrucciones se suelen repetir mucho, es decir, suelen estar en los bucles internos de los códigos

Algoritmo	Descripción	Datos	Características
JPEG	Compresor de imagen	penguin, vigo, specmun.ppm	Diferentes mapas de bits
H263	Compresor de vídeo	input_base.263	Vídeo de una joven
GSM	Compresor de voz	clinton.pcm, probando.pcm	Secuencias de voz

Tabla 5.1: Programas de prueba utilizados.

utilizan técnicas de memorización para evitar calcular dos veces funciones con los mismos parámetros y muchos compiladores las usan para optimizar la comprobación de las cadenas de dependencias de los programas a compilar.

En [SS97] se propone un mecanismo de reuso de instrucciones adicional basado en enlazar instrucciones con dependencia de datos en la tabla de reuso, intentando así explotar el reuso en el ámbito de la cadena de dependencias. En [HL99], el reuso de regiones se explota a nivel de bloques básicos (y es llamado, consecuentemente, reuso de bloques) y en [GTM99] el reuso se explota a nivel de trazas.

Connors et al. en [CmWH99] y [CHCmWH00], han propuesto un mecanismo híbrido donde el compilador es el responsable de identificar regiones de instrucciones donde se encuentra una cantidad significativa de localidad de valores (gracias, principalmente a profiling software[CmWH99] o hardware[CHCmWH00]). Una vez identificadas las regiones, instrucciones especiales utilizaban tablas hardware para reusar diferentes instancias de dichas regiones. Finalmente, en [SBS00] hay un interesante estudio teórico que identifica el reuso potencial en función de la cantidad de instrucciones para diferentes benchmarks, en el contexto de la optimización dinámica.

5.3 Los programas estudiados.

Los programas estudiados para el reuso de regiones han sido escogidos entre una selección de MediaBench [LPMS97] y MediaBench II [Con05], de forma que representasen un amplio espectro de aplicaciones multimedia. Así pues se ha escogido un compresor de imágenes estáticas JPEG (*cjpeg*), un compresor de imágenes dinámicas (*tmn*) que implementa el algoritmo definido en la norma H263 y un compresor de voz GSM (*toast*), es decir, el sistema usado en todos los dispositivos de comunicaciones móviles. La tabla 5.1 muestra un pequeño resumen de las características de estos programas.

Como se puede ver, cuando ha sido posible se ha utilizado más de un juego de entrada para intentar diferenciar entre características debidas a propiedades de las señales de entrada y las características intrínsecas al

Señal	Descripción	Características
penguin.ppm	Un hombre mirando un pingüino	Imagen color 1024×739, 24 bits por píxel.
specmun.ppm	Un grupo de amigos reunidos al aire libre	Imagen color 1024×688, 24 bits por píxel.
vigo.ppm	Una fuente ornamental	Imagen color 1024×768, 24 bits por píxel.
input_base.263	Vídeo de una chica andando por la calle	Vídeo de 8 escenas de 704×576, 24 bits por píxel.
clinton.pcm	Extracto de un discurso de Bill Clinton	116 secuencias de 160 muestras de voz de 16 bits.
probando.pcm	Típica secuencia de prueba	116 secuencias de 160 muestras de voz de 16 bits.

Tabla 5.2: Características de los datos de prueba.

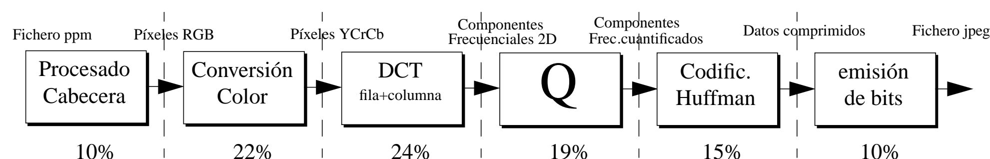


Figura 5.1: Etapas del codificador JPEG.

algoritmo. La tabla 5.2 muestra una breve descripción y las propiedades principales de los datos de entrada.

5.4 Reuso de regiones para multimedia.

Tal y como ya se muestra en varios trabajos previos ([CmWH99], [SBS00], [CHCmWH00]) el principal problema que se encuentra al tratar de realizar reuso de grupos de instrucciones (regiones) es la identificación de las susodichas regiones. El sistema de identificación tiene que enfrentarse al dilema de escoger entre la longitud de la región y su localidad. De hecho, todavía hay una investigación activa en este campo: [CmWH99], [SBS00], [GTM99], [CHCmWH00].

Sin embargo, las aplicaciones multimedia pueden presentar una vertiente nueva en lo que se refiere a la identificación de regiones. Muchas de las aplicaciones multimedia (especialmente las que involucran procesado de audio, vídeo o imagen) se caracterizan por utilizar un conjunto de algoritmos conocido (transformadas de frecuencia, convoluciones, transformadas bilineales, etc.) de forma secuencial (se puede ver un ejemplo en la figura 5.1). Practicamente cualquier aplicación multimedia puede, pues, ser descrita como una secuencia de etapas de alto nivel a través de las cuales los datos se procesan por grupos.

Nuestra idea es que cada una de estas etapas de alto nivel es un buen

objetivo para poder explotar el reuso de regiones. Estos algoritmos (los que definen cada etapa) se caracterizan por ser muy regulares (realizan el mismo procesado a cada uno de los grupos de datos) y por procesar estructuras de datos reconocibles (es decir, píxeles, muestras de audio...) que presentan una alta localidad de valores (pensemos en el cielo de una imagen, o en el agua, o...). Los desarrolladores de código para DSPs pueden identificar fácilmente regiones con un gran reuso potencial y conseguir usar de forma muy eficiente cualquier facilidad hardware que se introduzca en la arquitectura de un procesador DPS genérico (con o sin la ayuda de un compilador). Por si esto fuera poco, algunos trabajos ([CHCmWH00]) ya han demostrado la capacidad de los compiladores para detectar dichas regiones de forma autónoma, simplificando aún más el proceso.

En los siguientes apartados intentaremos demostrar que la mayoría de las etapas de procesado de algunos de los algoritmos más comunes son susceptibles de utilizar el reuso de regiones.

5.5 Hardware para el reuso de regiones.

A la hora de decidir una implementación para el reuso de regiones, se procedió a estudiar los diferentes algoritmos a reusar hasta decidir que características necesitaba un sistema de este tipo. Era importante definir parámetros como la cantidad de entradas y salidas que podría tener como mucho el sistema, el tamaño de estos datos y el tamaño final de la tabla. Asimismo era necesario definir un nuevo ISA que permitiera gestionar el hardware implementado de una forma, a poder ser, fácil y eficiente.

Después de diversos estudios se decidió implementar un sistema que admitiese 4 valores de entrada y salida distintos. Esta decisión está vinculada a las características propias de los algoritmos. Por un lado, 4 valores permiten adaptar el sistema a regiones que operan en bloques de datos de 8 (típicos de imagen y que suelen tener dos partes con 4 entradas y salidas cada una) y, por otra, el número de aciertos en las tablas decrece de forma significativa con cada nuevo valor de entrada (lógicamente, las posibles combinaciones aumentan de forma exponencial con la cantidad de valores de entrada). Así, 5 valores o más daban lugar a índices de aciertos demasiado bajos en las tablas, mientras que tres valores o menos limitaban mucho las regiones en las que se podría implementar nuestra solución.

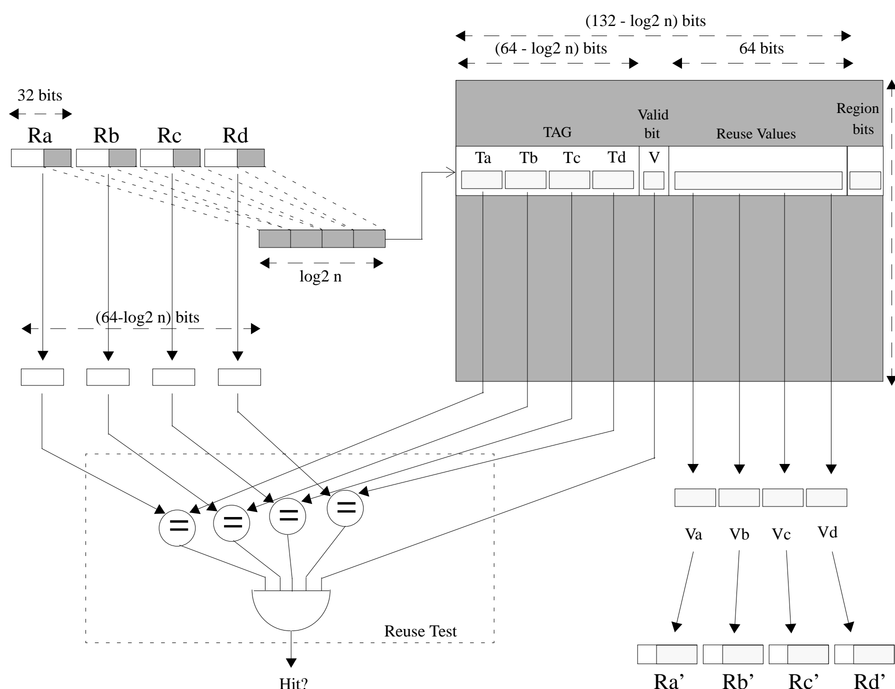


Figura 5.2: Mecanismo propuesto para reuso convencional de regiones.

La tabla de reuso.

Con todas las limitaciones comentadas en mente, finalmente se optó por la solución que se refleja en la figura 5.2. Como se puede ver, el mecanismo mostrado incluye una tabla de acceso directo (se han evaluado también soluciones con tablas asociativas de ancho dos o cuatro) a la que se accede mediante un máximo de 4 registros. La posición de acceso se calcula mediante los bits de menor peso de los valores de entrada. Si hay concordancia entre los valores de entrada y la etiqueta almacenada y el bit de validez está activado, se produce un acierto en la tabla y esta devuelve los valores de salida que la instrucción apropiada se encargará de escribir en los registros de destino.

Si, por el contrario, no hay acierto en la tabla, un registro interno se encarga de memorizar la posición accedida. En el siguiente acceso a la tabla se actualizará dicha posición con los valores correctos que el programa ha calculado mediante el código original.

Finalmente comentar que cada valor de acceso a la tabla puede tener como mucho 16 bits. Esto se ha decidido así porque, por un lado, representa un ahorro de espacio significativo frente a usar los usuales 32 bits (cada linea de la tabla ocupa la mitad) y por otro, realmente no tiene mucho sentido

usar la tabla de reuso de multimedia con valores tan grandes. Las muestras de voz o imagen suelen tener 8 o, como mucho 16 bits, ya que este tamaño es más que suficiente para representar todo el espectro (el color “verdadero”, por ejemplo, ocupa 24 bits, divididos en 3 componentes, RGB de 8 bits cada una). Así pues, una linea de la tabla de reuso implementada ocupará 129 bits.

El ISA.

Para poder utilizar el hardware explicado necesitamos introducir nuevas instrucciones en el ISA del procesador a modificar. Estas instrucciones deben ser capaces de:

- Cargar los parámetros de acceso a la tabla de reuso.
- Acceder a la tabla y verificar si ha habido un acierto o un fallo.
- En caso de acierto en la tabla, cargar la salida en los registros de destino y saltar el código que originalmente realizaba el cálculo.
- En caso de fallo ejecutar el código que realizaba el cálculo y cargar en la tabla los resultados adecuados cuando dicho cálculo esté finalizado (lo cual, desde el punto de vista del procesador, puede querer decir bastante tiempo, medido en instrucciones).

Este tipo de comportamiento nos da dos problemas a la hora de implementarlo de forma práctica:

- El número de parámetros de la tabla, tanto de entrada como de salida, puede ser mayor de la cantidad de datos que podemos utilizar en una instrucción.
- El acceso de consulta de la tabla y el acceso de actualización pueden estar muy separados en lo que a instrucciones se refiere.

Así pues hemos optado por separar totalmente la parte de lectura de parámetros (o de escritura) del acceso a la tabla, dando lugar así a varias instrucciones distintas que han sido implementadas en el simulador:

- LDT, Load Table: Esta instrucción carga N valores en los registros de entrada o salida de la tabla de reuso.

- LACT, Load and Access Table: Esta instrucción carga N valores en los registros de entrada de la tabla de reuso y realiza el acceso a la tabla. Si el acceso es un acierto el programa continuará, si es un fallo realizará un salto a la dirección indicada (y que será la del código original).
- WRT, Write Result from Table: Esta instrucción carga en M registros de destino los valores resultado de la tabla.
- UPT, Update Table: Esta instrucción actualiza los valores de salida de la última entrada consultada de la tabla con N registros de resultados.
- CLRT, Clear Table: Esta instrucción inicializa la tabla poniendo a 0 los bits de validez de todas las entradas.

Como se ve, en esta primera aproximación, las instrucciones definidas no han sido concretadas con unos valores de N y M fijos. En general estos valores dependerán de la arquitectura y pueden ser más o menos altos en cada caso concreto. En los experimentos realizados hemos supuesto que nos encontrábamos antes una arquitectura RISC típica en la que una instrucción puede definir dos registros de entrada y uno de salida. Hemos supuesto, además, que la instrucción puede incorporar un inmediato, de al menos 2 bits. En caso de que esta suposición fuese demasiado aventurada se podrían crear 4 instrucciones LACT distintas, cada una con un salto predefinido de 2, 3, 4 o 5 instrucciones o, incluso, utilizar códigos Nop para rellenar las posiciones vacías y saltar 5 instrucciones. Así pues, en estas condiciones, el valor de N sería 2 y el de M, 1.

Para entender de forma más exacta el funcionamiento de este sistema imaginemos un código a reusar. En alto nivel podríamos definir unas directivas que serían:

```
Acceder_Tabla(RE1,..,REx,RS1,..,RSy, Fin)
# Código a reusar con valores de entrada RE1 a REx y de salida RS1 a RSy
Actualizar_Tabla(RS1,..,RSy);
Fin:
```

La directiva Acceder_Tabla indicaría un acceso con los registros de entrada RE1 a REx que, en caso de acierto, actualizaría los valores de los registros de salida RS1 a RSy y saltaría a la etiqueta Fin. En caso de fallo el código se continuaría ejecutando de forma normal. En alto nivel, lógicamente, sería posible especificar variables en lugar de registros, aquí hemos preferido indicar directamente registros por claridad.

Dependiendo de los valores de x e y (que pueden variar de 1 a 4 según la región) tendremos la traducción indicada en la tabla 5.3. La tabla no presenta todas las posibles combinaciones, tan solo las más extremas, pero los casos intermedios se implementarían con la interpolación obvia. Como se puede ver, el número de instrucciones necesarias para implementar el reuso varia en función del número de parámetros, siendo de entre 4 y 9. No todas se ejecutan cada vez, sino que en caso de acierto ejecutaremos entre 3 y 7 instrucciones (así pues, no podremos reusar regiones de menos de 3 instrucciones, siempre perderíamos) y en caso de fallo ejecutaríamos entre 2 y 4 instrucciones extras. Estos valores pueden variar para otros tipos de procesadores, pero los valores presentados aquí se ha intentado que sean representativos y razonables para un procesador RISC estándar.

Es posible, de hecho, mejorar los números en caso de poder disponer de más códigos de operación, sobretodo en el caso $y = 1$, ya que se puede crear una nueva instrucción LAWT que sea capaz de leer 2 registros de entrada y escribir 1 registro de salida en caso de acierto de la tabla (y no hacerlo en caso de fallo). En este caso ahorraríamos la instrucción WRT. Si, además dispusiésemos de otra instrucción capaz de escribir, o no, un registro y saltar de 1 a 3 instrucciones, podríamos ahorrar también un WRT para cualquier valor de y. Este ahorro es útil, pero tan solo si el acceso es un acierto ya que las instrucciones WRT solo se ejecutan en dicho caso. En nuestros experimentos, de todas formas, hemos preferido ser conservadores y no lo hemos tenido en cuenta.

Otro detalle de las instrucciones propuestas es que utilizan el registro R0 como un registro que permanentemente vale 0. Es razonable asumir que un procesador RISC disponga de un registro de este tipo. Gracias al uso de dicho registro no es necesario tener en cuenta mediante hardware la cantidad de registros de entrada o de salida activos, ya que el código se encarga de realizar las operaciones correctas.

Finalmente, tenemos la última instrucción: CLRT. Esta instrucción debe realizarse una vez antes de entrar en una región de reuso nueva, de forma que se eviten errores por reusar para una región resultados de otra. En nuestros experimentos hemos visto que con una única tabla y un solo bit de validez suele ser suficiente, ya que las regiones están claramente diferenciadas entre ellas. En programas en los que esto no fuese así podríamos, o bien usar tablas distintas (como se hace en [CmWH99]) o bien incorporar un contador de regiones en cada entrada. Dicho contador bastaría con que dispusiese de 3 bits (lo que daría lugar a 8 regiones distintas alternadas en el tiempo) y un acierto implicaría coincidencia de los registros de entrada y coincidencia del número de región accedida. En este caso la instrucción de inicialización debería ser capaz de inicializar un número de región concreta. Esta última

x	y	Traducción del acceso	Traducción de la actualización	Instrucciones acierto	Instrucciones extras fallo

1	1	`LACT RE1,R0,2`
		`WRT RS1`
		`JMP Fin`
			`UPT RS1,R0`	3	2

2	1	`LACT RE1,RE2,2`
		`WRT RS1`
		`JMP Fin`
			`UPT RS1,R0`	3	2

3	1	`LDT RE1,RE2`
		`LACT RE3,R0,2`
		`WRT RS1`
		`JMP Fin`
			`UPT RS1,R0`	4	3

4	1	`LDT RE1,RE2`
		`LACT RE3,RE4,2`
		`WRT RS1`
		`JMP Fin`
			`UPT RS1,R0`	4	3

4	2	`LDT RE1,RE2`
		`LACT RE3,RE4,3`
		`WRT RS1`
		`WRT RS2`
		`JMP Fin`
			`UPT RS1,RS2`	5	3

4	3	`LDT RE1,RE2`
		`LACT RE3,RE4,4`
		`WRT RS1`
		`WRT RS2`
		`WRT RS3`
		`JMP Fin`
			`UPT RS1,RS2`
			`UPT RS3,R0`
				6	4

4	4	`LDT RE1,RE2`
		`LACT RE3,RE4,5`
		`WRT RS1`
		`WRT RS2`
		`WRT RS3`
		`WRT RS4`
		`JMP Fin`
			`UPT RS1,RS2`
			`UPT RS3,RS4`
				7	4

Tabla 5.3: Instrucciones ensamblador para reuso de regiones.

Programa	Región	Instrucciones	Peso
cjpeg	Conversor de color	41	22%
	DCT, Filas	48	14%
	DCT, Columnas	55	10%
	Cuantificación	15	19%
tmn	Estimador de movimiento	12	67%
tmndec	Interpolador vertical	13	19%
	Interpolador horizontal	13	37%
	Conversor de gama	48	15%
toast	Filtro corto	86	75%

Tabla 5.4: Regiones seleccionadas para reuso de regiones.

aproximación es la que hemos usado en nuestros experimentos cuando ha sido necesario.

Las instrucciones del nuevo ISA descritas han sido evaluadas mediante su implementación dentro del simulador SimpleReuse. Para ello se han utilizado códigos de operación no utilizados con anterioridad, se les ha añadido la nueva funcionalidad y, asimismo, se ha implementado la simulación de su consumo de energía y tiempo. En este proceso se ha tenido en cuenta, además, el coste de la tabla de reuso simulada, tanto su consumo por cada acceso como el derivado de las perdidas de corriente (que se da aunque la unidad esté apagada). El siguiente paso necesario ha sido incluir en los programas de alto nivel las directivas necesarias para que el compilador utilizara las nuevas instrucciones. No se han modificado los programas originales en ningún otro sentido.

5.5.1 Resultados del reuso de regiones.

Una vez diseñados el ISA y la tabla de reuso se buscaron regiones susceptibles de ser reusadas. La tabla 5.4 muestra, para cada programa estudiado, y por orden de ejecución, las diferentes regiones que pueden ser reusadas así como su longitud en instrucciones y su peso en porcentaje en la ejecución del programa (calculado mediante profiling en un Alpha 21164).

Tal y como se puede apreciar en la tabla5.4, todos los programas de prueba estudiados tienen regiones susceptibles de ser reusadas que ocupan, en conjunto, más del 50% del tiempo de cálculo del programa. Estas regiones, además, cumplen los criterios necesarios para poder usar el hardware propuesto anteriormente, es decir, dependen de 4 o menos valores de entrada, dan lugar a 4 o menos valores de salida y dichos valores se pueden

almacenar en 16 bits.

En los programas analizados existían, además, otras regiones de cálculo (ver figura 5.1) que no han podido ser reusadas. Esto se debe, principalmente a dos tipologías distintas:

- Hay etapas que realizan movimientos de datos, no transformaciones. Un ejemplo típico de estas etapas del procesado es la emisión de bits hacia la etapa posterior (un fichero o un visualizador, por ejemplo). Este tipo de algoritmos, al no realizar cálculos, no son susceptibles de ser reusados.

- Hay etapas que realizan cálculos demasiado baratos de computar. En realidad estas etapas son casi el mismo ejemplo que el caso anterior, pero con un pequeño cálculo incluido. Aunque estas regiones puedan ser reusadas, la sobrecarga introducida debido a nuestro sistema hace que el reuso no sea eficiente.

Como se puede ver, todas las regiones escogidas superan las 11 instrucciones ensamblador. Se puede analizar fácilmente, a partir de los datos de la tabla 5.3 que para el caso extremo de una región de 12 instrucciones y 4 datos de entrada y uno de salida, tendremos, en caso de acierto, que ejecutar 4 instrucciones (y por tanto ganaremos 8) mientras que en caso de fallo, ejecutaremos 15 instrucciones (perdiendo, por tanto 3 instrucciones al implementar el reuso). Así pues, a groso modo (y suponiendo que todas las instrucciones tienen el mismo coste de ejecución) necesitaremos tasas de acierto superiores al 30% para conseguir resultados positivos. Regiones más cortas hacen inviable el reuso.

Finalmente se han descartado un último tipo de regiones, susceptibles de ser reusadas y con una longitud aceptable debido a su bajo peso en el coste computacional de la aplicación. Como ejemplo, tanto los programas tmn como tmndec (codificador y decodificador de vídeo, respectivamente) contienen regiones que realizan el cálculo de la dct (reusada con éxito en el programa cjpeg) que no se han incluido en este estudio debido a la baja incidencia que tenían en el tiempo de ejecución del programa ($< 8\%$).

Una vez se han decidido las regiones susceptibles de ser reusadas se realizó un estudio para ver el potencial de reuso de dichas regiones, es decir, ¿son los datos de entrada lo bastante repetidos como para obtener buenos porcentajes de reuso? Para ello se realizó un estudio con tablas infinitas para todos los programas y entradas. Los resultados para el programa cjpeg se muestran en la tabla 5.5. Estos resultados son los más relevantes de todos

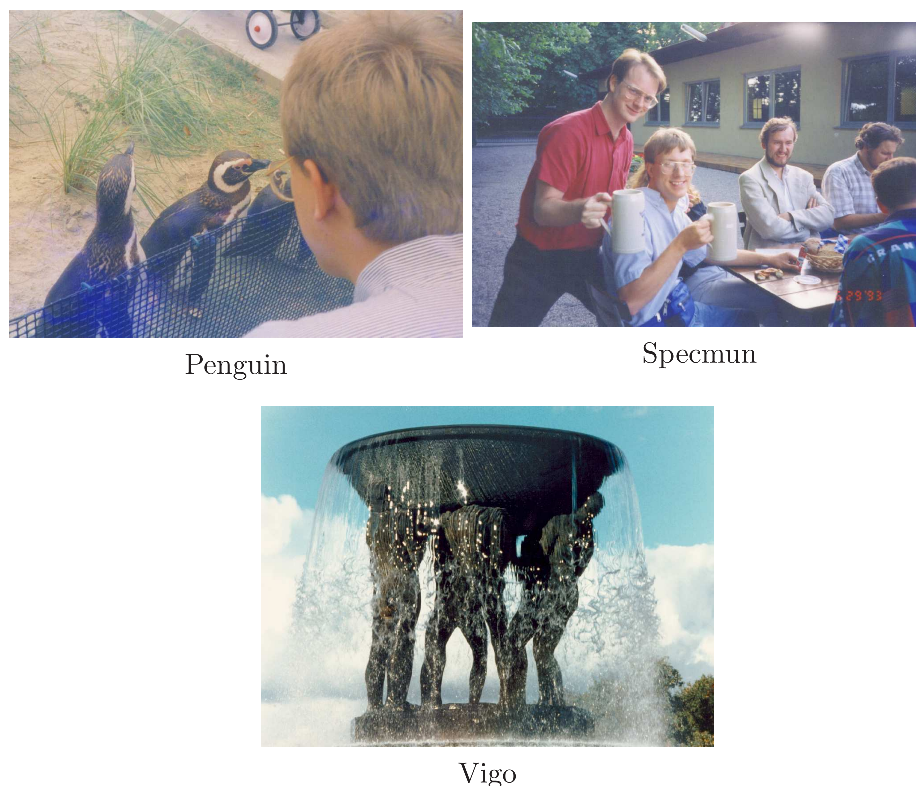
	Instrucciones	% Aciertos	Ins. con Reuso	% reducción
	penguin			
CColor	39371703	60.3	20649832	47.6
DCTrow	21907200	28.8	17944122	18.1
DCTcol	21907200	0.7	21797163	0.5
Q	34884383	98.0	17430613	50.0
Resto	63790808	0	63790808	0.0
Total	181861294	–	138919474	22.1
	specmun			
CColor	36654576	57.2	20124852	45.1
DCTrow	20185920	30.2	16363634	18.9
DCTcol	20185920	0.4	20134218	0.3
Q	32073913	97.9	16114210	49.8
Resto	50972305	0	50972305	0.0
Total	160072634	–	123709219	22.7
	vigo			
CColor	40916736	94.2	10557917	74.2
DCTrow	22533120	27.6	18620814	17.4
DCTcol	22533120	0.8	22408349	0.6
Q	35930219	97.2	18164836	49.4
Resto	61465508	0	61465508	0.0
Total	183378703	–	131217424	28.4

Tabla 5.5: Reuso potencial con tablas infinitas en las diferentes etapas de la aplicación JPEG.

los programas ya que nos muestran un caso especial que es el de la transformada DCT por columnas que tan solo es capaz de reusar menos de un 1% de las entradas (frente al resto de algoritmos, que, con tablas infinitas, alcanzan índices de aciertos de más del 90% en algunos casos). ¿A que se debe esta diferencia entre algoritmos? La explicación la hemos encontrado en la propia naturaleza del algoritmo de compresión. Por definición estos algoritmos lo que hacen es reducir la entropia de las señales tratadas, de forma que tengamos menos cantidad de señal, pero con más cantidad de información por elemento. La transformada DCT es la principal encargada de este proceso en el compresor JPEG, pero, al ser una transformada bidimensional se realiza en dos etapas, la horizontal (o por filas) y la vertical (o por columnas). De esta forma, la entrada a la etapa DCTcol es, directamente, la salida de la etapa DCTrow y, por tanto, dicha señal de entrada tiene una alta tasa de desorden y, consecuentemente una muy baja tasa de acierto.

Se podría argumentar, entonces que ¿como es posible que la cuantificación, que es la etapa posterior, tenga una tasa de acierto tan alta? Se debe a que para obtener la tasa de acierto debemos tener en cuenta que la cantidad de valores de entrada es distinta en cada etapa (concretamente, 2 valores en la cuantificación y 4 en la DCT) y, además, la cuantificación obtiene su señal, por un lado, de la salida de la DCT y, por otro de una matriz de coeficientes que no está tan desordenada.

Otro dato interesante que se puede observar en la tabla5.5 es que algunas



Penguin Specmun

Vigo

Figura 5.3: Las tres imágenes procesadas mediante JPEG.

imágenes muestran un porcentaje de aciertos significativamente más alto que otras en algunos algoritmos (concretamente la imagen “Vigo” en la conversión de color). Esto se debe a que, tal y como se ve en la figura5.3 la imagen “Vigo” presenta una mayor homogeneidad de colores. Por este motivo, entradas de datos demasiado homogéneas han sido descartadas de las pruebas ya que sus resultados son demasiado buenos[2].

Así pues, los resultados de la figura5.4 muestran la cantidad de aciertos obtenidos para tablas reales de todos los algoritmos estudiados menos la transformada DCT por columnas que no ha podido ser reusada. Cada gráfica muestra los resultados de cada uno de los algoritmos (más una última gráfica con la media) en función del tamaño de las tablas y de la asociatividad. Los resultados se encuentran agrupados por asociatividad para mostrar más claramente la evolución de la cantidad de aciertos en función del tamaño de la tabla. Como se puede ver en la gráfica se han estudiado 4 tamaños posibles de gráficas: de 256, 1024, 4096 y 16384 entradas, con un tamaño total respectivamente de 4KB, 16KB, 64KB y 256KB. Esta última tabla es claramente desproporcionada, sobretodo teniendo en cuenta que hay numerosos

[2] Lógicamente, no tiene gran mérito reusar el 99% de las muestras de una imagen negra.

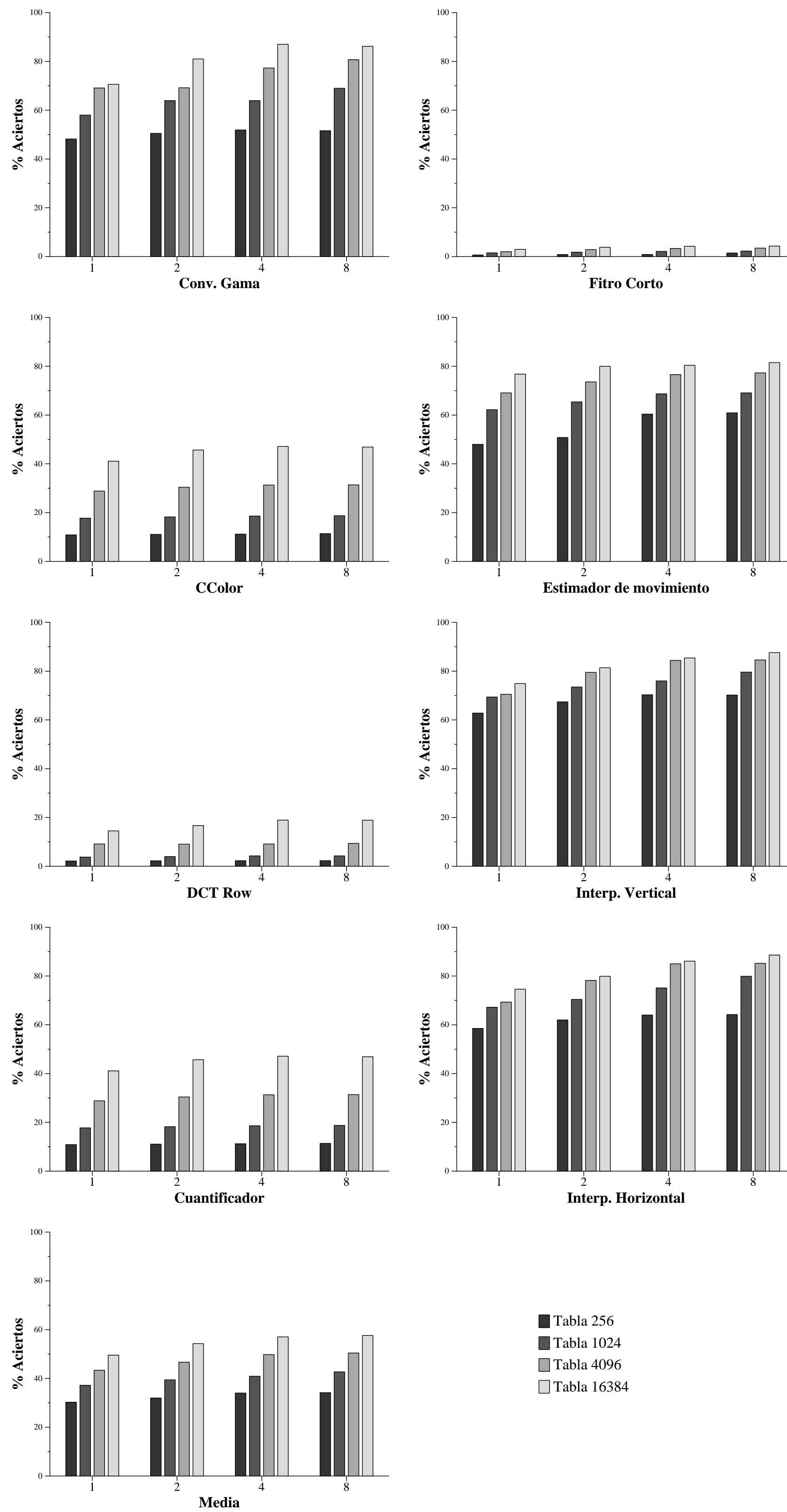


Figura 5.4: Porcentaje de aciertos de los algoritmos estudiados con tablas finitas.

procesadores (y no solo de bajo consumo) con menos memoria caché, pero dada la evolución de los procesadores de altas prestaciones, con cada vez más transistores por emplear de forma eficiente, nos ha parecido interesante desde un punto de vista teórico. La asociatividad empleada para cada tabla es 1, 2, 4, y 8, de forma que podemos ver la incidencia de esta en la cantidad de aciertos.

Los resultados de la figura 5.4 muestran que, en general, aumentar el tamaño de la tabla es bueno, ya que el porcentaje de aciertos se incrementa de forma significativa. Sin embargo, la asociatividad no parece influir mucho en los resultados, exceptuando algunos algoritmos concretos, todos pertenecientes a la aplicación de decodificación de vídeo. De todas formas, si se comparan los resultados de los tres primeros algoritmos de la figura 5.4 con los de la tabla 5.5, se puede ver que aún los resultados de la tabla mayor, con la mayor asociatividad, se quedan bastante alejados de los máximos teóricos obtenidos con la tabla infinita. Esto es debido a que la cantidad de muestras es demasiado alta y, en ocasiones, dos zonas similares se encuentran demasiado alejadas entre si en el procesado. En el caso del algoritmo JPEG, por ejemplo, la imagen se procesa dividida en bloques de 8×8 píxeles que, a su vez, se procesan por filas, de forma que, irónicamente, el punto de coordenadas [8,0] se encuentra muy alejado en el procesado del punto [7,0], cuando en la imagen son adyacentes[3].

Finalmente las figuras 5.5, 5.6 y 5.7 muestran los resultados de ahorro en tiempo, energía y producto tiempo×energía obtenidos en los distintos procesadores de prueba (ancho 1 y 2 en orden y ancho 4 fuera de orden). Para procesador se muestran los ahorros obtenidos en cada aplicación por separado y la media usando tablas de diferentes longitudes: 256, 1024, 4096 y 16384 entradas. La asociatividad escogida para cada tabla ha sido de 1, 2, 4 y 4, respectivamente, a raíz de los resultados de la figura 5.4. Esta asociatividad se emplea en las tablas de aquí en adelante en todos los experimentos restantes.

De estos resultados, lo que más llama la atención es que los mayores ahorros obtenidos se dan en el procesador de altas prestaciones. Al igual que sucedía con las instrucciones de punto flotante, esto se debe a la ley de Amdahl: conforme ejecutamos más rápido una parte del programa, el resto adquiere más importancia. Así pues, las instrucciones de fuera del núcleo del programa se ejecutan mucho más rápidamente en el procesador de altas prestaciones, de forma que la velocidad que podemos ganar en el núcleo se hace mucho más significativa.

Otro resultado que llama la atención es que el mejor resultado no se

[3]Y, muy probablemente, azules.

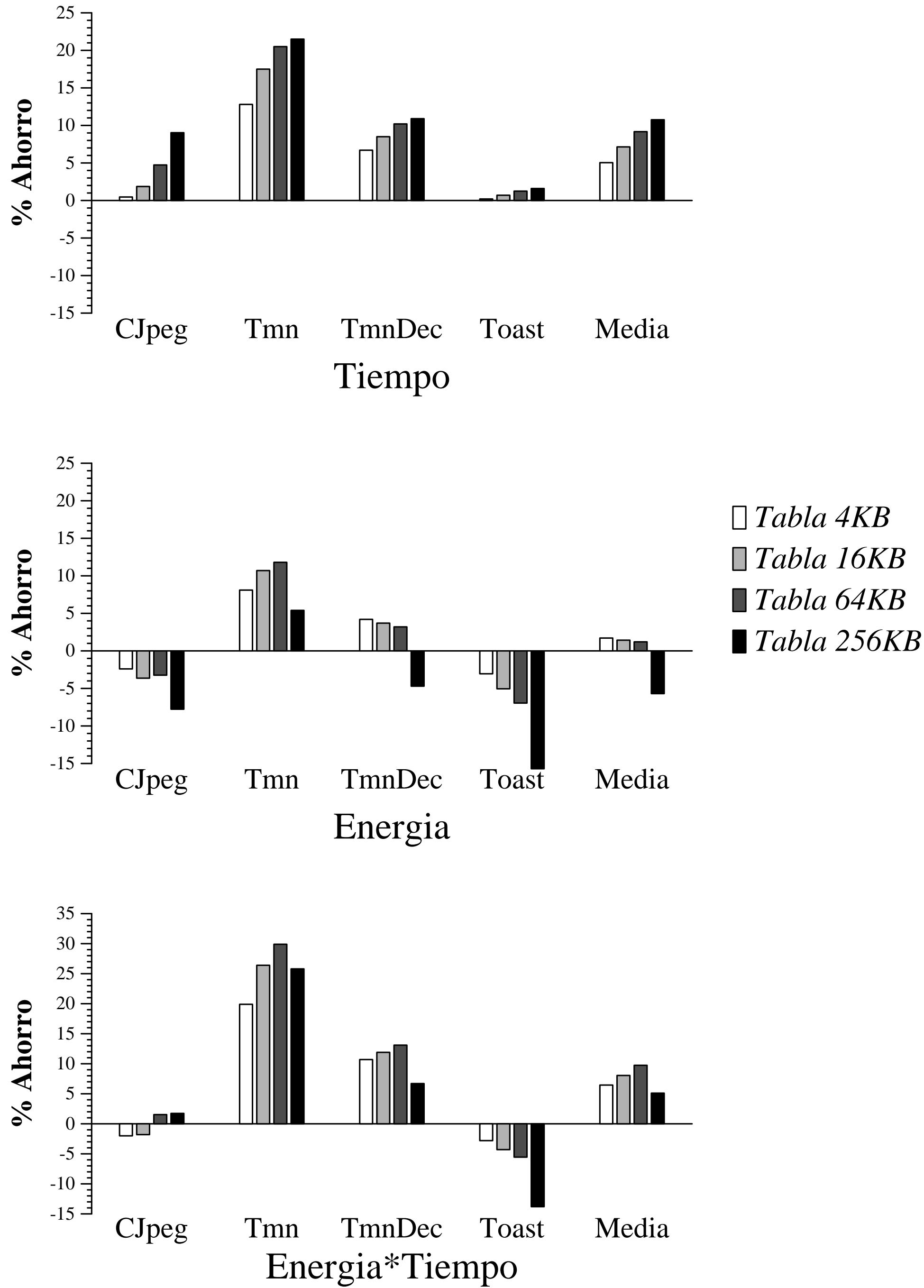


Figura 5.5: Resultados del reuso de regiones en un procesador de ancho 1.

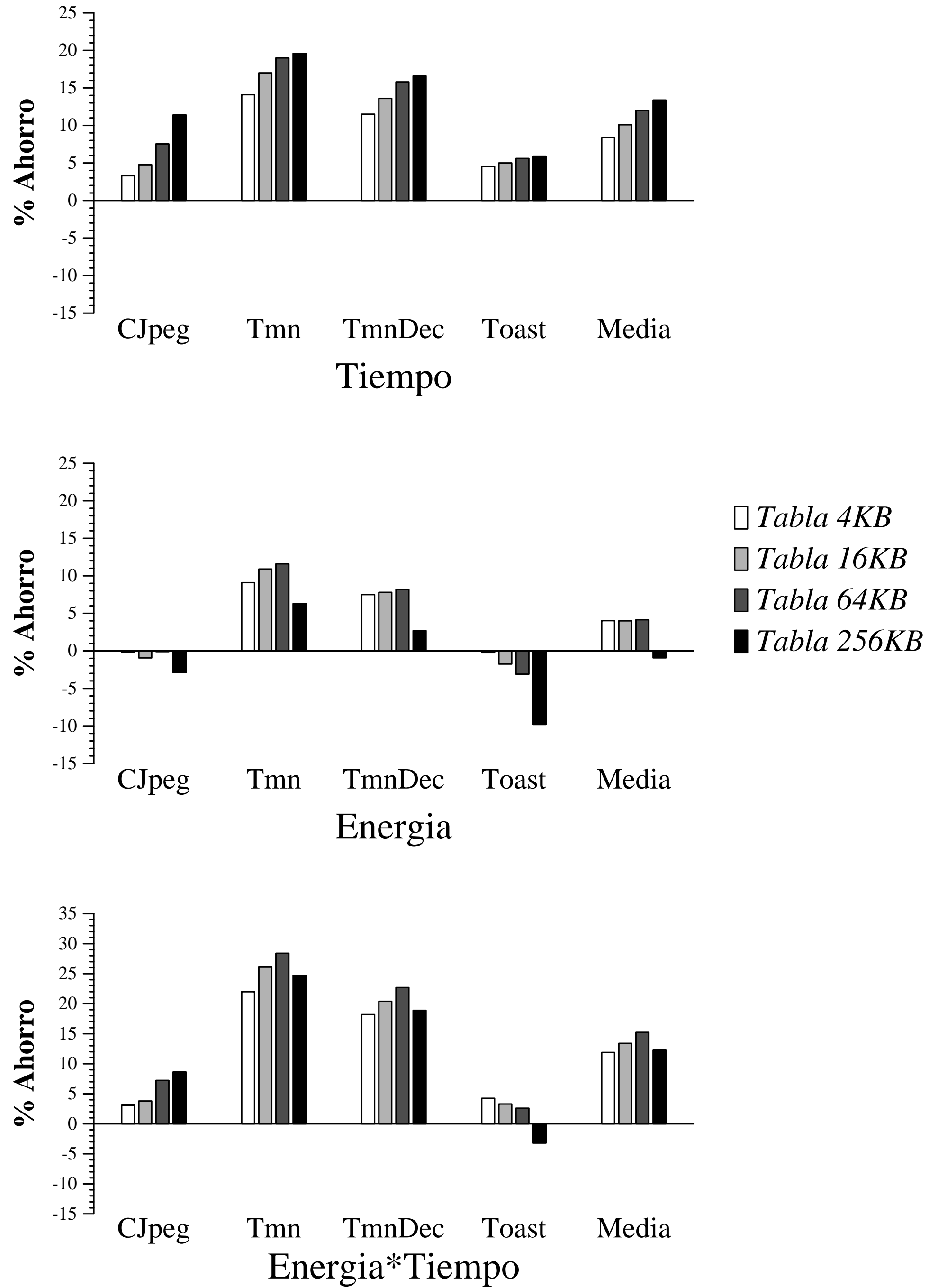


Figura 5.6: Resultados del reuso de regiones en un procesador de ancho 2.

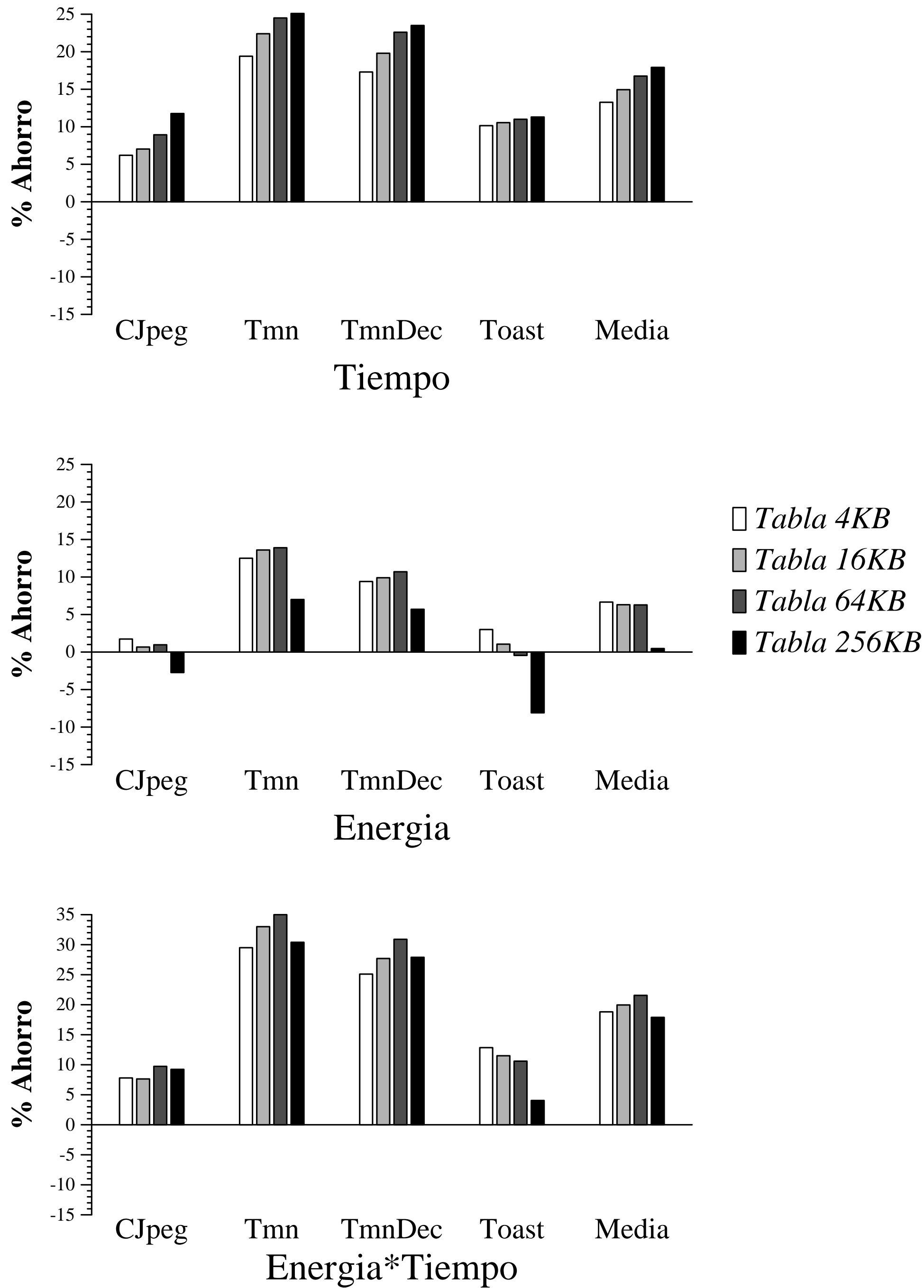


Figura 5.7: Resultados del reuso de regiones en un procesador fuera de orden de ancho 4.

obtiene con la tabla mayor (que es la que obtenía mejores índices de aciertos) sino con la inmediatamente anterior (de 64 KB). Esto es así porque el aumento de aciertos en la tabla y, por tanto, de reusos, no es lo bastante significativo como para compensar el aumento de consumo. Desde un punto de vista de tiempo de ejecución, siempre nos convendría maximizar el tamaño de la tabla. Desde el punto de vista de la energía los mejores resultados son para las dos tablas más pequeñas y, desde el punto de vista del producto energía×tiempo, el mejor resultado se obtiene para la tabla de 4096 entradas (que con 64KB solo puede plantearse para procesadores de altas prestaciones).

Asimismo se puede ver claramente que los resultados varían mucho dependiendo de la aplicación. Para el procesador más simple (ancho 1), las ganancias obtenidas se deben prácticamente en su totalidad a un solo programa y algoritmo (el estimador de movimiento del programa codificador de vídeo tmn), mientras que el procesador más ancho consigue repartir mejor las ganancias (debido, de nuevo, a que es capaz de procesar más rápido la parte no reusada). Aún así los resultados están bastante desequilibrados.

5.5.2 Conclusiones sobre el reuso de regiones.

Se puede decir vistos los resultados del reuso de regiones del apartado anterior que esta es una técnica adecuada para el procesado multimedia según el tipo de procesador en que estemos pensando. En procesadores de alto rendimiento esta técnica proporciona unas ganancias aceptables que pueden justificar su incorporación (sobretodo desde el punto de vista del tiempo), pero, habría que compararla atentamente con alguna otra técnica, por ejemplo de vectorización a nivel de subpalabra, antes de recomendar su inclusión en los procesadores de nueva generación.

En los procesadores de bajo consumo los resultados son incluso peores. En tres de las cuatro aplicaciones probadas los resultados son nulos o incluso negativos desde el punto de vista de la energía, y solo pequeños ahorros en el tiempo de ejecución podrían justificar su uso. Para la aplicación restante, todas las ganancias se consiguen en un algoritmo concreto, de forma que cualquier otro hardware específico podría, seguramente, obtener unos resultados similares a un coste de implementación, probablemente, menor.

5.6 El reuso tolerante de regiones.

Vistos los resultados mostrados en el apartado anterior, nuestra propuesta es aplicar el paradigma del cálculo difuso al reuso de regiones mediante el reuso tolerante. Esta aproximación debería permitir a los sistemas de reuso para aplicaciones multimedia alcanzar un porcentaje de aciertos en las tablas suficiente como para compensar el hardware empleado.

5.6.1 Hardware para el reuso tolerante.

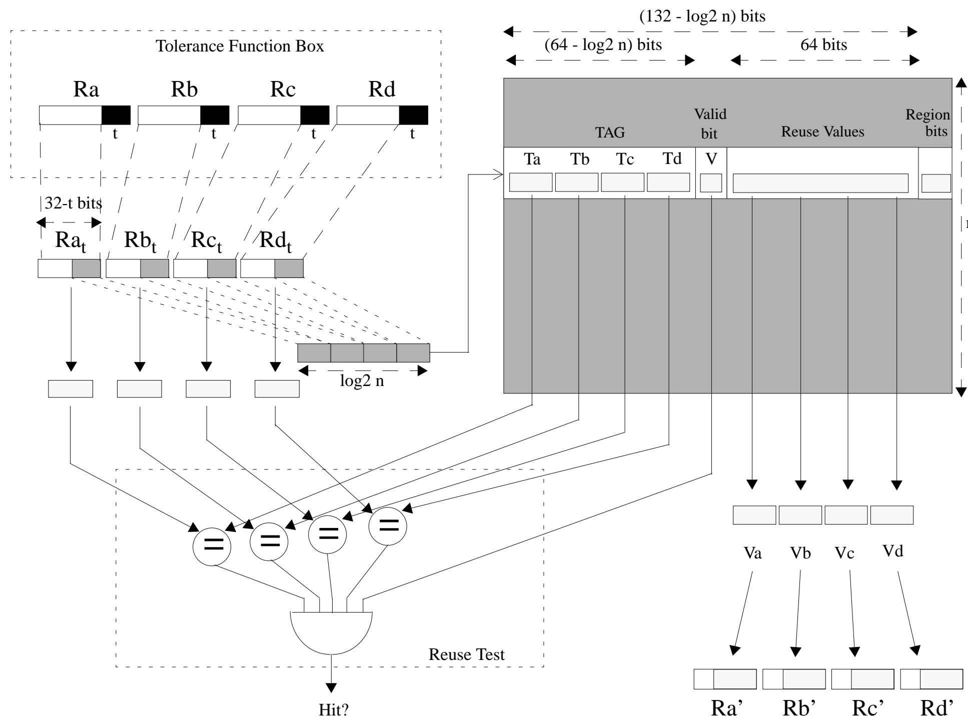


Figura 5.8: Tabla hardware para el reuso tolerante de regiones.

El sistema hardware para el reuso tolerante que proponemos es una modificación simple del sistema propuesto anteriormente para el reuso convencional. En la figura 5.8 podemos ver dicho esquema. Como se puede ver en la figura, la principal modificación es que a la hora de realizar el acceso a la tabla deberemos utilizar como valores de entrada los obtenidos a través de las instrucciones de reuso, pero eliminando de cada valor los N bits menos significativos. Estos N bits son el nivel de tolerancia y nos permitirán aumentar el nivel de aciertos en la tabla.

Salvada esta pequeña diferencia, el esquema funciona exactamente igual que en el reuso clásico, en caso de fallo se almacenan los valores de entrada y la posición accedida para que, una vez realizado el cálculo de forma normal, se pueda actualizar dicha posición con los valores de salida correspondientes. Si, en cambio, ha habido un acierto, los valores de salida almacenados se cargarán en los registros de destino, también mediante las instrucciones correspondientes. Evidentemente esta salida no será exactamente la misma que si hubiéramos realizado el cálculo (al menos en la mayoría de los casos), pero la hipótesis es que será lo suficientemente cercana.

Así pues, nuestra tabla de reuso tolerante incorpora sobre la tabla de reuso clásica un sistema comparador capaz de no comparar N bits de los operandos y un registro de unos 3 bits capaz de almacenar el valor N[4].

5.6.2 Modificaciones en el ISA.

Las modificaciones necesarias al ISA presentado anteriormente son mínimas. Los requisitos para acceder a este nuevo hardware son practicamente idénticos. Un solo punto nuevo es necesario, poder indicar al sistema de reuso la cantidad de bits que debe tolerar en cada entrada.

Para ello, existen dos aproximaciones posibles:

- Modificar la instrucción LACT: habría que añadir a esta instrucción un nuevo parámetro, un inmediato. No es necesario que este inmediato pueda llegar a ser muy alto, con tres bits (es decir, valores de 0 a 7) sería más que suficiente. De esta forma, cada instrucción de acceso a la tabla indicaría, además, cuantos bits habría que tolerar en dicho acceso. El problema de esta aproximación es que la instrucción LACT ya está de por sí bastante cargada de funcionalidades, así que esta aproximación no es muy adecuada.

- Modificar la instrucción CLRT: esta instrucción está desaprovechada, ya que tan solo indica que la tabla se debe inicializar. Es muy fácil añadirle un inmediato (nuevamente, con tres bits sería suficiente) que indique cuantos bits se deben tolerar en los accesos posteriores a la tabla.

[4]En realidad varios registros, uno por cada región distinta en un mismo programa. Un cantidad razonable podría ser de 3 o 7 registros de 3 bits. En todo caso, la cantidad debe coincidir con la cantidad de regiones diferentes que se pueden almacenar en una entrada de la tabla.

En este trabajo nos hemos decidido por la segunda aproximación, más realista y fácil de llevar a la práctica. Así pues, nuestro sistema tan solo permite tolerar una cantidad fija de bits por región (y no por acceso a la tabla).

5.6.3 La tolerancia, los aciertos y el error.

Al igual que sucedía en los sistemas de cálculo difuso de instrucciones, un problema importante del hecho de aplicar tolerancia al reuso de regiones es el error introducido. Para poder medir dicho error hemos aplicado diferentes niveles de tolerancia a cada una de las regiones reusadas y hemos comparado las salidas obtenidas con la salida original (que en nuestro caso es la ideal). Las salidas se han clasificado, como siempre, de dos formas, una por SNR y otra, por calidades subjetivas, aunque se ha de decir que para los programas evaluados en este caso, ambas medidas coinciden prácticamente de forma total (no tenemos el efecto de desplazamiento en frecuencia que presentaba el algoritmo de compresión MP3 y que distorsionaba la medida de la SNR).

Eso si, en este sistema, el ajuste de las variables tolerancia vs. error es ligeramente más complejo que el visto en el capítulo anterior, principalmente debido a los programas que contienen más de una región reusada. Así pues, deberemos ajustar las tolerancias de cada región por separado, pero el error introducido se acumulará de región en región dando lugar a comportamientos más difíciles de analizar.

Nuestra primera medida fue comprobar el incremento de aciertos que se podría obtener con los diferentes niveles de tolerancia. La figura 5.9 muestra estos resultados para cada una de las distintas regiones del algoritmo *cjpeg* y para cada uno de los programas de pruebas. Para cada región se muestran los resultados de porcentaje de aciertos en tablas de 256, 1024 y 4096 entradas y para una tabla infinita con valores de tolerancia (N) 0, 1, 2 y 3.

Los resultados de la figura 5.9 son interesantes por diversos motivos. El primero, y sin duda el más importante, es que se puede ver que el porcentaje de aciertos se incrementa significativamente conforme aumenta la tolerancia, independientemente del tamaño de la tabla y del algoritmo. Así pues, para la tabla de 256 entradas, y el algoritmo de la transformada DCT por filas, por ejemplo, los aciertos pasan de estar en torno al 2% a rondar el 60%. Para el caso de la DCT por columnas, los aciertos pasan de prácticamente el 0% a un 40% si pudiesemos disponer de una tabla infinita (como esto no es así, este algoritmo sigue sin merecer la pena implementarlo con tablas reales). Este incremento tan solo es menos importante en el caso de tablas infinitas

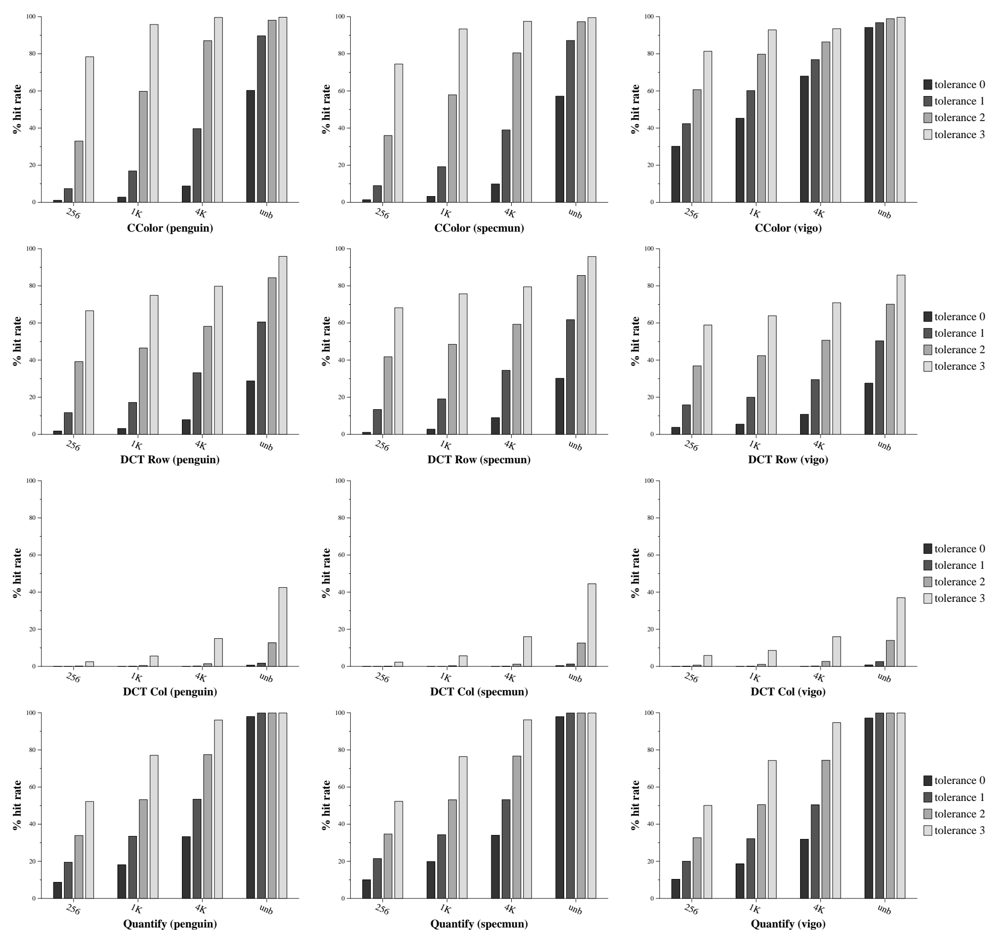


Figura 5.9: Porcentaje de aciertos en las tablas cuando se incrementa el grado de tolerancia en el programa *cjpeg*.

con los algoritmos que alcanzan más de un 90% de aciertos sin tolerancia ya que no hay margen para mejorar.

El segundo resultado importante, desde nuestro punto de vista, de la figura 5.9 es el que obtenemos al observar el comportamiento de las tres entradas distintas en el algoritmo de conversión de color conforme aumenta la tolerancia. Ya habíamos comentado que el porcentaje de aciertos variaba mucho según la imagen ya que *vigo*, por ejemplo, presentaba grandes zonas de un mismo color (y por lo tanto los aciertos en estas zonas crecían de forma significativa). Este efecto era lo bastante importante como para, incluso, distorsionar al alza la media de resultados. Sin embargo, conforme la tolerancia aumenta, este efecto disminuye. Es decir, la imagen *vigo* sigue presentando un numero de aciertos mayor en el algoritmo de conversión de color para tolerancia 3, que la imagen *penguin*, pero la diferencia ha

disminuido desde un 30% (para la tabla de 256K y tolerancia 0) a menos de un 5% (para tolerancia 3). Es decir, el sistema de tolerancias iguala las características de las imágenes.

Las gráficas de la figura 5.10 muestran los resultados de aciertos para todas las regiones estudiadas y 4 tamaños de tablas diferentes (la última de las cuales es tan grande, 16 K entradas, que casi se puede considerar infinita). En estas gráficas se muestra directamente la media de aciertos cuando hay más de una entrada distinta por programa. Como se puede ver los resultados mostrados por imagen en la figura 5.9 se repiten aquí en todos los casos y programas: el aumento de la tolerancia implica un aumento importante en el porcentaje de aciertos; además, en los casos en los que el porcentaje de aciertos inicial es bajo, el crecimiento es mayor en proporción, superando en la mayoría de los casos el efecto de incorporar tablas mucho más grandes. Tanto es así, que el efecto de introducir un nivel de tolerancia de 4 es claramente superior a aumentar el tamaño de la tabla en 64 veces en 6 de las 8 regiones.

Dado que la tolerancia da buenos resultados, nuestro siguiente experimento trató de relacionar para cada región reusable, el error introducido al tolerar las entradas. Para ello medimos las SNR entre las salidas obtenidas al variar la tolerancia en esa región y las salidas originales. Los resultados de estas medidas pueden verse en la figura 5.11 que muestra para cada región, las SNR obtenidas al variar la tolerancia según el tamaño de la tabla de reuso utilizada.

A partir de los datos mostrados en la figura 5.11 se puede ver que casi todas las regiones soportan tolerancias de uno o dos bits, y que algunas de ellas incluso soportan cuatro bits de tolerancia obteniendo unas SNR bastante buenas (JPEG 75, considerado calidad buena, genera unas SNR de alrededor de 25dB; JPEG 50, calidad regular, de alrededor de 20dB). Estos resultados, así en conjunto, son buenos, pero requieren un análisis más detallado, programa por programa.

Lo primero de todo es comentar el caso de la octava gráfica de la figura 5.11. Esta gráfica, a diferencia de las demás no muestra SNR obtenidas según la tolerancia, sino incrementos de tamaño. ¿A que se debe esta diferencia? Pues a la idiosincrasia particular del algoritmo estudiado en este caso: el estimador de movimiento. Este algoritmo es el único algoritmo reusado del programa compresor del estándar de codificación de vídeo H263. Este estándar, al igual que la gran mayoría de estándares actuales (MPEG2, H264), se basa en la codificación de unas imágenes a partir de las anteriores (o de las siguientes). La idea básica detrás de este comportamiento es que en un vídeo, si no hay cambios de plano, una imagen será muy similar a la anterior excepto por pequeñas variaciones (un paso hacia adelante, un giro

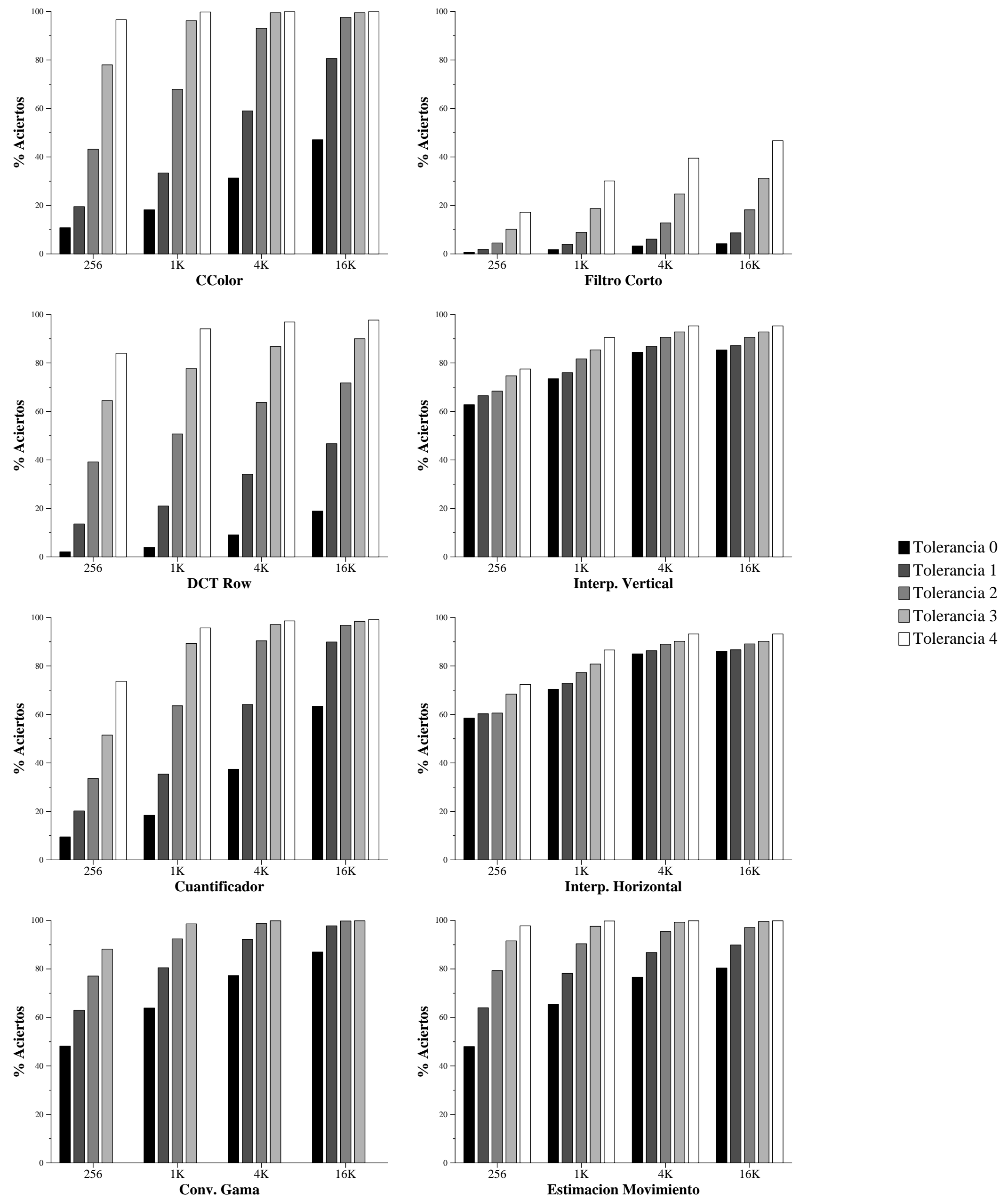


Figura 5.10: Porcentaje de aciertos en cada región con diferentes tolerancias.

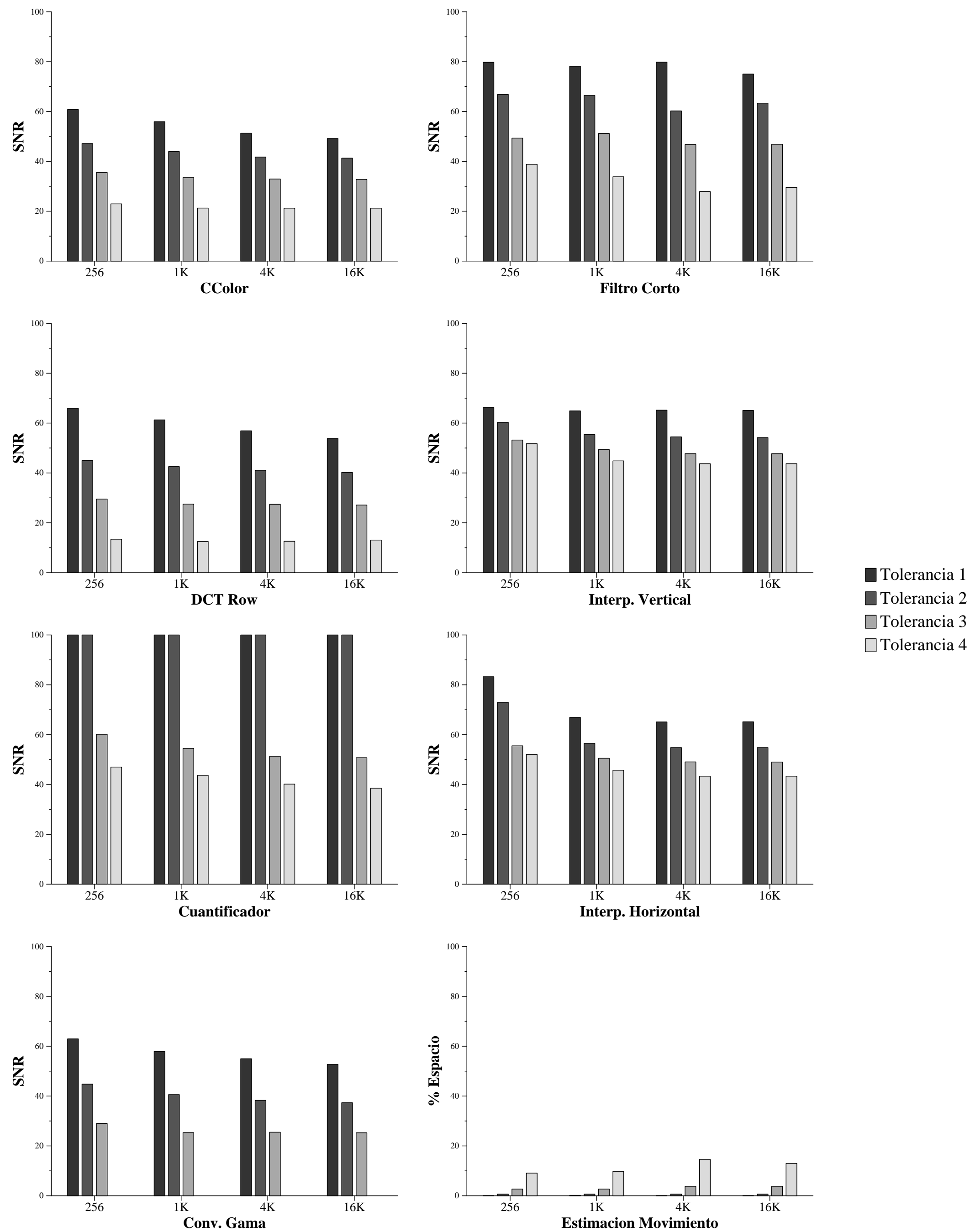


Figura 5.11: SNR según la tolerancia para las diferentes regiones estudiadas.

en la cabeza, un cambio de expresión). Así pues, el trabajo principal de estos algoritmos es buscar, para cada sector de la imagen (por ejemplo, para cada grupo de 8×8 píxeles) donde se encuentra en la escena siguiente. Esta busqueda es la estimación de movimiento.

Lo que tiene de particular la estimación de movimiento es que, a pesar de ser la fase más costosa de la codificación, no codifica en si misma, tan solo nos dice hacia donde parece haberse movido un bloque de la imagen. Como la codificación se realiza a continuación (usualmente mediante una resta u otros algoritmos de bajo coste computacional), introducir tolerancia en el estimador de movimiento no empeora la calidad de la señal codificada, tan solo hace que esta sea un poco mayor (es decir, comprimimos ligeramente peor). Por esta razón, la medida de calidad de la señal es el incremento de tamaño del archivo comprimido que, como se puede ver, no es muy significativa para tolerancias pequeñas.

El segundo caso a analizar, es el programa codificador de voz *toast*, que aplica el estándar de codificación GSM. En este caso el análisis es simple: el programa contiene un solo algoritmo reusado, un filtro corto FIR, que pierde calidad a medida que aumenta la tolerancia. Este programa admite tolerancias de hasta 4 bits sin perdida de calidad audible (de hecho, con 5 bits, la voz es perfectamente inteligible, pero ya sufre, a tramos, algunas pequeñas distorsiones).

Los dos programas restantes (*cjpeg* y el decodificador de vídeo *tmn*) son más complejos de analizar, ya que ambos contienen más de un algoritmo con tolerancia (tres en cada caso). Así pues, el resultado final de los programas perderá calidad de forma conjunta por todos los algoritmos, pero no de forma lineal con estos. Hallar la tolerancia óptima en este caso es complicado, ya que nos interesa mejorar de forma conjunta en todas las regiones pero no empeorar demasiado la calidad en ninguna de ellas. Las gráficas de la figura 5.12 muestran las SNR obtenidas con diferentes valores de tolerancia para cada una de las regiones. Se ha escogido una región como principal, aquella que más variación de la SNR produce y en función de esta se han representado el resto. La tabla escogida para presentar la SNR ha sido la tercera de las estudiadas (4096 entradas, asociatividad 4) ya que es la mayor realmente implementable. De todas formas, el tamaño de la tabla tampoco altera de una forma significativa las SNR (como se puede ver en la figura 5.11). Las 5 primeras gráficas de la figura 5.12 muestran resultados del programa CJPEG y las 4 últimas del programa TMN. En el primer caso cada gráfica muestra una tolerancia del algoritmo de conversión de color, las columnas se agrupan por tolerancia del algoritmo DCT y cada columna es una tolerancia distinta del algoritmo de cuantificación. En el segundo caso, cada gráfica muestra una tolerancia distinta del algoritmo de conversión de

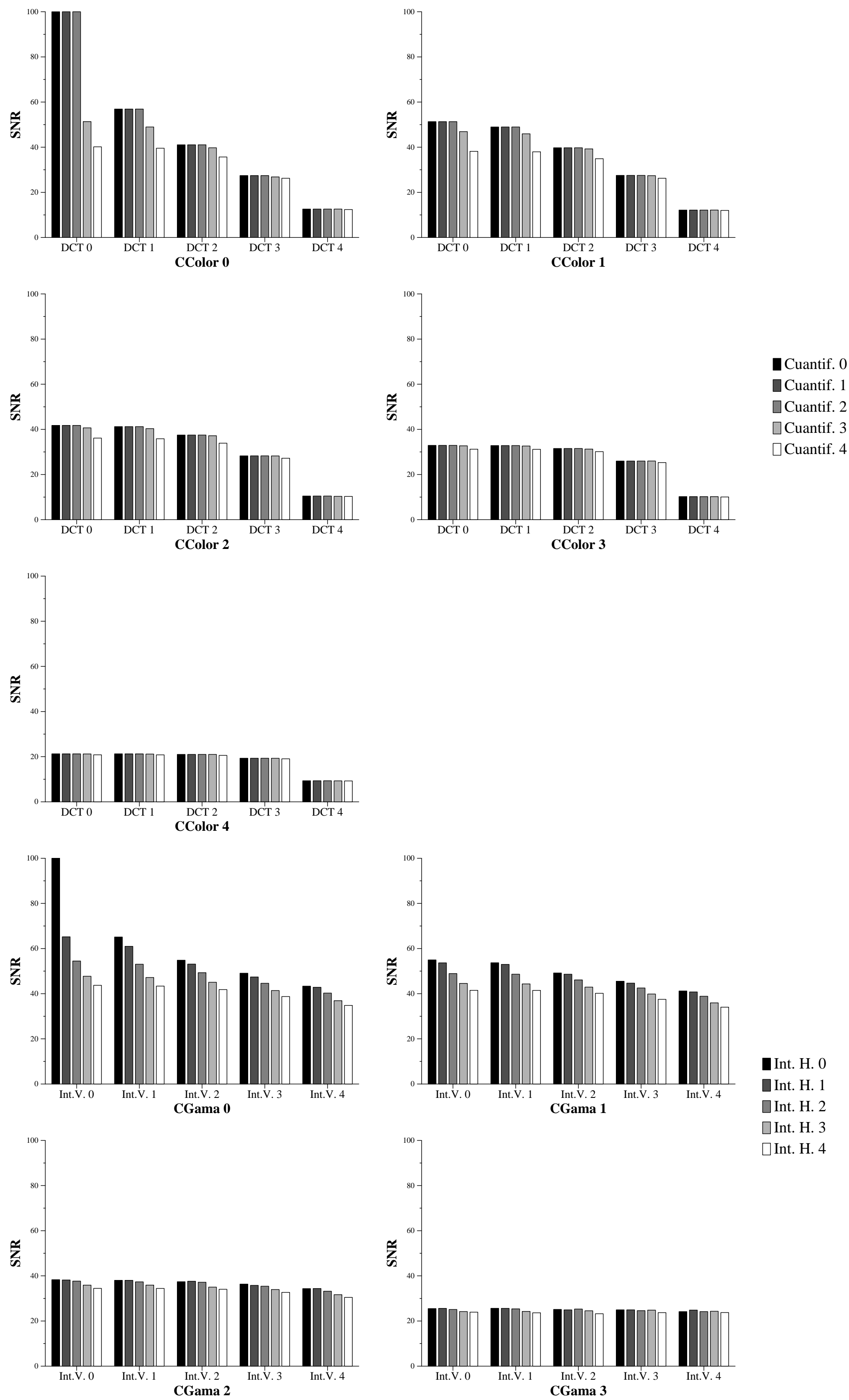


Figura 5.12: SNR con varias regiones toleradas de las aplicaciones JPEG y H263.

gama, las columnas se agrupan por el algoritmo de Interpolación Vertical y cada columna es una tolerancia distinta del algoritmo de Interpolación Horizontal.

Un efecto interesante (y conveniente) que se puede observar en la figura 5.12 es que la SNR perdida por tolerancias en un algoritmo no se suma a la de los siguientes, sino que hasta un cierto punto sirve para disminuir su influencia. Es decir, si no toleramos ningún bit en la conversión de color, por ejemplo, tolerar un bit en la DCT influye mucho en la SNR final. En cambio si toleramos 2 bits en la conversión de color, casi tenemos que tolerar 3 bits en la DCT para que esta influya. Este efecto provoca que, en las gráficas, parezca que el tercer algoritmo de cada programa (la cuantificación y la interpolación horizontal respectivamente), parezcan no tener influencia en el resultado final cuando esto no es cierto. Si invirtiéramos el orden en el que aplicamos las tolerancias, las gráficas presentarían prácticamente el mismo aspecto.

5.6.4 Resultados del reuso tolerante de regiones.

A partir de los resultados del apartado anterior se decidió estudiar los resultados de ganancia en tiempo y energía del reuso tolerante de regiones para los 4 programas. Los resultados de dichos programas, sin embargo, presentan dos vertientes distintas, debido a las características especiales del programa codificador de vídeo que ya se han comentado en el apartado anterior.

La figura 5.13 presenta los resultados de ganancia en tiempo, energía y tiempo×energía del programa codificador de vídeo *tmn* conforme aumentamos la tolerancia y para los distintos tamaños de tabla estudiados. Los resultados se muestran tan solo para el procesador "medio" (ancho 2, en orden) de los estudiados ya que los resultados son similares en todos los casos.

El resultado sorprendente de la figura 5.13 es que, conforme aumentamos la tolerancia, los resultados de ahorro en tiempo y energía mejoran hasta la tolerancia de nivel 2. A partir de ahí, empeoran hasta llegar, no solo a ser peores que los resultados sin tolerancia, sino hasta producir perdidas. ¿A que se debe este comportamiento? Como se puede ver en las gráficas respectivas de las figuras 5.10 y 5.11, el aumento del error (en este caso del tamaño del fichero final) es pequeño, y el porcentaje de aciertos en la tabla crece mucho con la tolerancia. Sin embargo, recordemos que en este programa el único algoritmo reusado es el estimador de movimiento. Este algoritmo, como ya se ha comentado, no codifica directamente sino que realiza una busqueda que da lugar a la posterior codificación. Así pues, al aumentar la tolerancia,

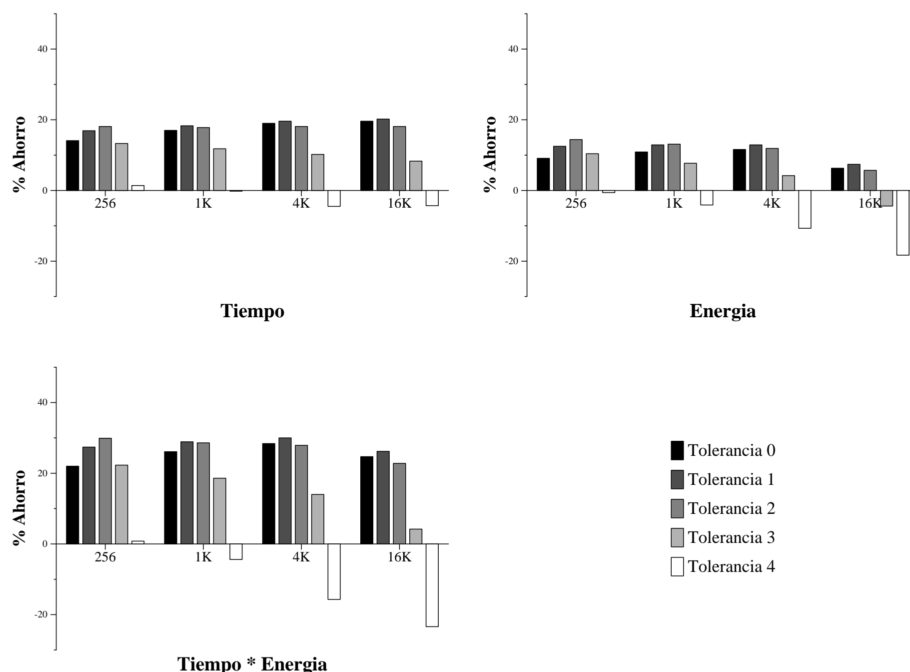
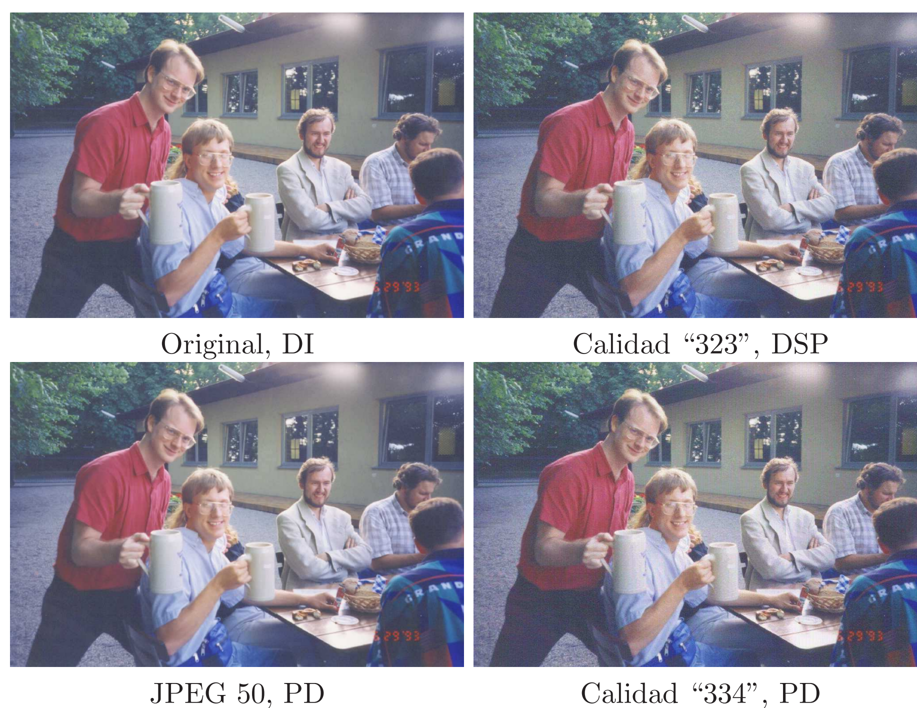


Figura 5.13: Resultados del reuso tolerante de regiones en el programa codificador de vídeo.

lo que hacemos es aumentar el rango de busqueda (ya que no encontramos el destino del bloque tan rápidamente) y, por lo tanto, el programa tarda más tiempo. Sin embargo, con tolerancias bajas, el rango de búsqueda apenas aumenta (introducimos muy poco error), el tamaño del fichero de destino tampoco, y el programa se ejecuta más rápidamente y con menos gasto de energía que sin tolerancia (y este programa era el que mejores resultados obtenía utilizando el reuso de regiones clásico tal y como se ve en la figura 5.6).

La segunda vertiente de resultados es la formada por todo el resto de programas y algoritmos. En estos casos el comportamiento es el esperado: a más tolerancia, más reuso, más error y menos tiempo y energía consumidos en la ejecución de los programas. El caso fácil es, entonces, el programa codificador de voz, *toast*, ya que tan solo contiene un algoritmo. Las pruebas subjetivas y las medidas de error coinciden en que 4 bits tolerados implican una buena calidad de los resultados, mientras que 5 bits obtienen una calidad que varía entre una calidad prácticamente perfecta o ligeras distorsiones (dependiendo del juego de pruebas y del tamaño de la tabla empleada. Así pues, se ha considerado que los resultados óptimos para este programa se obtienen con tolerancia 4.

Los dos programas restantes ya son más complicados de evaluar, debido



Original, DI — Calidad “323”, DSP

JPEG 50, PD — Calidad “334”, PD

Figura 5.14: Diferentes calidades de la imagen *specmun*.

principalmente a que podemos trabajar con varios factores. El criterio que hemos seguido para establecer la tolerancia “óptima” ha sido mantener una coherencia global, es decir, hemos intentado no maximizar la tolerancia de una región y mantener muy baja la tolerancia del resto sino intentar igualar entre ellas todas las tolerancias. La tabla 5.6 nos da una idea de las tolerancias “ideales” en el caso de la tabla de 4096 entradas (tablas menores soportarían, incluso, tolerancias un poco mayores). Como se puede ver hemos agrupado los conjuntos de tolerancias en cuatro categorías: Diferencias Imperceptibles (DI), es decir, diferencias con la imagen original que el programa medidor de errores es capaz de medir pero que no somos capaces de ver en una pantalla de buena calidad; Diferencias Sutilmente Perceptibles (DSP), es decir, somos capaces de percibir que las imágenes no son idénticas pero no sabemos exactamente porqué y, en algunos casos, no sabríamos decir cual (si la original o la “tolerada”) tiene mayor calidad; Pequeña Degradación de la imagen (PD), que nos daría una calidad similar a la que obtendríamos mediante JPEG 50; y, finalmente, No Aceptable (NA) donde se puede ver perfectamente la imagen pero se percibe claramente que hay errores de color o forma en ella. La figura 5.14 ilustra el tipo de imágenes escogidas y los errores obtenidos para la imagen *spemun* que es la que presenta peores resultados de SNR de las tres estudiadas en el algoritmo JPEG. Evidentemente se puede argumentar que las imágenes tienen un tamaño demasiado pequeño

para poder apreciar bien las diferencias (las mismas imágenes a un tamaño mayor se pueden ver en el apéndice B), pero están hechas pensando en el tamaño de la pantalla de un aparato reproductor de fotos.

Programa	Tol. 1	Tol. 2	Tol. 3	Calidad	SNR	Error Max.
JPEG	C. Color	DCT	Cuant.			
JPEG 75	–	–	–	DI	21,6	31,1
JPG 223	2	2	3	DI	21,0	30,4
JPG 323	3	2	3	DSP	19,4	30,0
JPEG 50	–	–	–	PD	18,6	27,4
JPG 233	2	3	3	PD	17,9	29,0
JPG 333	3	3	3	PD	16,6	30,4
JPG 334	3	3	4	PD	16,2	30,4
JPG 400	4	0	0	NA	15,3	40,4
JPG 340	3	4	4	NA	3,8	30,4
H263	C. Gama	Int. V.	Int. H			
H263 122	1	2	2	DI	46,1	62,9
H263 133	1	3	3	DSP	39,8	51,0
H263 222	2	2	2	DSP	37,2	53,1
H263 orig.	-	-	-	PD	40,1	53,4
H263 234	2	3	4	PD	32,7	42,2
H263 244	2	4	4	PD	30,4	39,8
H263 300	3	0	0	NA	25,5	48,2

Tabla 5.6: Tolerancias y calidades en JPEG y H263.

Como se puede ver en la tabla 5.6, el algoritmo JPEG nos permite tolerar dos bits en todas las regiones (y hasta 3 en una de ellas) manteniendo su misma calidad estándar (considerada alta)[5]. La tabla 5.6 nos muestra, además, una segunda medida objetiva del error que hemos creído interesante: el error máximo cometido en un solo píxel de toda la imagen. Este error máximo ha sido calculado como:

$$EMAX = \forall_{x_i} MAX \left(\frac{x_i^2}{MAX(x_i^2)} \right)$$

De esta forma, el error máximo es una medida de cuanto error cometemos como máximo en un píxel de la imagen. Es una medida interesante ya que grandes errores en un solo píxel distorsionan más la imagen que muchos pequeños errores repartidos (pensemos en un punto rojo en un cielo azul).

[5] Las SNR mostradas en la tabla 5.6 difieren de las mostradas en la figura 5.12 debido a que están calculadas sobre la entrada del programa y no sobre la salida original.

Esta medida es una explicación de por qué, a pesar de que nuestra técnica puede tener un error mayor que JPEG 50, sus imágenes se ven bien.

Salida original. Calidad “122”, DI

Calidad “222”, DSP Calidad “244”, PD

Figura 5.15: Diferentes calidades de vídeo.

Los datos del programa decodificador de vídeo H263 son más difíciles de comparar de una forma objetiva ya que no existe un solo parámetro de calidad estándar del vídeo (así que no podemos comparar con una calidad determinada). De todas formas, y usando una medida de calidad similar, los resultados que refleja la tabla 5.6 se ilustran en la figura 5.15. Nótese que todas las figuras, incluida la original, son de una calidad más baja que en el caso del algoritmo JPEG ya que la codificación es más agresiva (y hemos elegido la escena con más errores para que se aprecien mejor las diferencias).

Así pues, para medir los resultados de los programas *cjpeg* y *tmn* se han medido con dos calidades, aquella en la que no se pueden percibir diferencias con respecto al original (DI) y aquella en la se perciben diferencias sutiles (DSP) pero seguimos obteniendo resultados que se pueden considerar muy buenos. Es importante aquí, recordar que estamos hablando siempre de diferencias en una pantalla grande y de buena calidad y, además, de una tabla de reuso grande, con lo cual, en cualquier otra situación, los resultados

serían aún mejores.

El resumen de ganancias obtenidas en tiempo, energía y tiempo×energía para cada uno de los tres procesadores estudiados se puede ver en las figuras 5.16, 5.17 y 5.18 respectivamente. Para cada parámetro se pueden ver las ganancias de cada una de las tablas con la tolerancia de calidad muy buena y, superpuesta en la misma barra, se puede comparar con la ganancia, con el mismo tamaño de tabla, del sistema sin tolerancia (en blanco) y la ganancia con tolerancia de calidad buena (DSP, en negro).

Como se puede observar en las gráficas, el reuso tolerante incrementa de forma substancial las ganancias obtenidas en todas las medidas y en todos los procesadores. Estos aumentos, además, son muy significativos, ya que se producen de forma más pronunciada en los procesadores de bajo consumo y para los tamaños de tabla pequeños (del 6% al 18% para el procesador de ancho 1 y la tabla de 4KB de tamaño total en la medida energía×tiempo). Es decir, es una técnica muy indicada para sistemas portátiles.

Además, el sistema de reuso tolerante permite aumentar las ganancias en aquellos programas que apenas conseguían resultados positivos con el reuso de regiones clásico y elimina las perdidas que se producían en algunos parámetros. De esta forma se eliminan los principales inconvenientes que surgían a la hora de plantearse implementar un sistema de reuso de regiones para procesadores multimedia de bajo consumo.

Las imprecisiones cometidas por el sistema son, tal y como se puede apreciar en las imágenes presentadas, practicamente inapreciables, de forma que, para casos concretos con pantallas pequeñas o de calidad media (cámaras de foto, reproductores portátiles, teléfonos móviles) se pueden utilizar tolerancias todavía más agresivas sin que el usuario lo aprecie y, lógicamente, obtener todavía mejores resultados.

5.6.5 Ajuste dinámico de la tolerancia.

La técnica de reuso tolerante de regiones presenta numerosas ventajas sobre el reuso clásico: mayor cantidad de aciertos en las tablas, mayor impacto con tablas pequeñas y resultados más homogéneos entre aplicaciones, entre otras. Sin embargo presenta una significativa desventaja: precisa de profiling ya que es necesario ajustar la tolerancia antes de utilizar el programa.

Esta desventaja no es demasiado significativa ya que, en cualquier caso, el programador original del programa ya puede haber ajustado su programa para utilizar un determinado tipo de tolerancia en cada una de las etapas

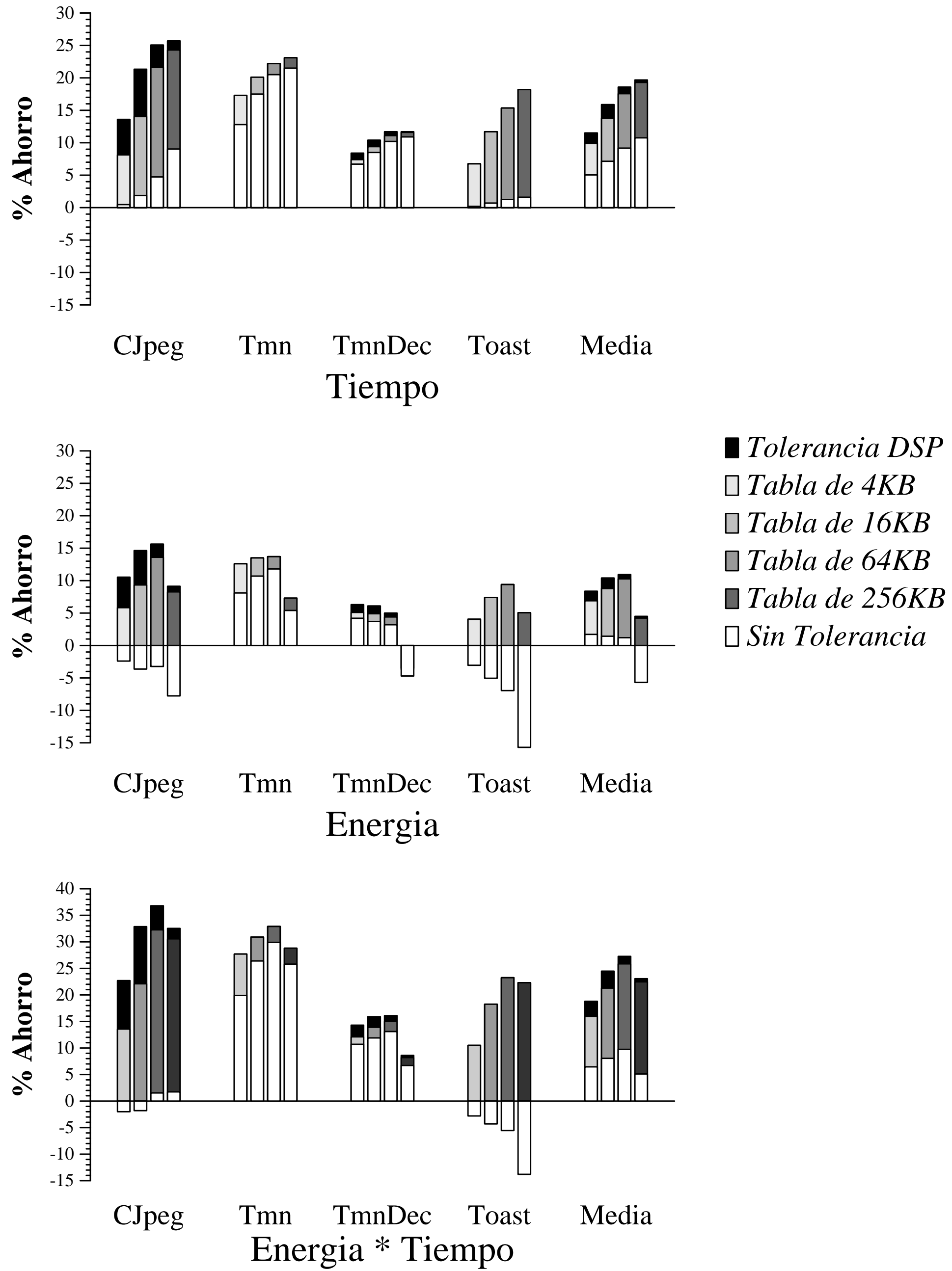


Figura 5.16: Resultados del reuso tolerante de regiones en un procesador de ancho 1.

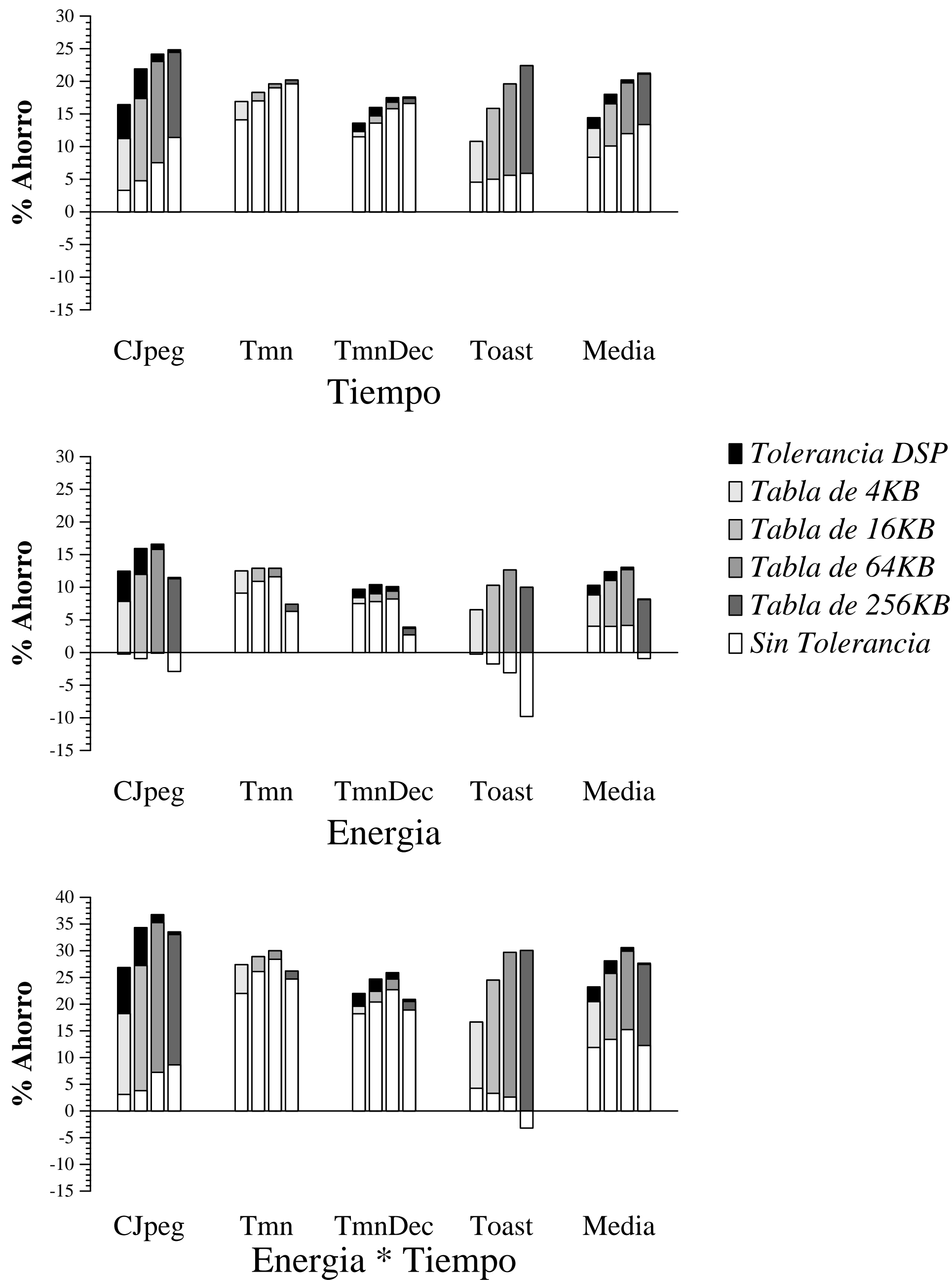


Figura 5.17: Resultados del reuso tolerante de regiones en un procesador de ancho 2.

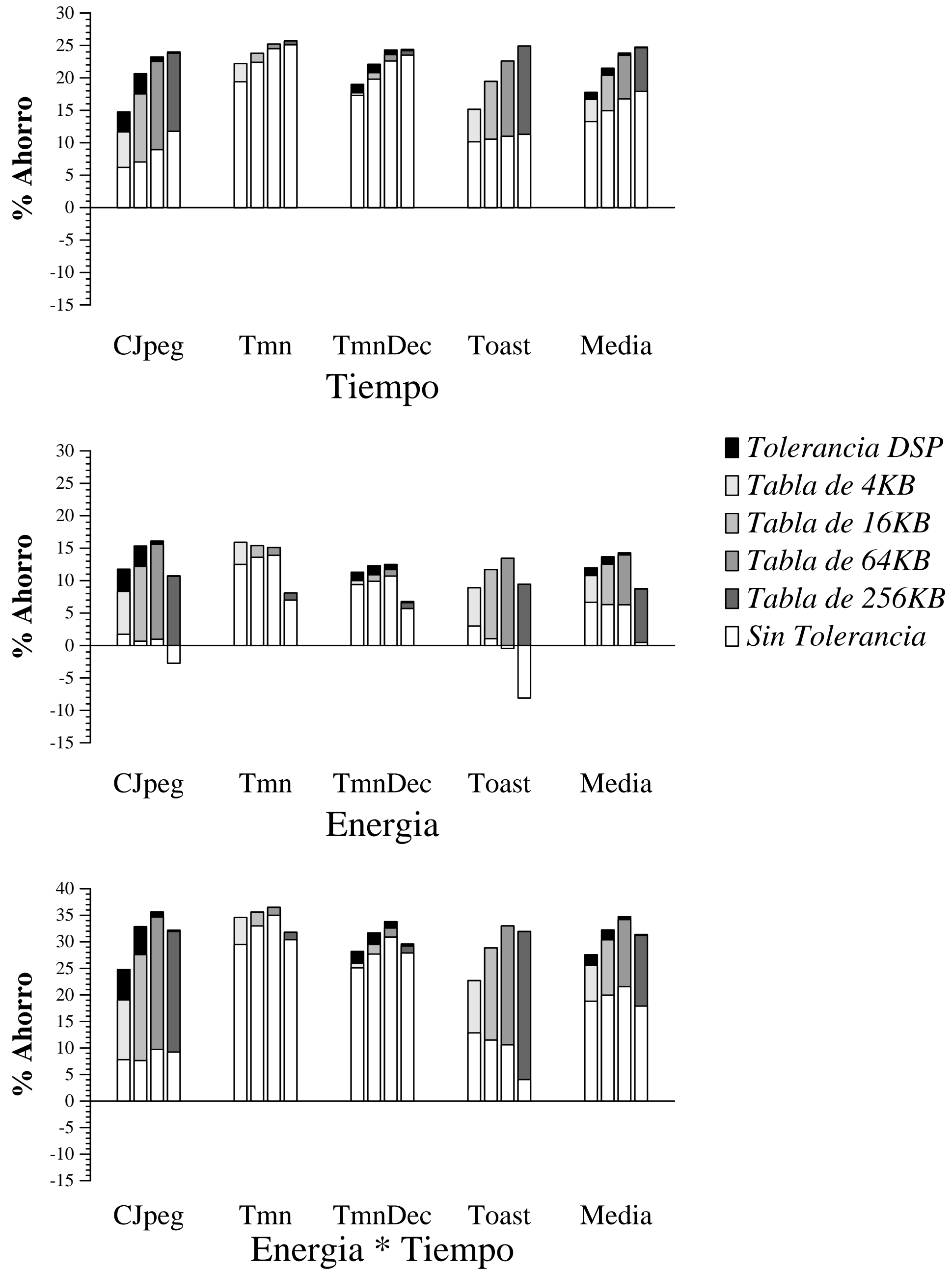


Figura 5.18: Resultados del reuso tolerante de regiones en un procesador fuera de orden de ancho 4.

(regiones) reusadas. Sin embargo, esta es una desventaja significativa si deseamos usar esta técnica junto con algún tipo de mecanismo de detección automática de regiones como los propuestos en [CmWH99] y [CHCmWH00]. En estos casos es necesario algún sistema automático de ajuste de la tolerancia.

Es muy sencillo pensar en un mecanismo software automático que, mediante la ayuda de diversas etapas de profiling sucesivo, ajuste la tolerancia del sistema mediante unas valores de prueba predeterminados, de los cuales conozcamos la salida ideal y una SNR "óptima". Un posible sistema sería:

1. Compilar el programa y obtener un programa la salida original sin tolerancia. Comprobar que es idéntica a la salida original ya almacenada (en realidad este paso es de comprobación y no es estrictamente necesario).
2. Aumentar la tolerancia en 1 en todas las regiones y volver a obtener la salida. Calcular la SNR entre la salida obtenida y la salida original.
3. Comparar la SNR obtenida con un valor umbral prefijado, o bien, con la SNR obtenida entre la salida original y la entrada (este valor puede servir de referencia de calidad). Si la SNR es menor del umbral volver al paso 2.
4. Si la SNR es menor que el umbral (se podrían fijar unos márgenes de unos 3 dB para la "igualdad") restar 1 a la tolerancia de cada región y pasar al siguiente paso.
5. Aumentar en 1 la tolerancia de una región (en cualquier orden) y repetir el cálculo de SNR.
6. Si la SNR se sigue manteniendo menor o en igualdad, cambiar la región y volver al paso anterior.
7. Si la SNR es inferior al umbral, restar 1 a la tolerancia de esa región, cambiar a la región siguiente (hasta que todas hayan sido comprobadas) y volver al paso 5.
8. Finalmente validar las tolerancias obtenidas y compilar la versión definitiva del programa.

Sin embargo, ya hemos visto que las diversas medidas de la SNR de diferentes programas no tienen por que coincidir (lo que para un programa es una buena SNR, para otro es mala) y, además, las señales de entrada (y salida) de una aplicación no tienen porqué tener las características "estándar". Por

si todo lo anterior fuera poco, estos mecanismos son lentos, complejos y, en la práctica, suelen acabar en desuso.

Un mecanismo hardware dinámico presenta numerosas ventajas. Por un lado, no precisa de ningún trabajo anterior del compilador ni del programador, ya que es capaz de funcionar en tiempo real, pero, además, es capaz de adaptarse a las diferentes entradas de la aplicación cada vez que estas varían de características. Las dificultades de un mecanismo de este tipo, sin embargo, también son numerosas: un sistema así necesita de una zona de aprendizaje en la que, habitualmente, registra perdidas de rendimiento; sus resultados no suelen ser tan exactos como los de una evaluación software; y, además, debe ser un mecanismo lo suficientemente sencillo como para que sea factible (y valga la pena) implementarlo.

En esta sección proponemos un mecanismo hardware de ajuste dinámico de la tolerancia para el reuso tolerante de regiones que permite utilizar este sistema propuesto exactamente igual que un sistema de reuso de regiones clásico pero obteniendo mayores ganancias.

La base teórica.

Para poder realizar un mecanismo de aprendizaje, lo primero que tenemos que decidir es ¿qué queremos que el mecanismo aprenda? En nuestro caso, esta pregunta tiene fácil respuesta: queremos que aprenda a reusar de forma tolerante introduciendo una cantidad determinada de error. Así pues, para poder conseguir este objetivo deberemos, en primer lugar, ser capaces de medir el error introducido en un reuso aproximado.

Conseguir que nuestro mecanismo obtenga el valor correcto y el "tolerado" en un reuso es fácil: basta con que, durante la etapa de aprendizaje, la tabla de reuso nos suministre el valor que devolvería y, a la vez, no saltemos la ejecución del código original, de forma que al finalizar dicho proceso, en la actualización de la tabla, tendremos dos juegos de valores de salida: el original y el "tolerado".

Una vez que tenemos ambos valores debemos encontrar una medida del error introducido. Medidas hay muchas, desde una simple distancia euclidiana (es decir, la resta entre los dos juegos de valores) hasta medidas cuadráticas ponderadas según la importancia de los factores involucrados. En nuestro caso no tenemos ninguna información sobre la importancia relativa de las salidas[6], así que no podemos usar ninguna ponderación, pero

[6]En la transformación RGB→YCbCr, por ejemplo, el factor Y es más importante que los factores Cb y Cr, pero ni esta es la única transformación a reusar ni el factor Y tiene

una simple resta es poco para conocer la importancia del error. Por poner un ejemplo, si nuestro valor de salida es 2, un error de una unidad es considerable. En cambio, si nuestra salida original es 10327, 100 unidades de error no son demasiado.

El error que nos vendría bien implementar es, sin duda, la SNR que se obtiene en esta operación en concreto, pero para ello deberíamos restar la salida original menos el error, calcular los cuadrados de la diferencia, sumarlos, obtener por otro lado la suma de los cuadrados de la salida original y dividir ambas magnitudes. Esta operación es demasiado compleja para poder realizarla en hardware en un sistema pensado, como el nuestro, para consumir poca energía. Un sistema mixto, hardware y software, probablemente sea muy complejo y, además, consuma gran parte de las ganancias que hemos obtenido.

Sin embargo, el significado de la SNR, más allá de la pura definición, es dar una magnitud del error introducido. Y precisamente, si pensamos en cantidades representadas en valores binarios, la magnitud es fácil de conocer: depende del primer bit significativo. Así pues, de forma muy sencilla, a partir de un registro se puede obtener su bit más significativo (basta con utilizar un codificador) y, por tanto, su magnitud. Si obtenemos por un lado la magnitud de la salida original del sistema y, por otro, la magnitud del error (que obtenemos mediante una resta), la diferencia entre ambas magnitudes nos da una idea de la SNR. Esto es así porque:

$$SNR = 20 \log \frac{A_S}{A_N} dB$$

Siendo A_S el valor de la amplitud de la señal y A_N el valor de la amplitud del ruido. Si suponemos que la señal se limita al valor de un registro, podemos entonces decir que:

$$A_S = x_i \times 2^i + x_{i-1} \times 2^{i-1} + \cdots + x_0 \times 2^0$$

donde x_i es el valor del bit i-ésimo del registro (y, análogamente, con el valor del registro que contiene el error). Si ahora aproximamos el valor de la A_S por el valor del bit más significativo (e igual para A_N) obtenemos que:

$$SNR \approx 20 \log \frac{x_i \times 2^i}{x_j \times 2^j}$$

porqué aparecer siempre en el primer registro.

siendo j el bit más significativo del error. Como tanto x_i como x_j son los bits más significativos de la señal y del error, su valor debe ser, obligatoriamente, 1, de forma que:

$$SNR \approx 20 \log \frac{2^i}{2^j} = 20 \log 2^{i-j} = (i-j) 20 \log 2 \approx (i-j) \times 6.02 dB$$

Así pues, por cada bit de diferencia entre el bit más significativo de la salida original y el bit más significativo del error, podemos asegurar que obtendremos una relación señal a ruido de 6dB. Si queremos introducir un error tal que la SNR no baje de 30dB deberemos asegurarnos que la distancia entre los bits más significativos no sea menor de 5 bits.

Ahora bien, nuestro sistema deseamos que no introduzca un error de más de 30dB en toda la región reusada en general y no en un valor de esta región, de forma que deberemos obtener de alguna forma el error máximo para toda una muestra de reusos y la señal máxima, también para toda una muestra de reusos.

El hardware de aprendizaje.

Así pues, nuestro hardware consta, por un lado de la tabla hardware de reuso tolerante propuesta en el apartado anterior. Debemos diseñarla de forma que almacene como entradas a la tabla, no las entradas toleradas sino las entradas originales. A partir de aquí tendremos varias opciones de comportamiento:

- Fallo en la tabla: todo funciona de forma normal, es decir, la tabla devuelve un fallo, se ejecuta el código original y al finalizar se actualiza la tabla.
- Acierto sin aprendizaje: fuera de la zona de aprendizaje (que nos definirá un registro contador interno de la tabla) los aciertos también se tratarán de la forma clásica, es decir, se enviará hacia la salida los valores almacenados en la tabla y se evitará la ejecución del código original. Para saber si se dan estos aciertos se compararan los valores de entrada con los valores almacenados en la tabla, quitandoles a todos ellos los bits indicados en el registro de tolerancia de la región en uso en ese momento.

- Acierto con aprendizaje. Si el contador de aprendizaje no vale 0, estaremos todavía en la zona de aprendizaje. En esta zona podremos tener dos tipos de acierto: un acierto exacto y un acierto “tolerado”. Para poder distinguirlos, nuestro comparador debe ser capaz de comparar las entradas a la tabla y los valores almacenados, por un lado sin los bits indicados en el registro de tolerancia y, por otro, debe comparar solo los bits tolerados. Si hay acierto en ambas comparaciones tendremos un acierto exacto, sino (solo hay acierto sin los bits tolerados) tendremos un acierto con tolerancia.

  - Acierto exacto. En este caso, no tenemos nada que aprender, ha habido un acierto exacto y por lo tanto la salida es la correcta. La tabla devuelve acierto, se envía la salida almacenada hacia los registros de destino, se salta el código original y se prosigue normalmente.
  - Acierto con tolerancia. En este caso, si estamos dentro de la zona de aprendizaje, debemos saber si nos equivocaríamos o no. Para ello la tabla hardware devuelve un fallo, de forma que el código original se ejecuta normalmente. Al acabar dicho código las instrucciones de actualización de la tabla le envían a esta el resultado original. La tabla en vez de actualizarse (no olvidemos que habíamos tenido un acierto) obtiene por un lado la magnitud (el número de bit más significativo) de cada valor original y, por otro, la magnitud de la resta entre cada valor original y cada valor almacenado. Estas magnitudes se almacenan en 2 registros especiales (nótese que bastarían con 4 bits por registro) que contienen el máximo de ambas a lo largo de toda una serie de aciertos con tolerancia en la zona de aprendizaje. Finalmente el registro de aprendizaje se decrementaría ya que hemos aprendido un nuevo valor.

Cada cierta cantidad de aciertos con tolerancia (necesitaríamos un segundo registro contador o establecer cuentas parciales del primero) la tabla comprobaría el valor de la resta entre la magnitud de las salidas originales y la magnitud del error. Si dicha resta es mayor de un cierto umbral (5 o 6, según lo expuesto en el apartado anterior) para todos los registros de salida, el registro de tolerancia se incrementaría en 1. Si en cambio, la resta es menor de otro umbral (4 o 5, para permitir una cierta histéresis) para cualquier registro de salida, el registro de tolerancia se decrementaría en 1. Obviamente, aquellos registros de salida cuya magnitud original sea 0 se ignorarían (ya que probablemente la región no los está usando). Una vez modificada la tolerancia, restableceríamos los valores de magnitud a 0 para poder iniciar otra secuencia de aprendizaje, así que el registro contador de

aprendizaje tiene que tener un valor lo bastante alto como para ser varias veces múltiplo de una secuencia de aprendizaje.

El sistema anterior tan solo tiene un punto débil y, es que es incapaz de realizar una transición de 0 a 1, ya que si el valor del registro de tolerancia es 0 nunca se produce un acierto tolerado y, por tanto, el sistema es incapaz de aprender. Para evitar este caso, cualquier fallo que se de mientras el registro contador de aprendizaje no valga 0 actualiza, además de la tabla, el valor de la magnitud de la salida original. Si esta magnitud, en cualquier momento del aprendizaje, es mayor que el umbral utilizado para incrementar en 1 el nivel de tolerancia, para todos los registros de salida (no se tendrían en cuenta aquellos cuyo valor fuese 0) y la tolerancia fuese 0, entonces se actualizaría a 1 el nivel de tolerancia. Este comportamiento tiene, además, la ventaja de actualizar la magnitud de la salida original en los fallos durante el aprendizaje. De esta forma nos aseguramos que si en una secuencia de aprendizaje solo acertamos en valores muy pequeños, no bajemos indefinidamente el nivel de tolerancia. Además, nótese que trabajando de esta forma, si la región no admite tolerancia alguna, el nivel de tolerancia recaerá en 0 al final de la última secuencia de aprendizaje[7]. De esta forma no habremos introducido error alguno durante la etapa de aprendizaje (nunca lo hacemos) pero sí habremos aprovechado los aciertos exactos.

Valores iniciales para el aprendizaje.

Ahora que ya tenemos definido un hardware de aprendizaje, debemos acabar de definir una serie de valores iniciales para dicho hardware. Estos valores son los siguientes:

- Valor de la secuencia de aprendizaje. Este valor nos determinará cuantos aciertos tolerados deberemos obtener antes de atrevernos a actualizar la tolerancia. Un valor muy bajo generará inestabilidades al sistema ya que la media no será efectiva, mientras que un valor muy alto provocará que la fase de aprendizaje dure demasiado dentro de la ejecución del programa. Hemos visto que prácticamente cualquier valor superior a 256 muestras es suficiente para obtener tendencias, así que hemos fijado el valor a 512.

- Valor inicial del contador de aprendizaje. Este valor debe ser un múltiplo entero del valor de la secuencia de aprendizaje. Nuevamente, cuanto menor sea este valor menor será la duración del aprendizaje,

[7] Esto será así siempre y cuando tengamos cuidado de hacer que el valor del contador de aprendizaje sea un múltiplo exacto del valor de la secuencia de aprendizaje.

pero debe de ser lo bastante alto como para alcanzar la salida estable. Dado que el sistema, como mucho necesita realizar 3 actualizaciones del nivel de tolerancia (ver valor inicial de la tolerancia más abajo), hemos decidido fijar este valor en 6 veces el valor de la secuencia de aprendizaje: 3072.

- Valor inicial de la tolerancia. La tolerancia inicial no es importante a la hora de determinar el valor final después del aprendizaje. Los experimentos realizados demuestran que la tolerancia siempre converge hacia los mismos valores. Sin embargo, si el valor inicial es 0, esta convergencia tarda mucho tiempo, ya que para ciertos programas el numero de aciertos con tolerancia 1 es pequeño y, además, la cantidad se saltos necesaria para llegar a tolerancia 5, por ejemplo, es alta, obligando a mantener muy alto el valor inicial del contador de aprendizaje. En cambio, con un valor inicial medio, 3 o 4, en aquellos programas donde esta tolerancia es demasiado alta, el valor decae rápidamente, así que hemos fijado el valor inicial en 3.
- Umbral de subida. Este valor decide cuando debemos subir un nivel de tolerancia. Como ya se ha razonado en el apartado anterior, este valor ha de estar alrededor de 5 o 6 (recordemos que se mide como diferencia entre magnitudes).
- Umbral de bajada. Este valor, en cambio, define cuando debemos bajar un nivel de tolerancia. Para ser coherentes con el umbral de subida, el umbral de bajada ha de ser como mucho igual a este, así que sus posibles valores son 4 o 5. Una combinación entre ambos valores que da resultados muy estables es 6 y 4 (aunque las combinaciones restantes: 6,5; 5,5 y 5,4 también dan lugar a buenos, y parecidos, resultados).

Por razones prácticas, además, hemos definido el nivel de tolerancia para una región como un contador saturado de 3 bits (valores de 0 a 7). Esto es suficiente para todos los programas probados y mantiene el hardware limitado a un nivel razonable.

Resultados.

Para comprobar la eficiencia del sistema de ajuste dinámico de la tolerancia, hemos procedido a realizar la medida de las tolerancias calculadas por el sistema, para cada una de las tablas, y los resultados de SNR obtenidos. El resumen de los resultados por programas se puede ver en la tabla 5.7. En la tabla se muestran, para cada programa y tamaño de tabla los valores

Programa	Tabla	Tol. dinámica	Tol. Ref.	SNR	SNR Ref.
Cjpeg	256	314 / 313	323	35,0	19,4
	1K	314 / 313	323	32,7	19,4
	4K	314 / 313	323	31,6	19,4
Tmn Cod.	256	2	1	–	–
	1K	1	1	–	–
	4K	1	1	–	–
Tmn Dec.	256	233	222	41,5	37,2
	1K	133	222	42,8	37,2
	4K	133	222	39,8	37,2
Toast	256	5 / 3	4	37,6 / 66,8	27,8
	1K	4 / 2	4	34,8 / 79,9	27,8
	4K	3	4	41,4	27,8

Tabla 5.7: Resultados de tolerancias ajustadas dinámicamente (umbral de subida=6).

de tolerancia y SNR que se obtienen dinámicamente frente a los valores de referencia escogidos en la sección anterior. Los valores de referencia estaban calculados para la tabla de 64KB (asumiendo que eran bastantes para tablas más pequeñas), así que no es de extrañar que en algunos casos (decodificador de vídeo *tmn* y tabla de 4KB) el sistema dinámico escoja tolerancias mayores y, aún así, consiga mejores SNR. Particularmente significativo es el caso de la imagen *specmun* codificada mediante el programa *cjpeg*, ya que el algoritmo ajusta, adecuadamente, la tolerancia más baja en la etapa de cuantificación a esta imagen que es la que presenta peor comportamiento de SNR de las tres.

En el único caso en el que los resultados variaban mucho entre las diferentes entradas (programa *Toast*), hemos incluido las SNR de cada resultado para que se pueda comprobar que el sistema dinámico siempre se mueve en la zona segura, es decir, nos da resultados sin ningún error apreciable aún con tablas grandes y pantallas de buena calidad, para todas las entradas. Es posible, fácilmente, volver el predictor más seguro (aumentando, por ejemplo, a 5 el umbral de bajada) o más agresivo (quizás más adecuado para sistemas de bajo consumo y tablas pequeñas) disminuyendo el umbral de subida a 5. Los resultados de este último caso se reflejan en la tabla 5.8.

Aunque los resultados de la tabla 5.8 son buenos en cuanto a SNR (tan solo hay un caso con resultados peores que la referencia), como podemos ver si comprobamos las tolerancias, estas no se ajustan de la forma esperada por nosotros. Como podemos ver en el caso del programa *cjpeg* sus resultados son menos homogéneos que los escogidos por nosotros deliberadamente y,

Programa	Tabla	Tol. dinámica	Tol. Ref.	SNR	SNR Ref.
Cjpeg	256	315	323	31,0	19,4
	1K	315	323	29,3	19,4
	4K	314 / 315	323	28,8	19,4
Tmn Cod.	256	1	1	–	–
	1K	1	1	–	–
	4K	2	1	–	–
Tmn Dec.	256	243	222	40,5	37,2
	1K	233	222	36,7	37,2
	4K	234	222	32,5	37,2
Toast	256	5 / 3	4	37,6 / 66,8	27,8
	1K	4 / 2	4	44,8 / 79,9	27,8
	4K	3	4	41,4	27,8

Tabla 5.8: Resultados de tolerancias ajustadas dinámicamente (umbral de subida=5).

consecuentemente, pueden aparecer errores más visibles en los resultados. La figura 5.19 muestra los dos casos más llamativos y, como se puede observar, aún así los errores son prácticamente inapreciables. Consecuentemente, esta configuración sería nuestra propuesta de implementación para procesadores de bajo consumo y/o con dispositivos de visualización de calidad media.

Finalmente las gráficas de las figuras 5.20, 5.21 y 5.22 muestran los resultados de ahorro en tiempo, energía y tiempo×energía obtenidos para cada uno de los tres procesadores estudiados mediante el sistema de ajuste dinámico del reuso. Hemos utilizado la aproximación más agresiva para las dos tablas de menor tamaño y la menos agresiva para las dos tablas mayores. Para mayor claridad, además, en la misma gráfica se muestran los resultados obtenidos con el reuso clásico sin tolerancia (color blanco) y con el reuso tolerante ajustado a mano en su versión más agresiva (DSP, color negro). En los casos en los que no se observa barra de color negro, se debe a que el reuso dinámico presenta resultados iguales o mejores (decodificador *tmn*) que el reuso tolerante clásico. En aquellos casos en los que no se observa barra gris se debe a que el reuso dinámico presenta los mismos resultados que el reuso clásico sin tolerancia.

A partir de las figuras 5.20, 5.21 y 5.22 se puede comprobar que el reuso dinámico presenta unas características muy adecuadas. En la tabla más pequeña y para procesadores de bajo consumo presenta perdidas de menos del 1,5% de ganancia con respecto al reuso tolerante ajustado a mano. Para el procesador de altas prestaciones y la tabla mayor implementable (la de 64KB) las perdidas respecto al ideal son de menos del 5%, presentando en

Figura 5.19: Peores resultados de calidad con ajuste dinámico agresivo de la tolerancia y tabla de 64 KB.

ambos casos unas ganancias de más del 10% sobre el reuso clásico.

El reuso dinámico, además, al igual que el reuso tolerante, presenta ganancias para todas las medidas y procesadores con una sola excepción: el procesador de voz *toast* donde presenta pequeñas pérdidas debidas a la baja tolerancia ajustada para la entrada *clinton.pcm*. Esto se debe a la gran variabilidad de la señal de voz que presenta largos (relativamente hablando) periodos de silencio alternados con largos periodos de actividad, periodos que pueden alterar los resultados del sistema de aprendizaje. Esto es así, porque si coinciden dos o tres secuencias de aprendizaje con un silencio, los errores cometidos serán muy altos en valor absoluto (con respecto al valor de la salida original que será idealmente 0) de forma que el sistema de aprendizaje se ajusta a la baja.

Se puede decir, pues, que el sistema de ajuste dinámico de la tolerancia presenta resultados muy buenos, dando lugar a resultados cercanos o incluso superiores al ajuste manual y con unos errores introducidos que, en el peor de los casos (tal y como se puede ver en la figura 5.19) son prácticamente inapreciables.

5.7 Conclusiones.

En este capítulo hemos presentado el problema del reuso de regiones aplicado a los algoritmos multimedia. Se ha visto que esta técnica presenta un alto potencial pero que, sin embargo, limitada a tamaños de tabla realistas, sus beneficios son escasos (o negativos) y poco útiles, sobretodo aplicada a procesadores de bajo consumo. Para este tipo de procesadores el tamaño de

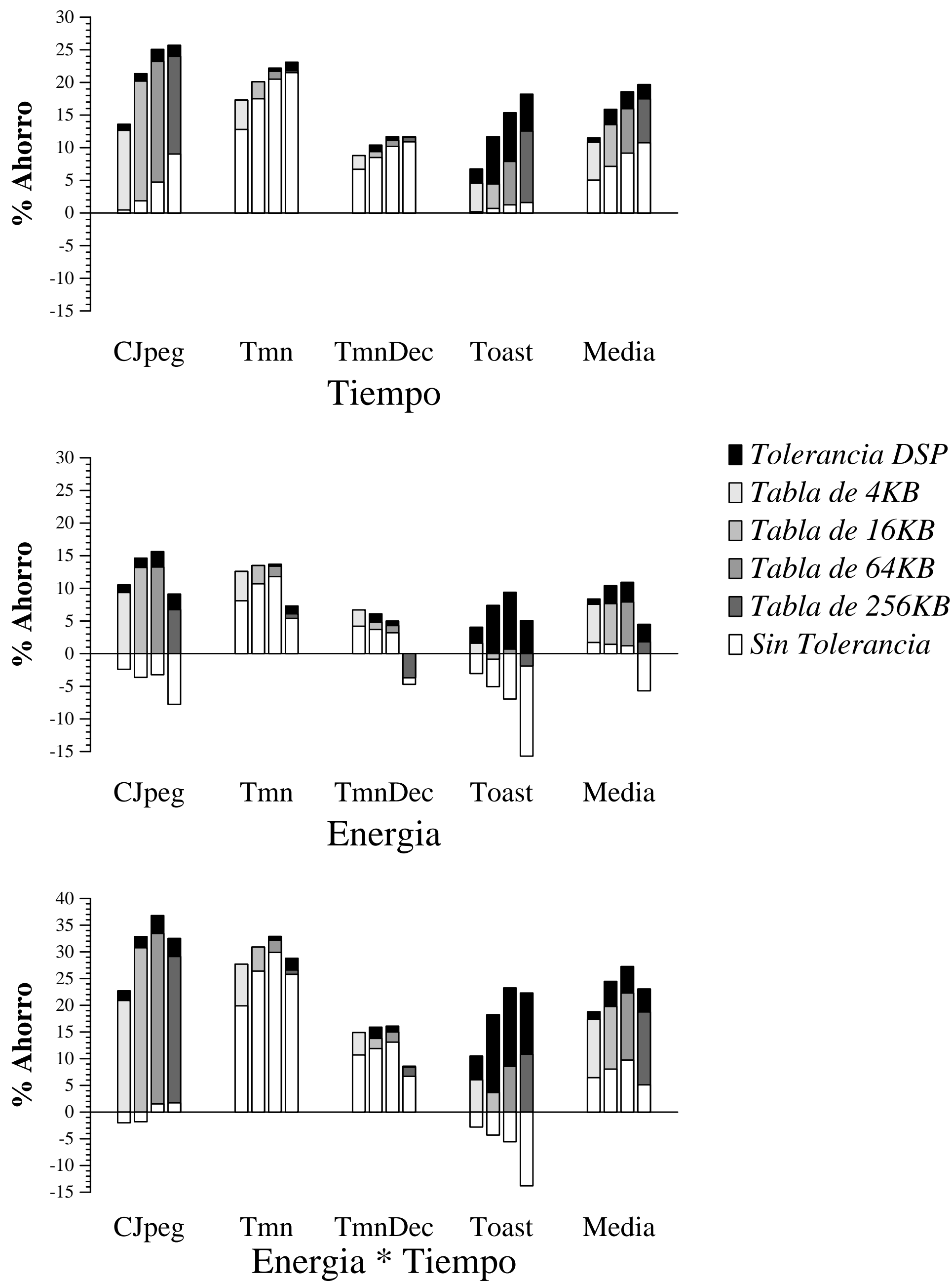


Figura 5.20: Resultados del reuso tolerante dinámico de regiones en un procesador de ancho 1.

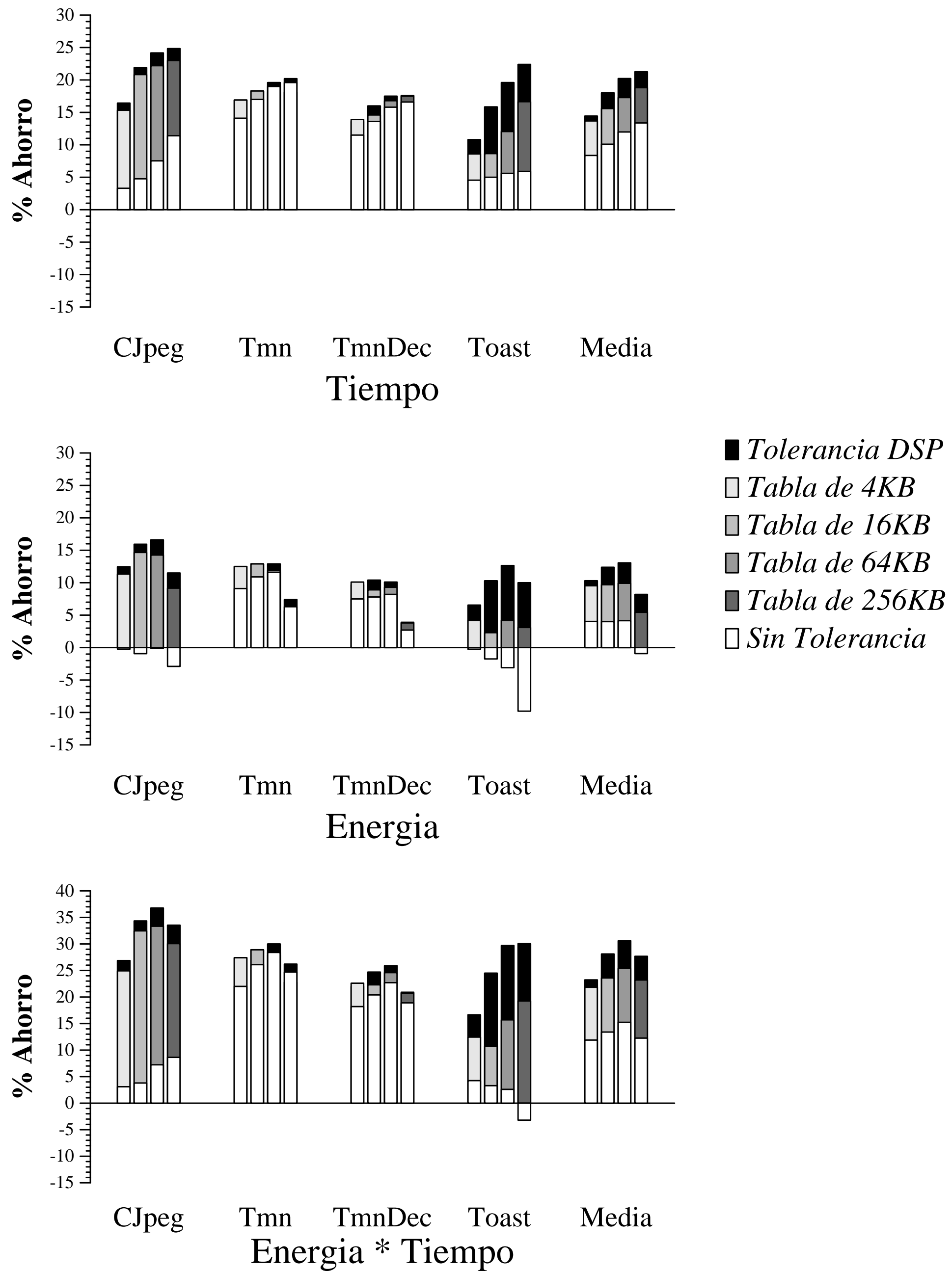


Figura 5.21: Resultados del reuso tolerante dinámico de regiones en un procesador de ancho 2.

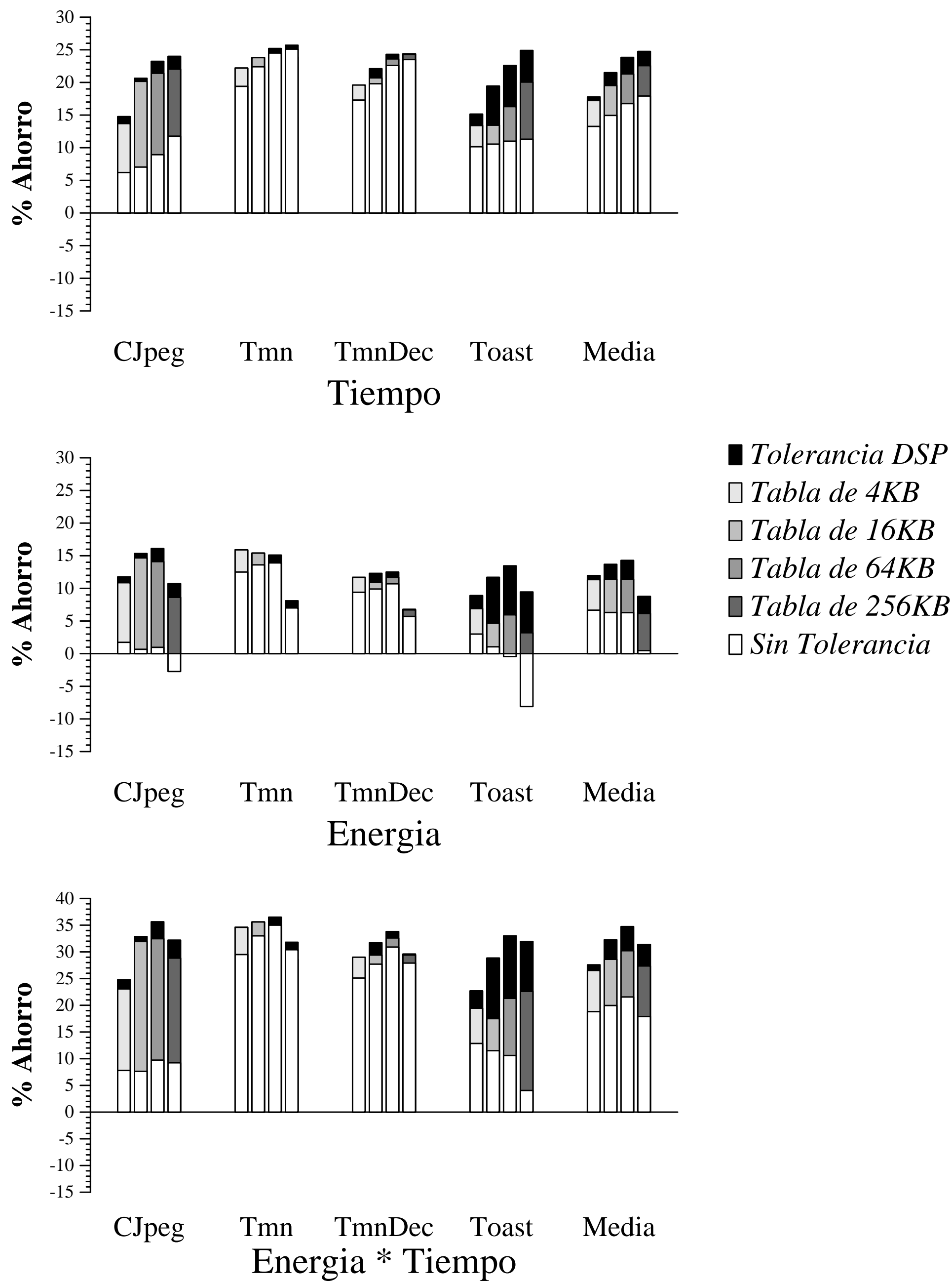


Figura 5.22: Resultados del reuso tolerante dinámico de regiones en un procesador fuera de orden de ancho 4.

las tablas y el consumo de estas supera las posibles ganancias obtenidas.

Se ha presentado a continuación un sistema para aplicar el paradigma del cálculo difuso a la técnica de reuso que hemos denominado reuso tolerante. El reuso tolerante permite incrementar mucho el porcentaje de aciertos de una técnica de reuso clásica con tamaños de tabla realistas de forma que se obtienen ganancias de tiempo y energía significativas (de casi el 20%) en procesadores de bajo (y alto) consumo. Esta técnica requiere de un hardware prácticamente idéntico a los sistemas de reuso clásico y de un ajuste de la tolerancia previo a la ejecución del programa.

Finalmente se ha presentado una técnica que ajuste dinámico de la tolerancia que permite a un sistema de este tipo adaptarse de forma automática a la entrada de un programa que presenta reuso de regiones. Con esta técnica se consiguen resultados casi tan buenos como con el ajuste manual de la tolerancia (en algunos casos incluso mejores) mientras que los errores introducidos se mantienen siempre dentro del rango de lo difícilmente perceptible (para el sistema de ajuste más conservador) o de las pequeñas pérdidas (para el sistema de ajuste más agresivo, indicado para sistemas móviles pequeños).

6

Conclusiones y extensiones futuras

Resumen

En este capítulo se resumen las motivaciones y aportaciones de esta tesis. Asimismo se resumen las principales conclusiones y resultados a los que ha dado lugar, así como las posibles extensiones futuras que pueden derivarse.

6.1 Objetivos y motivaciones.

Esta tesis surge de la observación de la importancia que los algoritmos multimedia han ido adquiriendo a lo largo del tiempo en el campo de la computación. Si hace no tantos años este tipo de programas se ejecutaban en una minoría de equipos de profesionales especializados, hoy en día este tipo de programas se han expandido hasta llegar a un punto en el que los usa toda la sociedad en general.

Si nos centramos en los equipos portátiles, este efecto es aún más focalizado e impresionante. Todo el mundo usa equipos portátiles capaces de reproducir música e imágenes y el vídeo es el siguiente paso a seguir. En este sentido, estos equipos se enfrentan a un doble problema: por un lado deben ser capaces de aumentar las prestaciones para poder procesar formatos cada vez más complejos y, por otro, deben mantener el consumo acotado de forma que las baterías no se agoten en un tiempo relativamente corto.

La forma en que se definirán los procesadores multimedia del futuro es todavía incierta. Muy diversas arquitecturas luchan por la supremacía mientras diversas orientaciones compiten entre si en el mercado sin que haya un claro vencedor. De momento parece haber sitio para todo el mundo.

El objetivo de esta tesis ha sido proponer una nueva técnica, el cálculo difuso, e implementaciones de esta que permitan a los procesadores de bajo consumo mejorar su rendimiento sin incrementar su consumo. Mediante esta técnica, estos procesadores serán capaces con mayor facilidad de trabajar con un espectro más amplio de aplicaciones y algoritmos alcanzando sus necesidades de capacidad de proceso.

6.2 Aportaciones y conclusiones.

En esta tesis se propone una nueva forma de trabajar con los datos multimedia: el cálculo difuso. Esta técnica se diferencia de todas las demás propuestas en el ámbito del hardware para cualquier tipo de datos en que propone intercambiar velocidad y consumo por precisión. De esta forma una unidad difusa será más rápida y consumirá menos que una unidad convencional, pero a cambio su salida no será del todo exacta.

La gran ventaja de este tipo de unidades a la hora de computar datos multimedia es que estos datos ya presentan una resistencia intrínseca a pequeñas imprecisiones. Esta resistencia es aprovechada por numerosos algoritmos de compresión de datos ampliamente difundidos para conseguir

una alta ratio de compresión que no se pueden alcanzar en algoritmos sin perdidas. Nosotros proponemos extender este intercambio al hardware y, de forma ortogonal a las ganancias en compresión, conseguir altos índices de ahorro en tiempo y energía.

Hemos realizado diversas propuestas de implementación del cálculo difuso que se pueden clasificar en dos grandes grupos: implementaciones para el cálculo difuso de instrucciones e implementaciones para el cálculo difuso de regiones.

Dentro del primer grupo se han estudiado principalmente las instrucciones de punto flotante. Estas instrucciones son especialmente adecuadas para nuestros objetivos ya que, por un lado, se utilizan en cada vez más implementaciones de algoritmos multimedia y, por otro, consumen una gran cantidad de recursos (tiempo y energía) de los procesadores de bajo consumo.

Para las instrucciones de punto flotante se han propuesto dos aproximaciones: la implementación de unidades funcionales difusas (menos precisas que las convencionales) y la expansión de las capacidades del reuso de instrucciones clásico mediante el reuso tolerante. Esta última técnica consigue mejorar los resultados del reuso clásico para instrucciones de punto flotante (que no es rentable utilizando debido al enorme rango de los valores en punto flotante), de forma que es posible obtener altos índices de acierto en las tablas. Cualquiera de las dos técnicas propuestas consigue ahorros cercanos al 25% en el factor energía×tiempo en procesadores de bajo consumo, mientras que la combinación de ambas técnicas alcanza ahorros del 35% en el factor energía×tiempo (12% en tiempo y 25% en energía).

El cálculo difuso de regiones se ha elaborado a partir de las técnicas clásicas de reuso de regiones. Se ha estudiado una posible implementación de estas técnicas y se ha visto que sus resultados no son buenos con tamaños de tabla realistas debido, principalmente, a la enorme cantidad de datos de una señal multimedia. Así pues se ha propuesto un sistema de reuso tolerante de regiones que permite obtener buenos índices de reuso de regiones en datos multimedia manteniendo las tablas dentro de unos límites realistas.

Dado que uno de los principales inconvenientes del reuso tolerante de regiones es la necesidad de ajustar, para cada algoritmo, la tolerancia utilizada, se ha propuesto además un sistema dinámico capaz de ajustar la tolerancia de forma automática. El sistema dinámico presenta muy buenos resultados, consiguiendo ajustar la tolerancia casi a sus niveles óptimos y dando lugar a que el sistema de reuso consiga ahorros para todo el programa del 25% en el factor energía×tiempo.

Las técnicas propuestas, además, se han evaluado para procesadores de alto rendimiento donde el consumo no es una variable tan importante, pero si el tiempo de proceso. En estos procesadores se ha podido ver que estas técnicas no influyen tanto en la energía global consumida debido, principalmente, al consumo del resto de secciones del procesador. Sin embargo, por este mismo motivo, se ha podido ver que los puntos conflictivos hacia los que se dirigen nuestras técnicas (instrucciones de punto flotante y regiones intensas computacionalmente de los núcleos de las aplicaciones) son cuellos de botella también en los procesadores de alto rendimiento y, por tanto, las ganancias en tiempo obtenidas en estos procesadores son significativas. Los resultados obtenidos en el factor energía×tiempo son aún mejores en estos procesadores que en los de bajo consumo.

6.3 Extensiones futuras.

El cálculo difuso presenta todo un abanico de nuevas posibilidades. Existen muchos posibles caminos de desarrollo que merecen la pena ser estudiados.

- Los sistemas presentados se mantienen todos en la parte segura en cuanto a las imprecisiones cometidas en la señal. Es posible ser mucho más agresivo con la tolerancia si nos restringimos al ámbito de los equipos portátiles (cámaras de fotos, reproductores multimedia, teléfonos móviles, agendas electrónicas, etc.)
- Es posible combinar las técnicas estudiadas en algoritmos que combinen regiones de cálculo entero con zonas de cálculo en coma flotante.
- Para dispositivos portátiles se debería evaluar la posibilidad de incorporar, en vez de una unidad de coma flotante de doble precisión, una de simple y un sistema de reuso de instrucciones tolerante ya que estos dos sistemas pueden, conjuntamente, ser más rápidos y consumir menos que el habitual.
- Un hardware reconfigurable (tipo FPGA) y un sistema de configuración basado en redes neuronales podrían ser una alternativa interesante al reuso tolerante de regiones. Este hardware se convertiría en una unidad tolerante de proceso de regiones que se ejecutaría en lugar del sistema normal, una vez que se hubiera realizado el aprendizaje.

Aparte de estas ideas, la idea de que un hardware no siempre tiene que generar una respuesta exacta y aún así el resultado puede ser correcto es un campo totalmente nuevo. Otras aplicaciones están todavía por descubrir, y

quizás la clave sea, simplemente, plantearse la posibilidad de procesar los datos de alguna otra forma, diferente y más adecuada.

Bibliografía

[3DN99] 3dnow! technology manual. Technical Report http://www.amd.com, Advanced Micro Devices, Inc., 1999.

[75485] ANSI/IEEE Standard 754-1985. Standard for binary floating point arithmetic. Technical report, IEEE, 1985.

[75406] ANSI/IEEE Standard 754-1985. Revising ansi/ieee std 754-1985. Technical report, http://754r.ucbtest.org, 2006.

[ACS+03] Carlos Alvarez, Jesus Corbal, Esther Salami, Jose A. Fonollosa, and Mateo Valero. A fast, low-power floating point unit for multimedia. *2nd Workshop on Application Specific Processors in conjuction with MICRO 36*, 2003.

[ACSV01] Carlos Alvarez, Jesus Corbal, Esther Salami, and Mateo Valero. On the potential of tolerance reuse for multimedia applications. *International Conference on Supercomputing, ICS-01, Sorrento,* Italy, 2001.

[ACSV02] Carlos Alvarez, Jesus Corbal, Esther Salami, and Mateo Valero. Initial results on fuzzy floating point computation for multimedia processors. *Computer Architecture Letters, Vol. 1,No. 1*, 2002.

[ACV05] Carlos Alvarez, Jesus Corbal, and Mateo Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Trans. Comput.*, 54(7):922–927, 2005.

[AD98] H. Akkary and M. Driscoll. A dynamic multithreaded processor. *International Symposium on Microarchitecture*, 1998.

[AFL97] Mir Azam, Paul Franzon, and Wentai Liu. Low power data processing by elimination of redundant computations. In *ISLPED '97: Proceedings of the 1997 international symposium on Low power electronics and design*, pages 259–264, New York, NY, USA, 1997. ACM Press.

[ANUN98] F. Arakawa, O. Nishii, K. Uchiyama, and N. Nakagawa. Sh4 risc multimedia processor. March-April 1998.

[AS90] E H Adelson and E P Simoncelli. Subband image coding with three-tap pyramids. In *Proc Picture Coding Symposium*, pages 3.9.1–3.9.3, Cambridge, MA, 1990.

[ATI01] ATI. Radeon 8500. *Whitepaper*, http://www.ati.com/na/pages/products/pc/radeon_8500/, 2001.

[BA97] Douglas C. Burger and Todd M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, 1997.

[BM99] David Brooks and Margaret Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, page 13, Washington, DC, USA, 1999. IEEE Computer Society.

[BOP99] Bops. http://www.bops.com, 1999.

[BTM00] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA*, pages 83–94, 2000.

[CEV99] Jesus Corbal, Roger Espasa, and Mateo Valero. Mom: a matrix simd instruction set architecture for multimedia applications. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 15, New York, NY, USA, 1999. ACM Press.

[CEV01] Jesus Corbal, Roger Espasa, and Mateo Valero. Dlp +tlp processors for the next generation of media workloads. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 219, Washington, DC, USA, 2001. IEEE Computer Society.

[CEV02] Jesus Corbal, Roger Espasa, and Mateo Valero. Three-dimensional memory vectorization for high bandwidth media memory systems. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 149–160, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[CF00a] D. Citron and D. Feitelson. Low power instruction memoization. In *Kool-Chips Workshop in Conjunction with MICRO 33*, 2000.

[CF00b] D. Citron and D. Feitelson. The organization of lookup tables in instruction memoization. In *Technical Report, 2000-4*, Hebrew University of Jerusalem, 2000.

[CFR98] Daniel Citron, Dror Feitelson, and Larry Rudolph. Accelerating multi-media processing by implementing memoing in multiplication and division units. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 252–261, New York, NY, USA, 1998. ACM Press.

[CH05] Xueqi Cheng and Michael S. Hsiao. Region-level approximate computation reuse for power reduction in multimedia applications. In *ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design*, pages 119–122, New York, NY, USA, 2005. ACM Press.

[CHCmWH00] Daniel A. Connors, Hillery C. Hunter, Ben-Chung Cheng, and Wen mei W. Hwu. Hardware support for dynamic activation of compiler-directed computation reuse. *SIGARCH Comput. Archit. News*, 28(5):222–233, 2000.

[CmWH99] Daniel A. Connors and Wen mei W. Hwu. Compiler-directed dynamic computation reuse: Rationale and initial results. In *International Symposium on Microarchitecture*, pages 158–169, 1999.

[Con05] MediaBench Consortium. Mediabench ii. http://euler.slu.edu/ fritts/mediabench/, 2005.

[CSEV99] Jesus Corbal, Esther Salami, Roger Espasa, and Mateo Valero. An evolution of different dlp alternatives for the embedded multimedia domain. In *MP-DSP 1st Workshop on Media Processors and DSP´s*, 1999.

[CVE99] Jesus Corbal, Mateo Valero, and Roger Espasa. Exploiting a new level of dlp in multimedia applications. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 72–79, Washington, DC, USA, 1999. IEEE Computer Society.

[DD97] Keith Diefendorff and Pradeep K. Dubey. How multimedia workloads will change processor design. *Computer*, 30(9):43–45, 1997.

[DDHS00] K. Diefendorff, P.K. Dubey, R. Hochsprung, and H. Scales. Altivec extension to powerpc accelerates media processing. *IEEE Micro*, pages 85–95, March-April 2000.

[Dev] Analog Devices. Introducing tigersharc. *Whitepaper*, http://www.analog.com/new/ads/html/SHARC2.

[EAE+02] Roger Espasa, Federico Ardanaz, Joel Emer, Stephen Felix, Julio Gago, Roger Gramunt, Isaac Hernandez, Toni Juan, Geoff Lowney, Matthew Mattina, and Andre Seznec. Tarantula: a vector extension to the alpha architecture. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 281–292, Washington, DC, USA, 2002. IEEE Computer Society.

[Fon06] José A. Rodríguez Fonollosa. Voice recognition software. http://gps-tsc.upc.es/veu/, 2006.

[GCO+04] Ruben Gonzalez, Adrian Cristal, Daniel Ortega, Alexander Veidenbaum, and Mateo Valero. A content aware integer register file organization. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 314, Washington, DC, USA, 2004. IEEE Computer Society.

[GCP+05] R. Gonzalez, A. Cristal, M. Pericas, M. Valero, and A. Veidenbaum. An asymmetric clustered processor based on value content. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 61–70, New York, NY, USA, 2005. ACM Press.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[GTM99] Antonio González, Jordi Tubella, and Carlos Molina. Trace-level reuse. In *International Conference on Parallel Processing*, 1999.

[Har80] Samuel Pollock Harbison. *A computer architecture for the dynamic optimization of high-level language programs.* PhD thesis, 1980.

[Har82] Samuel P. Harbison. An architectural alternative to optimizing compilers. In *ASPLOS-I: Proceedings of the first international symposium on Architectural support for programming languages and operating systems*, pages 57–65, New York, NY, USA, 1982. ACM Press.

[HL99] Jian Huang and David J. Lilja. Exploiting basic block value locality with block reuse. In *HPCA*, pages 106–114, 1999.

[Ho99] S. Hagiware and I. oliver. Sega dreamcast: Creating a unified entertainment world. November-December 1999.

[htt00] http://developer.intel.com/design/processor/index.htm. Willamette Architecture Software Developer Manuals. Technical report, Intel, 2000.

[Int00] 3DFX Interactive. Voodoo 3. *Web Page*, http://digilander.libero.it/F1Land/3dfxarchive/, 2000.

[JCV01] Roger Espasa Jesus Corbal and Mateo Valero. On the efficiency of reductions on micro-simd media extensions. In *PACT '01: IEEE Parallel Architectures and Compiler Techniques*. IEEE Computer Society, 2001.

[LPMS97] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[LS96] M.H. Lipasti and J.P. Shen. Exceeding the dataflow limit. pages 226–237, December 1996.

[MDM97] Mips extension for digital media with 3d. Technical Report http://www.mips.com, MIPS technologies, Inc., 1997.

[MMG+99] J. McCorman, R. McNamara, C. Gianos, N.P. Jouppi, T.Dutton, J. Zurawski, L. Seiler, and K. Correll. Implementing neon: A 256-bit graphics accelerator. March-April 1999.

[MMX98] Mmx technology programmers reference manual. Technical Report http://developer.intel.com/drg/mmx/manuals, INTEL corporation, 1998.

[Mob] Motorola razr v3i spec sheet. http:/-/www.mobiledia.com/phones/motorola/razr-v3i.html.

[Mot98] Inc. Motorola. Altivec Technology. Technical Report http://www.mot.com/SPS/PowerPC/AltiVec/, 1998.

[NJ99] Huy Nguyeni and Lizy Kurian John. Exploiting simd parallelism in dsp and multimedia algorithms using the altivec technology. *International Conference on Supercomputing*, 1999.

[Nvi01] Nvidia. Nvidia geforce. *Technical Report*, http://www.nvidia.com/, 2001.

[oCaB02] University of California at Berkeley. V-iram. http://iram.cs.berkeley.edu/, 2002.

[oCaBI95] University of California at Berkeley and ICSI. T0 vector microprocessor. http://www.icsi.berkeley.edu/real/spert/t0-intro.html, 1995.

[Pau97] B. Paul. The mesa 3-d graphics library. http://www.mesa3d.org, 1997.

[PDA06] Tomtom go 910 - features. *TomTom*, http://www.tomtom.com/products/features.php?ID=212&-Category=0&Lid=1, 2006.

[Pen06] Intel® pentium® m processor on 90 nm process with 2-mb l2 cache datasheet. *Intel Corporation*, http://www.intel.com/design/intarch/pentiumm/pentiumm-.htm?iid=ipp_embed+proc_pmp&, 2006.

[PMP06] Portalplayer pp5020 soc, ipod nano. *Portal Player*, http://www.portalplayer.com/products/documents/-5020_Brief_0108_Public.pdf, 2006.

[Pro06] The Lame Project. Lame. http://lame.sourceforge.net, 2006.

[PSP] Sony's psp specs released. http://www.geek.com/-news/geeknews/2003Jul/gee20030731021096.htm.

[PVP06] Archos portable video player 604 tech specs. *Archos*, http://www.archos.com/products/video/archos_604wifi/-tech_specs.html?country=global, 2006.

[PW96] A. Peleg and U. Weiser. Mmx technology extension to the intel architecture. *IEEE Micro*, pages 43–45, August 1996.

[QCEV99] Francisca Quintana, Jesus Corbal, Roger Espasa, and Mateo Valero. Adding a vector unit to a superscalar processor. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 1–10, New York, NY, USA, 1999. ACM Press.

[QCEV01] Francisca Quintana, Jesus Corbal, Roger Espasa, and Mateo Valero. A cost effective architecture for vectorizable numerical and multimedia applications. In *SPAA '01: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 103–112, New York, NY, USA, 2001. ACM Press.

[RAJ99] Parthasarathy Ranganathan, Sarita V. Adve, and Norman P. Jouppi. Performance of image and video processing with general-purpose processors and media ISA extensions. In *ISCA*, pages 124–135, 1999.

[RAJ00] Parthasarathy Ranganathan, Sarita V. Adve, and Norman P. Jouppi. Reconfigurable caches and their application to media processing. In *ISCA*, pages 214–224, 2000.

[Ric93] Stephen E. Richardson. Exploiting trivial and redundant computation. *11th IEEE Symposium on Computer Arithmetic*, pages 220–227, 1993.

[RS01] A. Roth and G.S. Sohi. Speculative data-driven multithreading. *Seventh International Symposium on High-Perfomance Computer Architecture*, 2001.

[SBS00] S. Sastry, R. Bodik, and J. Smith. Characterizing coarse-grained reuse of computation. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optmization, in conjuction with MICRO 33*, 2000.

[Sem99] Philips Semiconductors. Trimedia tm-1300. http://www-us3.semiconductors.com/trimedia/, 1999.

[SM98] J.G. Steffan and T.C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. *Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[SS97] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *ISCA*, pages 194–205, 1997.

[SV05] Esther Salami and Mateo Valero. A vector-usimd-vliw architecture for multimedia applications. In *ICPP, IEEE International Conference on Parallel Processing*, 2005.

[TI99] TI. TMS320C62XX family. Technical Report http://www.ti.com/sc/docs/products/dsp/tms320c6201.html, Texas Instruments, 1999.

[TONH96] M. Tremblay, J.M. O'Connor, V. Narayanan, and L. He. Vis speeds new media processing. *IEEE Micro*, August 1996.

[UMP06] Ultra mobile pc 2006 platform overview. *Intel Corporation*, http://www.intel.com/design/mobile/platform/downloads/umpc2006.pdf, 2006.

Uso del simulador SimpleReuse

Resumen

Este apéndice explica el funcionamiento del simulador SimpleReuse (una evolución de SimpleScalar y Wattch) para su uso en simulación de reuso de instrucciones o regiones. Este simulador realiza medidas de tiempo, instrucciones y consumo de potencia de códigos Alpha ejecutados en máquinas de muy distinta complejidad. El simulador no se encuentra disponible en la Web en este momento debido a que está en constante mejora, pero si hay alguien interesado en su uso puede pedirlo directamente a través de la dirección: carlos.alvarez@upc.edu.

A.1 Introducción

En los últimos años la proliferación de los sistemas de baja potencia (móviles, PDAs, etc.) ha generado una enorme demanda de capacidad de cálculo hacia estos. Los sistemas móviles de tercera generación se espera que sean capaces de ejecutar aplicaciones multimedia que hasta ahora estaban restringidas a los sistemas de propósito general de sobremesa. Dichas aplicaciones (codificadores y decodificadores MP3, reproducción de vídeo, reconocimiento de voz, etc.) poseen unos requerimientos de capacidad de cálculo muy superiores a los que actualmente es capaz de proporcionar un procesador de baja potencia. Por ello los sistemas se baja potencia han de enfrentarse al difícil reto de mejorar sus prestaciones de cálculo (incluyendo muchas veces el cálculo en punto flotante) manteniendo limitado su consumo.

En este ámbito, el cálculo difuso se propone como un sistema que permite obtener mayores velocidades de cálculo con bajo consumo a cambio de pequeñas perdidas de calidad en las señales finales generadas. Esta aproximación cobra sentido en el ámbito de las aplicaciones multimedia. Dichas aplicaciones se caracterizan por el hecho de que su destinatario final es una persona y por lo tanto la calidad de su salida depende de un criterio subjetivo. En este entorno pequeñas variaciones de los valores de salida no son percibidas y así pues podemos aprovechar esta circunstancia para conseguir beneficios adicionales. Un ejemplo claro de esto lo encontramos en los sistemas de compresión con perdidas como JPEG o MP3. En dichos sistemas, pequeñas perdidas de calidad en las señales se usan para conseguir mayor ratio de compresión. Las perdidas no sólo son aceptables sino muchas veces imperceptibles para el usuario mientras que las ganancias en cantidad de información almacenada son significativas.

Para poder medir los efectos de los sistemas de cálculo difuso en programas reales ha sido necesario "ejecutar" dichos programas en un sistema que implementara el cálculo difuso, para ello, se ha debido construir un simulador que, basado en una arquitectura real, añadiese a esta las características propias del sistema a estudiar. En este apéndice se explica el funcionamiento de este simulador que se ha denominado "SimpleReuse".

A.2 El simulador SimpleReuse.

El simulador SimpleReuse se ha elaborado a partir de los simuladores SimpleScalar[BA97] y Wattch[BTM00] como el simulador de trabajo de la tesis. Este simulador ha partido de la necesidad de elaborar estudios de la viabilidad de realizar

computación difusa en las aplicaciones multimedia. Para ello, los simuladores de origen, tenían las siguientes limitaciones:

- No realizaban detalladamente los cálculos del consumo de las unidades aritméticas.
- No permitían realizar cálculos a medida, es decir, todas las operaciones aritméticas se realizaban usando el procesador de la máquina que ejecutaba el simulador y, por tanto, no era posible estudiar sistemas alternativos de realizar las operaciones y evaluar sus resultados.
- No tenían integrado un sistema de configuración detallado. Es decir, su integración permitía estudiar la potencia consumida por una máquina tipo Alpha (la máquina principal que simula SimpleScalar) pero, así como SimpleScalar admite otras configuraciones, Wattch no admitía dichas configuraciones en su estructura y por lo tanto no permitía cálculos de consumo de energía basados en diferentes ficheros de configuración.
- No disponían de un sistema que permitiese realizar las operaciones aritméticas en diferentes unidades funcionales y medir sus efectos.
- No entraban en el detalle del consumo de las operaciones aritméticas de punto flotante.

Dadas todas estas limitaciones enumeradas, se decidió elaborar un nuevo simulador integrado que, partiendo del sistema integrado de SimpleScalar y Wattch, fuese operativo a todos estos niveles. El simulador resultante de bautizó como SimpleReuse.

A.3 Uso del simulador.

El simulador resultante se ejecuta a través del comando `sim-reuse`, que es una versión incrementada del programa `sim-outorder` de SimpleScalar, con Wattch integrado y todas las modificaciones realizadas. Como ya se ha comentado, Wattch ha sido totalmente integrado, de forma que no es necesario editar el simulador Wattch y recompilarlo para aceptar nuevas configuraciones, sino que directamente este simulador activa y desactiva los componentes de Wattch necesarios (por ejemplo, si no disponemos de memoria caché, el componente de consumo de `sim-reuse` no computa gasto de energía por la caché).

Así pues, para poder usar el simulador simplemente hemos de tener los ficheros de configuración adecuados para ello. Dichos ficheros son:

- El fichero de configuración de SimpleScalar. Dicho fichero se indica en línea de comandos mediante la opción `-config`. Sus opciones son las mismas que se indican en la versión original del programa, pero han sido integradas de forma que el consumo de potencia se adapta REALMENTE a lo que indica el fichero. En el apéndice A.6 se muestran varios ejemplos de estos ficheros. Todas las opciones de este fichero se podrían especificar también en línea de comandos, pero como son muchas esta posibilidad resulta incómoda.

- El resto de ficheros de configuración son específicos del sistema de reuso de instrucciones. El fichero principal (que tiene nombre único, no configurable) es el fichero `ISA_Gral.conf`. Si dicho fichero no existe el simulador genera un mensaje de error y acaba su ejecución. Este fichero contiene dos valores numéricos (habitualmente 0 o 1). El primero indica si está activado el reuso de regiones y el segundo si está activado el reuso de instrucciones. Si ambas opciones están desactivadas el simulador, simplemente, simulará un sistema sin reuso (pero seguirá contando bien el consumo de los elementos activados o desactivados mediante el fichero indicado en -config.

A.3.1 Reuso de regiones.

En la opción de reuso de regiones, los ficheros utilizados por el simulador son:

- En el reuso de regiones, el primer fichero de configuración, cuyo nombre por defecto es `ISA_bitr.conf` contiene el número de bits que se perderán en cada región reusada. Consta de una sola línea de texto que contiene un número entero para cada región distinta a reusar (conviene recordar que una región distinta implica una secuencia de operaciones distinta y por lo tanto diferentes regiones no pueden compartir resultados ni han de compartir necesariamente parámetros).

- El segundo es el fichero `ISA_tabr.conf`. Este fichero contiene 3 parámetros que indican, respectivamente, la cantidad de regiones distintas, el logaritmo en base 2 de la longitud de la tabla de regiones y la asociatividad de dicha tabla.

Las regiones a simular en este entorno se definen manualmente sobre el programa simulado. Para ello se han reinterpretado instrucciones del procesador de Alpha que se encontraban hasta ahora sin uso y que el compilador no utiliza. El código de los programas a estudiar debe ser modificado y recompilado usando estas instrucciones (que se pueden insertar en el código C mediante la directiva `asm` si se incluye la librería `c_asm.h`):

- `excb`. Esta instrucción activa y desactiva el simulador de reuso de regiones. Dentro de este modo el resto de instrucciones se comportan como se describe, fuera de él se comporta como lo harían normalmente (lo que en la mayoría de casos significa que no hacen nada ya que no se encontraban implementadas en el simulador).
- `wmb`. Esta instrucción genera un acceso a la tabla de reuso.
- `mb`. Esta instrucción inicializa las tablas de reuso de regiones.
- `bne`. Esta instrucción, dentro de la simulación del reuso de regiones, no salta cuando ha habido un acierto en la tabla. Sirve para implementar los dos caminos posibles que ocurren después de acceder a la tabla.
- `bisi`. Esta instrucción, dentro de la simulación del reuso de regiones, sirve para pasar los parámetros hacia la tabla de reuso de regiones. El primer valor es el parámetro de entrada y el inmediato es el número de bits a descartar de dicho parámetro. Si la tabla ya ha sido accedida, en cambio, lo que hace es asignar los resultados.
- `trapb`. Esta instrucción actualiza la tabla de regiones despues de un fallo.

Mediante estas instrucciones ensamblador es posible modificar cualquier código para crear una región reusable. El código se modifica de la siguiente manera. Imaginemos que las instrucciones que queremos sustituir son las siguientes:

```
if ((v = t0  - t1)<0)  v = -v; res = v;
if ((v = t2  - t3)<0)  v = -v; res += v;
```

El código que deberemos insertar es:

```
   asm("excb ");
      res=t0|1; res=t1|1;
```

```
      res=t2|1; res=t3|1;
      asm("wmb");
      if (res) goto seguirR1;
      res=1;
   asm("excb ");
   goto pruebaR1;
seguirR1:

if ((v = t0  - t1)<0)  v = -v; res = v;
if ((v = t2  - t3)<0)  v = -v; res += v;

   asm("excb ");
     res=res|1;
     asm("trapb");
   asm("excb ");
pruebaR1:
```

Como se puede ver el código insertado tiene dos partes, una previa al código a reusar y otra posterior. La parte previa inicializa la tabla, y realiza el acceso. La parte posterior actualiza la tabla en caso de que hubiera un fallo. Vamos a comentarlo:

La primera instrucción (`excb`) activa el modo de reuso de regiones del simulador. A continuación tantos OR (que se traducen como la orden `bisi`) como sean necesarios cargan en la tabla los parámetros de entrada. Fijémonos que en nuestro código, todo depende de los valores `t0` a `t3`, así que cada instrucción OR carga uno de estos parámetros de entrada en la tabla. El 1 (parámetro inmediato de la instrucción OR) dice que toleraremos 1 solo bit de cada parámetro, mientras que el destino de la operación no se usa para nada.

Una vez cargados los parámetros de acceso, la instrucción `wmb` realiza el acceso a la tabla y determina si ha habido un fallo o un acierto. La siguiente orden (el `if`), contra lo que pueda parecer, solo saltará si ha habido un fallo en el acceso (recordemos que la instrucción `bne` ha sido puenteada), así que si ha habido fallo se continuará a partir de la etiqueta `seguirR1`. Si ha habido un acceso a la tabla, las instrucciones siguientes al if asignan a las variables resultantes (en el ejemplo solo la variable `res`) el valor devuelto por la tabla (la instrucción `res=1` también queda codificada mediante la instrucción ensamblador `bisi`). En ese caso, después de realizada la asignación se desactiva el modo de simulación de regiones (siguiente instrucción `excb`) y se continúa la ejecución a partir de la etiqueta `pruebaR1`, de forma que se saltan las instrucciones de la región reusada.

Si no ha habido acierto en la tabla, la región a reusar se ejecuta normalmente y a continuación se vuelve a activar el modo de reuso de regiones para actualizar la tabla de reuso. Mediante la primera instrucción se carga el resultado (o resultados) en la tabla y mediante la segunda (`trapb`) se actualiza la última entrada accedida de la tabla. El último `excb` desactiva de nuevo el modo de reuso de regiones.

A.3.2 Reuso de instrucciones.

La simulación de reuso de instrucciones, al contrario que la de regiones, no precisa ninguna modificación de los programas originales. Estos simplemente, una vez compilados, se ejecutan bajo el simulador y este genera los distintos resultados según su configuración. Los resultados de prueba pueden obtenerse, o bien mediante la ejecución de los programas bajo la máquina original, bien bajo el simulador desactivando el sistema de reuso.

Los ficheros de configuración para el reuso de instrucciones son:

- `ISA_act.conf`. Este fichero (cuyo nombre se puede modificar mediante la opción `-f:act`) contiene una lista con un valor para cada posible instrucción en la que se puede activar el reuso. Dicho valor es verdadero o falso según si se reusa dicha instrucción o no. Las instrucciones que se pueden reusar son, por orden del fichero: Suma en doble precisión; Suma en simple precisión; Resta en doble precisión; Resta en simple precisión; Multiplicación en doble precisión; Multiplicación en simple precisión; División en doble precisión; División en simple precisión; Raíz cuadrada en doble precisión y Raíz cuadrada en simple precisión;

- `ISA_pos.conf`. Este fichero (también se puede modificar su nombre mediante `-f:pos`) contiene o bien una linea o bien tantas lineas como operaciones, con las reglas de acceso a la tabla de reuso para cada operación reusada. Si hay una sola linea, se supone que todas las operaciones tienen las mismas reglas de acceso. Cada linea debe contener 7 cifras. Las dos primeras contienen la mascara que permite decidir cuantos bits de cada operando hay que usar para acceder a la tabla. Por ej. si queremos usar 9 bits, la máscara será 1FF. La regla de acceso es, hacer la AND, desplazar a la derecha o a la izquierda los bits enmascarados (habitualmente no se desplazan) y a continuación hacer la XOR de ambos operandos junto con el último valor de la regla. (El acceso normal, para una tabla de 1K entradas, y asociatividad 2, es una regla "1FF 1FF 0 0 0 0 0").

- `ISA_acc.conf`. (Se puede modificar su nombre mediante `-f:acc`) Este fichero contiene una linea con 20 números enteros, pero es obsoleto y por tanto no tiene ningún efecto.

- `ISA_con.conf`. (Se puede modificar su nombre mediante `-f:con`) Fichero que indica si se debe aplicar la propiedad conmutativa a las tablas de reuso. Contiene una secuencia de 10 valores booleanos (1 por operación). Si dicho valor es verdadero, antes de acceder a la tabla de reuso el sistema ordenará los operandos de forma que valores iguales en distinto orden en una operación conmutativa no ocupen dos entradas en la tabla. Es una opción de configuración debido a que el hardware que realiza la operación no es trivial y en muchos casos es mejor no realizar esta comprobación ya que hay muy poca ganancia.

- `ISA_tol.conf`. (Se puede modificar su nombre mediante `-f:tol`) Fichero que contiene para cada operación (diez entradas) un entero que indica los bits tolerados al reusar dicha operación (OJO, el simulador no comprueba que este valor sea razonable o válido).

- `ISA_tot.conf`. (Se puede modificar su nombre mediante `-f:tot`) Fichero que contiene, para cada operación, la tolerancia en el caso de sumas o restas triviales (en los demás casos no se aplica su contenido). Si la configuración es de detectar de forma avanzada operaciones triviales (ver fichero `ISA_var` más adelante), este fichero determina que tolerancia hay que aplicar a una suma antes de hacerla y dar por valido uno de los operandos. Esto tiene sentido desde el punto de vista de que una suma tolerante de dos operandos donde uno es mucho mayor que otro da como resultado el propio número mayor.

- `ISA_tab.conf`. (Se puede modificar su nombre mediante `-f:tab`) Este fichero contiene el número de tablas de reuso a utilizar y a que operaciones se asignan. Para ello contiene 10 enteros (uno por operación) que especifican a que tabla asignar cada operación. Si dos o más operaciones comparten una misma tabla el simulador realizará los cálculos consecuentemente.

- `ISA_cap.conf`. (Se puede modificar su nombre mediante `-f:cap`) Este fichero contiene para cada operación cuantos bits menos significativos de los operandos deben eliminarse antes de realizar la operación. Es importante notar que esto no es lo mismo que tolerar dichos bits, ya que si los bits se eliminan la operación será mal realizada aunque no se acierte en la tabla o no haya tabla. Esta opción permite simular unidades funcionales más cortas y ver los resultados.

- `ISA_var.conf`. (Se puede modificar su nombre mediante `-f:var`) Fichero de configuraciones varias. Contiene 8 enteros, cada uno con

un significado específico:

1. Asociatividad de la tabla. Este parámetro y la máscara del fichero ISA_pos, determinan el número de entradas de la tabla.
2. Tipo de tabla usada. Admite tres valores: 0 para tablas finitas (reales), 1 para simular tablas infinitas y 2 para simular tablas finitas pero que solo memorizan las mantisas de las operaciones y calculan los exponentes.
3. Detección de las operaciones triviales: 0 no activada, 1 activada y 2 detección avanzada, es decir, evita hacer sumas cuando los operandos son muy distantes.
4. Tabla en acceso paralelo a la unidad aritmética (1) o secuencial con ella (0). En el primer caso siempre hay consumo de potencia en la unidad aritmética, mientras que en el segundo el tiempo de cálculo en caso de fallo en la tabla aumenta.
5. Activa el sistema de reuso tolerante consistente en guardar la media del resultado en lugar del resultado completo en la tabla. Como consecuencia la tabla de reuso es más estrecha. Con un valor falso no activa este sistema, mientras que un valor positivo indica cuantos bits adicionales (a los de los operandos) hay que guardar en el resultado.
6. Activa o desactiva la tolerancia de las operaciones de comparación.
7. Activa o desactiva el filtrado de operaciones. Con el filtrado activado hay que tener en cuenta otros ficheros.
8. Activa o desactiva el límite de reuso. Si el valor es un número positivo, además indica el número máximo de veces a reusar una entrada de la tabla antes de borrarla. Esta opción es útil para intentar limitar el error recurrente introducido por una misma operación.

Filtrado de operaciones.

Otra de las características del simulador es que permite filtrar operaciones. El filtrado de operaciones permite experimentar el reuso de instrucciones con profiling o dinámico. En el filtrado dinámico, la tabla de reuso trata de adaptarse al comportamiento del programa en tiempo de ejecución, mientras que con profiling, se realizan dos pasadas, en la primera se decide que instrucciones son útiles para reusar sobre el procesado de una imagen de muestra y en la segunda pasada, que ya simula una ejecución real, se utiliza el resultado de la primera pasada para intentar mejorar los resultados de reuso de otra imagen distinta.

Para poder realizar este filtrado, el uso es el siguiente: existen 4 posibles valores para la opción de filtrado (un 0, falso, significa que no se realiza filtrado de operaciones). Si la opción está activa, los posibles valores van del 1 al 4. Cada valor representa una fase del filtrado:

1. Con este tipo de filtro activado, solo ciertas instrucciones acceden a la tabla de reuso, realizándose un filtrado dinámico de instrucciones. Se define una tabla de filtro de tamaño la constante `TamFiltro` y la primera vez que una instrucción accede a la tabla se le asigna un valor de confianza (`MAXFIL`). Si una instrucción falla en la tabla de reuso `MAXFIL` veces seguidas no vuelve a acceder a la tabla, es decir, es un filtro que intenta eliminar las instrucciones que fallan siempre.

2. Este filtro, en cambio, establece un intervalo de confianza mediante un contador saturado de tamaño $\log_2$`MAXFIL`. El número de entradas del filtro también lo establece constante `TamFiltro`.

3. En esta opción, al igual que la siguiente, se utiliza para el filtrado con profiling. En este paso del proceso, concretamente, se realiza el filtrado usando el contenido del fichero `ISAFILTRO.conf`. Aquellas instrucciones cuya dirección no esté contenido en este fichero no accederán a la tabla de reuso.

4. Esta opción imprime una estadística indicando que instrucción acaba de ejecutarse y si ha acertado o no. Un programa de ayuda, `filtrar.c` (ver anexo A.5) recoge esta salida y la transforma en un fichero que contiene para cada dirección de instrucción, la cantidad de aciertos y de accesos a las tablas de la instrucción. El fichero resultante puede ser procesado por cualquier método para conseguir un fichero con solo aquellas direcciones que contienen un porcentaje de aciertos aceptable. Dicho fichero deberá llamarse `ISAFILTRO.conf` y se usará en una posterior ejecución con la opción 3 de filtrado.

A.4 Conclusiones y extensiones futuras

El simulador SimpleReuse es un simulador basado en Simplescalar y Wattch que añade a estos simuladores 3 funcionalidades básicas:

1. Permite su configuración conjunta mediante un solo juego de ficheros de configuración y elimina la necesidad de recompilar los simuladores.

2. Permite la simulación de la capacidad de la CPU de reuso de instrucciones.

3. Permite la simulación de un sistema hardware de reuso de regiones.

El simulador puede ser todavía mejorado, incorporando principalmente dos líneas de análisis:

1. La capacidad de analizar, en tiempo real, regiones de instrucciones susceptibles de ser reutilizadas.
2. La incorporación de sistemas dinámicos de control del error.

A.5 Código de filtrar.c

```
#include <stdio.h>

#define DESBALANCEO 5

typedef struct {
   long ins;
   long hits,accs;
   int amayor,amenor;
   void *mayor,*menor;
} elemento_arbol;

typedef elemento_arbol* ptr_elemento_arbol;

ptr_elemento_arbol inicio=NULL;
long ins=0;
int hit;

void ponerins(ptr_elemento_arbol punt, ptr_elemento_arbol ant)
{
   ptr_elemento_arbol temp;

   if (punt==NULL) {
      printf("Error, mal programado\n");
      exit();
   } else {
      if ((punt->ins)==ins) {
         punt->accs++;
         punt->hits+=hit;
      } else {
```

```
if (ins>(punt->ins)) {
   if (punt->mayor==NULL) {
      punt->mayor=malloc(sizeof(elemento_arbol));
      if (punt->mayor==NULL) {
         printf("Error, memoria agotada\n");
         exit();
      }
      (punt->mayor)->ins=ins;
      (punt->mayor)->accs=1;
      (punt->mayor)->hits=hit;
      (punt->mayor)->amayor=0;
      (punt->mayor)->amenor=0;
      (punt->mayor)->mayor=NULL;
      (punt->mayor)->menor=NULL;
   } else {
      if (((punt->amayor)-(punt->amenor))>DESBALANCEO) {
         temp=punt->mayor;
         punt->mayor=temp->menor;
         temp->menor=punt;
         if (ant!=punt) {
            if (ant->mayor==punt)
               ant->mayor=temp;
            else
               ant->menor=temp;
         }
         punt->amayor=0;
         punt->amenor=0;
         if (inicio==punt) {
            inicio=temp;
            ant=temp;
         }
         ponerins(temp,ant);
      } else {
         punt->amayor++;
         ponerins(punt->mayor,punt);
      }
   }
} else {
   if (punt->menor==NULL) {
      punt->menor=malloc(sizeof(elemento_arbol));
      if (punt->menor==NULL) {
         printf("Error, memoria agotada\n");
         exit();
      }
```

```
            (punt->menor)->ins=ins;
            (punt->menor)->accs=1;
            (punt->menor)->hits=hit;
            (punt->menor)->amayor=0;
            (punt->menor)->amenor=0;
            (punt->menor)->mayor=NULL;
            (punt->menor)->mayor=NULL;
         } else {
            if (((punt->amenor)-(punt->amayor))>DESBALANCEO) {
               temp=punt->menor;
               punt->menor=temp->mayor;
               temp->mayor=punt;
               if (ant!=punt) {
                  if (ant->mayor==punt)
                     ant->mayor=temp;
                  else
                     ant->menor=temp;
               }
               punt->amayor=0;
               punt->amenor=0;
               if (inicio==punt) {
                  inicio=temp;
                  ant=temp;
               }
               ponerins(temp,ant);
            } else {
               punt->amenor++;
               ponerins(punt->menor,punt);
            }
         }
      }
   }
}
}

void Imprimir(ptr_elemento_arbol punt)
{
   if (punt->menor!=NULL) {
      Imprimir(punt->menor);
   }
   if (punt->mayor!=NULL) {
      Imprimir(punt->mayor);
   }
   printf("%ld %d %d\n",punt->ins,punt->hits,punt->accs);
```

```
}

void main()
{
   while (ins!=-1) {
      scanf("Instruccion: %ld %d\n",&ins,&hit);
      if (ins!=-1)
         if (inicio==NULL) {
            inicio=malloc(sizeof(elemento_arbol));
            inicio->ins=ins;
            inicio->hits=hit;
            inicio->accs=1;
            inicio->amayor=0;
            inicio->amenor=0;
            inicio->mayor=NULL;
            inicio->menor=NULL;
         } else {
            ponerins(inicio,inicio);
         }
   }

   if (inicio!=NULL)
      Imprimir(inicio);
   else
      printf("VACIO!!!!!!!!!!!!!\n");
}
```

A.6 Ejemplos de ficheros de configuración.

A.6.1 Configuración para procesador de ancho 1 en orden.

```
# random number generator seed (0 for timer seed)
-seed                            1

# instruction fetch queue size (in insts)
-fetch:ifqsize                   1

# extra branch mis-prediction latency
-fetch:mplat                     1

# speed of front-end of machine relative to execution core
```

```
-fetch:speed   1

# branch predictor type {nottaken|taken|perfect|bimod|2lev|comb}
-bpred                       nottaken

# instruction decode B/W (insts/cycle)
-decode:width                    1

# instruction issue B/W (insts/cycle)
-issue:width                     1

# run pipeline with in-order issue
-issue:inorder               true

# issue instructions down wrong execution paths
-issue:wrongpath              true

# instruction commit B/W (insts/cycle)
-commit:width   1

# register update unit (RUU) size
-ruu:size                       4

# load/store queue (LSQ) size
-lsq:size                        2

# perfect memory disambiguation
#-lsq:perfect false

# l1 data cache config, i.e., {<config>|none}
-cache:dl1            dl1:128:32:4:f

# l1 data cache hit latency (in cycles)
-cache:dl1lat                    1

# l2 data cache config, i.e., {<config>|none}
-cache:dl2            none

# l1 inst cache config, i.e., {<config>|dl1|dl2|none}
-cache:il1            il1:512:32:1:f

# l1 instruction cache hit latency (in cycles)
-cache:il1lat                    1
```

```
# l2 instruction cache config, i.e., {<config>|dl2|none}
-cache:il2                    none

# flush caches on system calls
-cache:flush                  false

# convert 64-bit inst addresses to 32-bit inst equivalents
-cache:icompress              false

# memory access latency (<first_chunk> <inter_chunk>)
-mem:lat               64 1

# memory access bus width (in bytes)
-mem:width                        4

# memory accesses are fully pipelined
#-mem:pipelined          false

# instruction TLB config, i.e., {<config>|none}
-tlb:itlb              itlb:16:4096:4:f

# data TLB config, i.e., {<config>|none}
-tlb:dtlb              dtlb:32:4096:4:f

# inst/data TLB miss latency (in cycles)
-tlb:lat                          30

# total number of integer ALU's available
-res:ialu                         1

# total number of integer multiplier/dividers available
-res:imult                        1

# total number of memory system ports available (to CPU)
-res:memport                      1

# total number of floating point ALU's available
-res:fpalu                        1

# total number of floating point multiplier/dividers available
-res:fpmult                       1

# operate in backward-compatible bugs mode (for testing only)
-bugcompat                    false
```

```
# latencia de las operaciones:
-lat:IALU 1
-iss:IALU 1
-lat:IMUL 4
-iss:IMUL 1
-lat:IDIV 20
-iss:IDIV 19
```

A.6.2 Configuración para procesador de ancho 2 en orden.

```
# random number generator seed (0 for timer seed)
-seed                           1

# instruction fetch queue size (in insts)
-fetch:ifqsize                  2

# extra branch mis-prediction latency
-fetch:mplat                    2

# speed of front-end of machine relative to execution core
-fetch:speed   1

# branch predictor type {nottaken|taken|perfect|bimod|2lev}
-bpred                      2lev

# instruction decode B/W (insts/cycle)
-decode:width                   2

# instruction issue B/W (insts/cycle)
-issue:width                    2

# run pipeline with in-order issue
-issue:inorder              true

# issue instructions down wrong execution paths
-issue:wrongpath             true

# instruction commit B/W (insts/cycle)
-commit:width   2

# register update unit (RUU) size
```

```
-ruu:size                       8

# load/store queue (LSQ) size
-lsq:size                       4

# l1 data cache config, i.e., {<config>|none}
-cache:dl1             dl1:128:32:4:f

# l1 data cache hit latency (in cycles)
-cache:dl1lat                   1

# l2 data cache config, i.e., {<config>|none}
-cache:dl2             ul2:256:64:4:l

# l1 inst cache config, i.e., {<config>|dl1|dl2|none}
-cache:il1             il1:512:32:1:f

# l1 instruction cache hit latency (in cycles)
-cache:il1lat                   1

# l2 instruction cache config, i.e., {<config>|dl2|none}
-cache:il2                     dl2

# flush caches on system calls
-cache:flush                 false

# convert 64-bit inst addresses to 32-bit inst equivalents
-cache:icompress             false

# memory access latency (<first_chunk> <inter_chunk>)
-mem:lat               64 1

# memory access bus width (in bytes)
-mem:width                      8

# instruction TLB config, i.e., {<config>|none}
-tlb:itlb              itlb:16:4096:4:f

# data TLB config, i.e., {<config>|none}
-tlb:dtlb              dtlb:32:4096:4:f

# inst/data TLB miss latency (in cycles)
-tlb:lat                       30
```

```
# total number of integer ALU's available
-res:ialu                       2

# total number of integer multiplier/dividers available
-res:imult                      1

# total number of memory system ports available (to CPU)
-res:memport                    2

# total number of floating point ALU's available
-res:fpalu                      1

# total number of floating point multiplier/dividers available
-res:fpmult                     1

# operate in backward-compatible bugs mode (for testing only)
-bugcompat                  false

# latencia de las operaciones:
-lat:IALU 1
-iss:IALU 1
-lat:IMUL 3
-iss:IMUL 1
-lat:IDIV 20
-iss:IDIV 19
```

A.6.3 Configuración para procesador de ancho 4 fuera de orden.

```
# random number generator seed (0 for timer seed)
-seed                           1

# instruction fetch queue size (in insts)
-fetch:ifqsize                  4

# extra branch mis-prediction latency
-fetch:mplat                    3

# branch predictor type {nottaken|taken|perfect|bimod|2lev}
-bpred                      comb

# bimodal predictor BTB size
```

```
-bpred:bimod                  2048

# 2-level predictor config (<l1size> <l2size> <hist_size>)
-bpred:2lev           1 1024 8 0

# instruction decode B/W (insts/cycle)
-decode:width                    4

# instruction issue B/W (insts/cycle)
-issue:width                     4

# run pipeline with in-order issue
-issue:inorder               false

# issue instructions down wrong execution paths
-issue:wrongpath              true

# register update unit (RUU) size
-ruu:size                       16

# load/store queue (LSQ) size
-lsq:size                        8

# l1 data cache config, i.e., {<config>|none}
-cache:dl1            dl1:128:32:4:l

# l1 data cache hit latency (in cycles)
-cache:dl1lat                    1

# l2 data cache config, i.e., {<config>|none}
-cache:dl2            ul2:1024:64:4:l

# l2 data cache hit latency (in cycles)
-cache:dl2lat                    6

# l1 inst cache config, i.e., {<config>|dl1|dl2|none}
-cache:il1            il1:512:32:1:l

# l1 instruction cache hit latency (in cycles)
-cache:il1lat                    1

# l2 instruction cache config, i.e., {<config>|dl2|none}
-cache:il2                     dl2
```

```
# flush caches on system calls
-cache:flush                 false

# convert 64-bit inst addresses to 32-bit inst equivalents
-cache:icompress             false

# memory access latency (<first_chunk> <inter_chunk>)
-mem:lat               64 1

# memory access bus width (in bytes)
-mem:width                       8

# instruction TLB config, i.e., {<config>|none}
-tlb:itlb              itlb:16:4096:4:l

# data TLB config, i.e., {<config>|none}
-tlb:dtlb              dtlb:32:4096:4:l

# inst/data TLB miss latency (in cycles)
-tlb:lat                        30

# total number of integer ALU's available
-res:ialu                        4

# total number of integer multiplier/dividers available
-res:imult                       1

# total number of memory system ports available (to CPU)
-res:memport                     2

# total number of floating point ALU's available
-res:fpalu                       4

# total number of floating point multiplier/dividers available
-res:fpmult                      1

# operate in backward-compatible bugs mode (for testing only)
-bugcompat                   false

# latencia de las operaciones:
-lat:IALU 1
-iss:IALU 1
-lat:IMUL 3
-iss:IMUL 1
```

```
-lat:IDIV 20
-iss:IDIV 19
```

Lista de Figuras

2.1 Gráfico de consumos por cm^2 de procesadores reales. 11

3.1 La percepción es traicionera. 18
3.2 Base de la propuesta de la tesis. 19
3.3 Dos imágenes para comparar, originales (a, c) y con errores (b, d). 21
3.4 Dos nuevas imágenes de la chica de la pamela, ambas con errores. 23
3.5 Diagrama de bloques del procesador fuera de orden. 27

4.1 Esquema del reuso difuso de instrucciones. 35
4.2 Multiplicación en formato IEEE 754. 38
4.3 SNR mínima según los bits de mantisa de un número real. . . 38
4.4 SNR al introducir unidades funcionales difusas. 41
4.5 Dos sonidos idénticos desfasados: SNR=0 dB. 41
4.6 SNR del codificador Lame con unidades funcionales difusas, corrigiendo los desfases. 42
4.7 Resultados de utilizar unidades funcionales difusas en diferentes procesadores. 43
4.8 Esquema de una tabla de reuso secuencial. 46
4.9 Tamaño de la tabla de reuso frente a la ALU de FP. 48
4.10 Consumo de la tabla de reuso frente a la ALU de FP. 49
4.11 Resultados de la memorización clásica. 50
4.12 Estructura del sistema hardware de reuso tolerante. 52
4.13 Mecanismo de relleno del resultado almacenado en la tabla tolerante. 52

4.14 Error introducido vs. aciertos en las tablas de reuso. 54
4.15 Distribución de los puntos reusados. 55
4.16 Ahorros de energía en la unidad de coma flotante. 56
4.17 Ahorros de tiempo y energía obtenidos mediante reuso tolerante. 57
4.18 Configuración en Serie vs. configuración en Paralelo. 58
4.19 Procesador de ancho 1. 59
4.20 Procesador de ancho 2. 60
4.21 Procesador de ancho 4 fuera de orden. 60
4.22 Ganancias en el procesador de ancho 1. 62
4.23 Ganancias en el procesador de ancho 2. 63
4.24 Ganancias en el procesador fuera de orden de ancho 4. 64

5.1 Etapas del codificador JPEG. 72
5.2 Mecanismo propuesto para reuso convencional de regiones. . . 74
5.3 Las tres imágenes procesadas mediante JPEG. 82
5.4 Porcentaje de aciertos de los algoritmos estudiados con tablas finitas. 83
5.5 Resultados del reuso de regiones en un procesador de ancho 1. 85
5.6 Resultados del reuso de regiones en un procesador de ancho 2. 86
5.7 Resultados del reuso de regiones en un procesador fuera de orden de ancho 4. 87
5.8 Tabla hardware para el reuso tolerante de regiones. 89
5.9 Porcentaje de aciertos en las tablas cuando se incrementa el grado de tolerancia en el programa *cjpeg*. 92
5.10 Porcentaje de aciertos en cada región con diferentes tolerancias. 94
5.11 SNR según la tolerancia para las diferentes regiones estudiadas. 95
5.12 SNR con varias regiones toleradas de las aplicaciones JPEG y H263. 97
5.13 Resultados del reuso tolerante de regiones en el programa codificador de vídeo. 99
5.14 Diferentes calidades de la imagen *specmun*. 100
5.15 Diferentes calidades de vídeo. 102
5.16 Resultados del reuso tolerante de regiones en un procesador de ancho 1. 104
5.17 Resultados del reuso tolerante de regiones en un procesador de ancho 2. 105
5.18 Resultados del reuso tolerante de regiones en un procesador fuera de orden de ancho 4. 106
5.19 Peores resultados de calidad con ajuste dinámico agresivo de la tolerancia y tabla de 64 KB. 116
5.20 Resultados del reuso tolerante dinámico de regiones en un procesador de ancho 1. 117

5.21 Resultados del reuso tolerante dinámico de regiones en un procesador de ancho 2. 118
5.22 Resultados del reuso tolerante dinámico de regiones en un procesador fuera de orden de ancho 4. 119

Lista de Tablas

3.1 Diferentes medidas de calidad para las figuras 3.3 y 3.4 24
3.2 Configuración de los procesadores de referencia. 28

4.1 Formatos definidos en el IEEE 754. 36
4.2 Programas de prueba utilizados. 37
4.3 Instrucciones Triviales. 47
4.4 Variables evaluadas para las tablas de reuso. 47
4.5 Características óptimas de las tablas de reuso. 47
4.6 Porcentajes de aciertos del reuso clásico. 50
4.7 Calidades subjetivas y niveles de tolerancia para diferentes aplicaciones. 56

5.1 Programas de prueba utilizados. 71
5.2 Características de los datos de prueba. 72
5.3 Instrucciones ensamblador para reuso de regiones. 78
5.4 Regiones seleccionadas para reuso de regiones. 79
5.5 Reuso potencial con tablas infinitas en las diferentes etapas de la aplicación JPEG. 81
5.6 Tolerancias y calidades en JPEG y H263. 101
5.7 Resultados de tolerancias ajustadas dinámicamente (umbral de subida=6). 114
5.8 Resultados de tolerancias ajustadas dinámicamente (umbral de subida=5). 115