Máster en Computación

Facultad de Ciencias – Universidad de Cantabria

Proyecto Fin de Máster

Microarquitecturas Dinámicas con Apilamiento 3D
(3DDM: 3D Stacked Dynamic Micro-architecture)

Autor: Adrián Colaso Diego

Tutor: Valentín Puente Varona

Grupo: Arquitectura y Tecnología de Computadores

Departamento: Electrónica y Computadores

Julio – 2012

A mi abuelo Cristóbal

Índice

Índice de figuras

Índice de tablas

Capítulo I
Introducción

Este primer capítulo introductorio expone el contexto, la motivación y los objetivos de este Proyecto Fin de Máster así como la organización del documento.

1.1 Contexto

Durante décadas el método para incrementar el rendimiento de los procesadores consistía en añadir más y más transistores, gracias a las mejoras en la tecnología, que permitía paulatinamente ir reduciendo su tamaño [1], para así incrementar su funcionalidad y ganar en rendimiento.

Esta tendencia alcanzó su límite, en parte, debido al Power-Wall [2] y al ILP-Wall [2]. El primer problema viene dado por el aumento de la frecuencia del reloj que provoca una mayor disipación de energía, mientras que el segundo radica en la dificultad de encontrar suficientes instrucciones como para mantener ocupado al procesador y obtener un buen rendimiento. Estas dificultades han forzado un cambio drástico en la tendencia, los chips han pasado a estar formados por dos o más cores idénticos trabajando en paralelo.

A partir de entonces, la mejora en el rendimiento dejó de ser resultado del incremento de la frecuencia del reloj, la cual se estabilizó, pasando a ser el resultado del paralelismo. Para un conjunto de aplicaciones que se ejecutan en un chip *multicore*, la mejora en el rendimiento se debe al balanceo de las mismas entre los distintos *cores*, mientras que para una determinada aplicación viene dada por el porcentaje de código paralelizable que se puede repartir entre los distintos *cores*.

Podemos ir incluso más allá con el 3D stacking [3], lo que nos permitirá incrementar de manera significativa el número de transistores por chip sin incurrir en el coste de diseñar e implementar mecanismos de fabricación para escalarlos y aumentar la densidad de integración. La dimensión adicional proporcionada atenuará el coste de la reducción continua del tamaño del transistor que viene produciéndose.

La madurez de los simuladores es otro aspecto destacable, nos permiten modelar un sistema real de un modo muy detallado sin llegar a abarcar la totalidad de las características del sistema real pero acercándose en grado suficiente como para que los resultados obtenidos puedan considerarse completamente válidos.

Sin estos simuladores, avanzar e innovar en la arquitectura de computadores se convertiría en una ardua tarea, puesto que una nueva idea o un cambio en un diseño anterior implicaría la construcción de un nuevo chip físico, lo cual resultaría terriblemente caro e ineficiente en términos temporales. Los simuladores nos permiten cambiar los distintos componentes micro-arquitecturales de una manera relativamente sencilla, ya sea modificando el número de *cores* en el chip, aumentando o disminuyendo el tamaño y los niveles de *cache*, cambiando las latencias,… permitiendo experimentar con una infinidad de configuraciones diferentes.

La gran desventaja viene dada por el tiempo requerido por las simulaciones, más aún cuando paulatinamente se van incorporando más y más *cores*, ya que los simuladores son secuenciales. Estaríamos hablando de que las herramientas más apropiadas en la innovación de los

procesadores, actualmente chips *multicore*, no se benefician de los avances ya que son difícilmente paralelizables y se ven obligadas a ejecutarse de un modo secuencial.

1.2 Motivación

Como se ha introducido en el apartado anterior, los procesadores *multicore* se han establecido en multitud de sistemas informáticos desde los computadores domésticos hasta los supercomputadores, incluso hay teléfonos móviles con chips con múltiples núcleos.

Dentro de estos chips, cada uno de los *cores* en los que está dividido tiene sus propios recursos como son los registros, la *cache* de nivel 1 formada por datos e instrucciones, las unidades aritmético-lógicas, etc. Los *cores* son simétricos, pero el uso de los recursos por parte de cada *core* potencialmente no es así, es decir, se producen asimetrías en el uso de los recursos micro-arquitecturales.

Cuando se ejecuta una aplicación *multithreaded*, las tasas de utilización de los recursos micro-arquitecturales, potencialmente, pierden su homogeneidad apareciendo asimetrías en el uso de los mismos. En determinados momentos, el rendimiento de ciertos *cores* puede caer debido a que deben sincronizarse con otros *cores*, a que las cargas de trabajo están mal balanceadas, a comportamientos asimétricos en memoria,.. y como consecuencia, los recursos se infrautilizan; es aquí, cuando entran en juego los procesadores con recursos micro-arquitecturales dinámicos [4][5][6].

Las *cores* que componen estos procesadores comparten, de manera dinámica, determinados recursos micro-arquitecturales, como las unidades funcionales o las vías del *issue/retire*. Cuando los *cores* trabajan de un modo normal los recursos se encuentran repartidos de manera equilibrada pero ante una asimetría, es decir, ante un fallo en la *cache* de instrucciones o un salto mal especulado, los recursos libres se pueden repartir entre el resto de *cores* para así aprovechar esos recursos que momentáneamente quedan libres y aumentar la eficiencia en el uso de los recursos micro-arquitecturales.

Ante este planteamiento, nos encontramos con que un *core* puede utilizar recursos que pertenezcan físicamente a otro *core* y aquí es donde el *3D stacking* muestra un gran potencial con respecto a los chips en dos dimensiones. Este potencial radica en la latencia de las comunicaciones en la tercera dimensión, puesto que es menor que en las otras dos dimensiones y el ancho banda disponible es mucho mayor [7], de este modo, el impacto del retardo propiciado por el uso de recursos que no se encuentran físicamente dentro del *core* no afecta al tiempo de ciclo.

Teniendo en cuenta lo anterior, un procesador en tres dimensiones, con microarquitectura dinámica, permitirá un mejor aprovechamiento de los recursos, que se manifestará en un mayor rendimiento y además de manera totalmente transparente para el programador.

1.3 Objetivos

Este proyecto tiene un único objetivo y es el estudio de los potenciales beneficios de un procesador en 3D con recursos micro-arquitecturales dinámicos. Podemos dividir este estudio en dos partes:

- Modificar el *back-end* de un *CMP* para que utilice una microarquitectura dinámica.

- Evaluar los potenciales beneficios a través de la simulación de un gran conjunto de cargas de trabajo de diferente naturaleza.

1.4 Organización del Documento

Tras este capítulo introductorio, el resto del documento se estructura como sigue:

Capítulo 2: Herramientas utilizadas. Describe la estructura de simuladores que se ha utilizado para modelar un sistema real, así como el conjunto de aplicaciones representativas utilizadas en las simulaciones.

Capítulo 3: *3DDM*. Explica las características y el funcionamiento de un chip en 3D con microarquitectura dinámica.

Capítulo 4: Evaluación. Evalúa la implementación con multitud de cargas de trabajo y analiza los resultados.

Capítulo 5: Conclusiones. Se presentan las conclusiones del proyecto

Capítulo 6: Trabajos futuros. Se analizan las posibles líneas futuras de continuación con el presente proyecto.

Capítulo II
Herramientas utilizadas

Este capítulo describe las herramientas utilizadas para la elaboración del proyecto.

2.1 Entorno de simulación

El entorno de simulación de sistema completo, el cual permite simular computadores con un nivel de detalle tan alto que es posible ejecutar sobre él un sistema operativo comercial, junto con sus aplicaciones, sin ninguna modificación adicional, está compuesto por varias piezas que se comunican entre sí. Por un parte, tenemos *Simics*, un simulador de sistema completo sobre el cual se cargan los módulos *Opal* y *Ruby* pertenecientes a *GEMS* [8].

Los módulos se comunican con *Simics* y entre sí. La comunicación entre los módulos y *Simics* se realiza a través de la *application programming interface* (*API*) que este último proporciona, mientras que la comunicación entre los distintos módulos se basa en interfaces que se registran en *Simics*, las cuales, los módulos pueden descargarse.

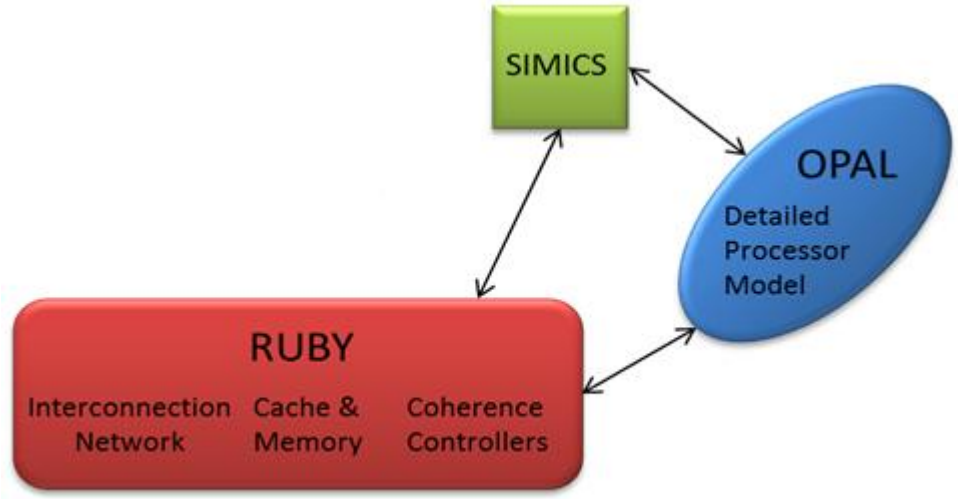

Figura 1 – Detalle de los simuladores.

2.2 *Simics*

Simics [9] es un simulador funcional del sistema completo desarrollado por la empresa Virtutech, adquirida recientemente por Intel. Permite ejecutar un sistema operativo junto con sus aplicaciones sobre el sistema simulado de manera transparente, es decir, sin requerir ninguna modificación.

Simics soporta un total de 9 repertorios de instrucciones: Sparc, x86, AMD x86-64, Alpha, PowerPC, IA-64, ARM y MIPS.

2.3 *GEMS*

GEMS [8], *General Execution-driven Multiprocessor Simulator*, desarrollado por la Universidad de *Wisconsin*, está compuesto, principalmente, por dos módulos que complementan

la simulación funcional de *Simics* incorporando modelos arquitecturalmente precisos de sistemas multiprocesador, incluyendo chips multiprocesador (*CMPs*).

GEMS solventa la limitación de *Simics* en cuanto a los modelos de temporización (*timing models*), que incorpora como extensiones, permitiendo la simulación temporal precisa para procesador y memoria de forma mucho más flexible.

2.3.1 *Opal*

Opal es el encargado de simular el modelo de temporización del procesador. Simula las distintas etapas del *pipeline*, modelando un procesador con ejecución fuera de orden así como el acceso especulativo a la memoria. Es el encargado de controlar el número de ciclos que requiere la ejecución de cada instrucción. Aunque dispone de un simulador de jerarquía de memoria sencillo, emplea *Ruby* para determinar cuántos ciclos invierte en memoria cada instrucción, tanto en *fetch* como en acceso a datos.

Permite una configuración detallada de los distintos aspectos del procesador y emplea una arquitectura acorde con el estado del arte.

2.3.2 *Ruby*

Ruby es un simulador temporal de un sistema de memoria multiprocesador. Modela las *caches* junto con sus controladores, los sistemas de interconexión, los controladores de memoria y la memoria principal.

Permite modelar distintas jerarquías de memoria, fundamentalmente orientado hacia el desarrollo y verificación de protocolos de coherencia, etc.

2.4 Cargas de Trabajo

A continuación, se enumeran cada uno de los cuatro conjuntos de aplicaciones. Éstos conjuntos de aplicaciones están compuestos por *benchmarks* estandarizados, aplicaciones que permiten medir el rendimiento del sistema o de un componente del mismo. El sistema operativo sobre el que va a ejecutar el sistema simulador es Solaris 10.

2.4.1 *TRANS*

Conjunto de *benchmarks* transaccionales provenientes de *Wisconsin Comercial Workload Suite* [10]. Este tipo de aplicaciones trabajan con datos críticos, tanto personales como corporativos, ejecutándose sobre grandes y complejos servidores multi-puesto, el volumen de datos con el que trabajan es grande y dependen, en gran medida, de diversos servicios del sistema operativo, por lo que se debe utilizar un entorno de simulación de sistema completo. Para poder trabajar con este tipo de aplicaciones comerciales en sistemas al alcance de la investigación, los cuatro *benchmarks* que componen esta suite se han reducido en términos de tamaño y de tiempo de ejecución.

- **JBB** – Servidor de aplicaciones modelado a partir de SpecJbb.
- **Apache** – Servidor web dinámico modelado a partir de SURGE.
- **Oltp** – Procesamiento de transacciones online emulando TPC-C.
- **Zeus** – Servidor web estático modelado a partir de SURGE.

2.4.2 *PARSEC*

The Princeton Application Repository for Shared-Memory Computers [11] (*PARSEC*) es una colección de *benchmarks* compuesta de programas *multithreaded*. Contiene un conjunto de cargas de trabajo representativas de problemas en los que se emplean los chips multiprocesador. Fue creado de manera conjunta por Intel y la Universidad de *Princeton* y es ampliamente utilizado para la investigación por numerosas instituciones tales como la propia Intel para el desarrollo de nuevos procesadores.

- **Blackscholes** – Calcula el precio de una cartera de opciones bursátiles mediante ecuaciones diferenciales parciales *Black-Scholes*.
- **Canneal** – Optimiza el coste de encaminamiento en el diseño de chips.
- **Fluidanimate** – Dinámica de fluidos para animación con el algoritmo *Smoothed Particle Hydrodynamics* (*SPH*).
- **Streamcluster** – Agrupamiento online de datos.
- **Swaptions** – Permutas financieras.

2.4.3 *NPB*

NAS Parallel Benchmarks [12] (*NPB*) son un conjunto de *benchmarks* desarrollados por la NASA para medir el rendimiento en supercomputadores paralelos.

- **MG** (*MultiGrid*) – Aproxima la solución de la ecuación discreta de *Poisson* tridimensional usando el método V-cycle.
- **CG** (*Conjugate Gradient*) – Algoritmo para resolver sistemas lineales de ecuaciones.
- **FT** (Transformada de Fourier rápida) – Resuelve una ecuación diferencial parcial tridimensional usando la transformada rápida de Fourier.
- **SP** (*Scalar Pentadiagonal*) - Resuelve un sistema sintético de ecuaciones diferenciales parciales utilizando el algoritmo "*Scalar Pentadiagonal*".
- **LU** (*Lower-Upper symmetric Gauss-Siedel*) - Resuelve un sistema sintético de ecuaciones diferenciales parciales utilizando el algoritmo "*Symmetric Successive Over-relaxation*".

2.4.4 *SPEC*

Standard Performance Evaluation Corporation [13] (SPEC2006) es un conjunto de benchmarks para medir el rendimiento de la CPU en modo carga multi-programada. De esta manera se evalúa el rendimiento del sistema en un escenario orientado hacia el throughput, como el cloud-computing.

- **Astar** – Librería de búsqueda para mapas 2D, incluyendo el algoritmo A*.
- **Bzip2** – Versión modificada del algoritmo de compresión para que trabaje sobre la memoria en vez de sobre la entrada-salida.

- **Gcc** – Compilador de C, basado en la versión 3.2, generando código para el procesador Opteron.
- **Hmmer** - Análisis de secuencias de proteínas utilizando el modelo de perfiles ocultos de Markov.
- **Lbm** – Implementa el método Lattice-Boltzmann para simular fluidos en 3D.
- **Libquantum** – Simula un ordenador cuántico con el algoritmo de factorización de Shor.
- **Mcf** – Planifica el transporte público utilizando el algoritmo *network simplex*.
- **Milc** – Estudia las interacciones de la física subatómica.
- **Omnetp** - Utiliza el simulador OMNeT + + de eventos discretos para modelar una gran red *ethernet*.
- **Sphinx3** – Reconocimiento de voz de la Universidad *Carnegie Mellon*.

Se utilizan 3 tipos diferentes (Tabla 1) de cargas de trabajo en función del mapeado de los *benchmarks* sobres los *cores*:

- **RATE 3THR**: 4 cargas de trabajo donde se mapea la misma aplicación en todos los *cores*, salvo en uno, el cual no ejecuta ningún *benchmark*.
- **SINGLE 1THR**: 6 cargas de trabajo donde solamente se mapea un determinado *benchmark* a un único *core*.
- **HYBRID 4THR**: 20 cargas de trabajo híbridas resultantes de combinar distintos *benchmarks* tanto de *SpecINT* como de *SpecFP* donde cada *core* tiene mapeado un *benchmark* diferente.

RATE	astar-astar-astar
	hmmer-hmmer-hmmer
	lbm-lbm-lbm
	omnetpp-omnetpp-omnetpp
SINGLE	astar
	bzip2
	gcc
	lbm
	libquantum
	mcf
HYBRID	20 combinaciones de:
	astar(int), bzip2(int), gcc(int),hmmer(int)
	mcf(int), lbm(fp), milc(fp), spinx3(fp)

Tabla 1 – Cargas de trabajo multiprogramadas *SPEC*.

2.5 Configuración del sistema

La configuración del sistema utilizada para la evaluación de la propuesta queda resumida en la Tabla 2:

Parámetros del *core*		Parámetros de la memoria	
ISSUE/RETIRE WIDTH	2/2	L1 INSTRUCTION	PRIVATE, 32KB, 4-WAYS, 2-CYCLES, PIPELINED, 64B
SCHEDULER SIZE	UNIFIED, POINTED BASED [14], 32 ENTRIES	L1 DATA	PRIVATE, 32KB, 4-WAYS, 2-CYCLES, PIPELINED, 64B
FUNCTIONAL UNITS	2 ALU-INT/ 1 LD-ST/ 1 ALU-FP/ 1 BR	L2 SHARED S-NUCA [15]	2 SLICES PER CORE, EACH ONE: 512KB, 8-WAY, 8-CYCLES, 64B
FETCH-TO-DISPATCH	7 CYCLES	COHERENCE PROTOCOL	TOKEN COHERENCE BROASCAST [16]
BRANCH PREDICTOR	YAGS [17] 16K PHT 8K 8KB BTB, 16-ENTRY RAS	MAIN MEMORY	1 CONTROLLER EACH 4 CORES, 250 CYCLES
MEMORY SCHEDULING	UNLIMITED STORE-SETS	INTERCONNECTION NETWORK	MESH WITH ROUTERS BASED ON [18]

Tabla 2 – Configuración del sistema por *core*.

Capítulo III
3DDM

Este capítulo expone el trabajo realizado sobre el módulo *Opal* así como los aspectos más destacables de la arquitectura dinámica con apilamiento vertical.

3.1 Introducción

Como se ha introducido en el capítulo I, el núcleo de este proyecto es la implementación de un chip en 3D con microarquitectura dinámica.

Partimos de una implementación estática del módulo *Opal*, por tanto tenemos que modificar este módulo para que determinados aspectos de la microarquitectura sean compartidos de manera dinámica entre los distintos *cores* del chip, en concreto, vamos a compartir los siguientes componentes:

- La ventana de instrucciones o *reorder buffer* (*ROB*)
- Las unidades funcionales
- Las vías del *issue*
- Las vías del *retire*

En lo que resta del capítulo veremos distintos aspectos de la arquitectura dinámica con apilamiento vertical, concretamente, en el simulador se ha incluido:

- Asignación dinámica de los cuatro recursos anteriores
- Control de la compartición de los recursos con el fin de evitar la inanición
- Mecanismo *hardware* para la detección de *spinning*
- Estadísticas detalladas de la ocupación de los recursos

Además, los distintos elementos del sistema se configuran según lo especificado en la

Tabla 2.

El módulo *Opal* está compuesto por más de 800.000 líneas de código C++ prácticamente sin documentar, luego cualquier modificación importante, como las llevadas a cabo en este proyecto, requieren un alto grado de esfuerzo, ya que para modificar cierta funcionalidad hay que:

- Localizar la clase o clases donde está implementada esa funcionalidad que hay que modificar.
- Entender cómo está implementada esa funcionalidad, ya que el módulo *Opal* modela el funcionamiento de un procesador con arquitectura agresiva y por tanto el código no es nada intuitivo.

- Implementar las modificaciones pertinentes sin alterar en la medida de lo posible el funcionamiento del resto del módulo.

- Depurar el código, ya que cualquier parte del mismo interactúa con el resto de manera intensiva y cualquier pequeño error provoca no solo un incorrecto funcionamiento sino que lo usual es que las simulaciones se interrumpan de manera abrupta.

En lo restante de la presente memoria no se hace más hincapié en las modificaciones realizadas en el simulador puesto que lo importante no son las habilidades en la programación sino el funcionamiento de las modificaciones implementadas que sirven para modelar un *CMP* con apilamiento vertical.

3.2 Fuentes de mala utilización de los recursos en un *CMP*

Para maximizar la utilización de los recursos en un *CMP*, todos los *cores* del mismo deberían realizar trabajo útil durante la mayor parte de tiempo. Eso implica que cada *core* debe ejecutar instrucciones útiles bien de una tarea independiente o de una tarea paralela.

Para tareas independientes, basta con mapear las distintas tareas en los *cores* disponibles, mientras que para las tareas paralelas, las fuentes de ineficiencia son difíciles de descubrir para el *hardware* o de solucionar para el programador.

Primero, la completa paralelización de un programa es prácticamente inalcanzable y por lo tanto hay fases de la ejecución que son secuenciales. Segundo, el balanceo perfecto del trabajo sobre los distintos *cores* es una tarea complicada y difícil de conseguir de manera estática y además se agrava, por ejemplo, con el impacto de la jerarquía de memoria sobre la ejecución de los *threads*, como consecuencia hay *threads* que llegan antes que otros a una barrera y que deben esperar. Y tercero, los *threads* que componen un programa paralelo normalmente tienen que interactuar entre ellos, lo que les obliga a sincronizarse para, por ejemplo, acceder a una sección crítica.

Cada unos de los tres aspectos mencionados implica que, durante ciertos momentos no despreciables de la ejecución, los *threads* no están haciendo trabajo útil, ya que están en fases de sincronización. Varios trabajos recientes [19] intentan solucionar este problema incorporando arquitecturas heterogéneas que incluyen uno o más *cores* más potentes en los que se ejecutan los *threads* causantes de la sincronización con el fin de acelerar su ejecución y que todos los *threads* continúen haciendo trabajo útil lo antes posible.

A pesar de que la idea ataca los tres puntos de ineficiencia mencionados no está exenta de problemas, ya que implica migración de procesos hacia los *cores* potentes y, en ciertos casos, la sobrecarga de la migración del *thread* de un *core* normal a un *core* potente puede ser más ineficiente que mantener al *thread* en el *core* en el que se está ejecutando, luego hay que decidir en qué casos merece la pena acelerar la ejecución del *thread* causante de la ineficiencia y cuando no. Otro problema sería la serialización innecesaria que se produce cuando el número de secciones críticas independientes excede el número de *cores* potentes ya el *thread* tiene que esperar a que haya un *core* potente libre para la ejecución de la sección crítica.

La propuesta de este proyecto ataca el problema aprovechando las oportunidades que brinda el 3D *stacking*. Se asignan dinámicamente recursos a los distintos *cores* sin favorecer explícitamente a los *threads* que están haciendo trabajo útil sino que se les arrebatan a aquellos

que están realizando un trabajo improductivo ya que, por ejemplo, están en una fase de sincronización.

De manera ortogonal a la sincronización, como fuente de mala utilización de los recursos, encontramos las especulaciones incorrectas. La ejecución fuera de orden utilizada en los procesadores de hoy en día requiere ejecución especulativa y por tanto, en determinadas ocasiones, la ejecución es errónea y hay que volver atrás. Imaginemos una carga de trabajo multiprogramada con un *thread* cuyo comportamiento no es capaz de capturarlo correctamente la lógica de predicción de saltos, junto con otros *threads* con una ejecución más regular. Por una parte, el *thread* irregular tiene que desechar gran cantidad de trabajo, debido a las especulaciones incorrectas, lo que se traduce en una mala utilización de los recursos, así como de la energía, mientras que los *threads* regulares no sufren este problema. Sería adecuado poder transferir parte de los recursos del *thread* irregular para así acelerar la ejecución de los *threads* regulares de tal modo que se utilicen correctamente los recursos y no se desperdicie energía. Veremos que la propuesta es capaz de realizarlo de manera transparente mientras que la sincronización requerirá un esfuerzo mayor.

3.3 Arquitectura base

Para la arquitectura base se utiliza una arquitectura con apilamiento vertical en 3D con "lógica sobre lógica", es decir, donde los *cores* de apilan sobre otros *cores* y donde las *caches* se apilan sobre otras *caches* y la distribución de los distintos elementos que conforman los *cores* de distribuyen de la misma manera. Los *cores* son superescalares, con ejecución fuera de orden, donde el número de instrucciones en vuelo es pequeño; partiendo de esta configuración y de acuerdo con [20], podemos utilizar recursos comunes sin que ello afecte al ciclo de reloj o a la longitud del *pipeline* del procesador.

En particular, supondremos estructuras apiladas para *physical register file* (*PRF*), *reorder buffer* (*ROB*), *instructions queue* (*IQ*) y *load-stores queue* (*LSQ*). Las técnicas descritas en [21] [22] se usarán para compartir cada estructura.

El análisis llevado a cabo en [20] indica que, para un modelo realista con apilamiento 3D, es posible conectar cuatro capas diferentes, con un *core* por capa, en menos de 3ps con tecnología en 32nm, lo que posibilita la compartición de recursos entre *cores* sin incrementar la longitud del *pipeline* o la frecuencia de operación. Si llevásemos a cabo la misma compartición de recursos en un sistema 2D, la comunicación entre recursos compartidos requeriría más de 3ns para la misma tecnología, lo que resulta tres órdenes de magnitud superior al tiempo requerido en 3D. Esta gran diferencia dificulta el uso de propuestas similares en 2D, puesto que la compartición de recursos requeriría hasta 10 ciclos de reloj.

Para mantener el análisis de la disipación de potencia y el *delay* de las conexiones llevado a cabo en [20], restringimos nuestra arquitectura a 4 capas. Analizaremos esta arquitectura en *CMPs* con cuatro y ocho *cores* (Figura 2) para analizar el comportamiento cuando se añaden más *cores* al *CMP* con respecto a [20].

El 3D *stacking* también afectará a la jerarquía de *cache*, concretamente, la latencia de acceso a la *cache* de último nivel (*LLC*) será mucho menor que en una arquitectura plana. En nuestra arquitectura utilizamos una red similar a la propuesta por [18] para minimizar la latencia *on-chip*. Para maximizar la utilización del ancho de banda *on-chip*, utilizamos el protocolo de coherencia *Token Broadcast* [16].

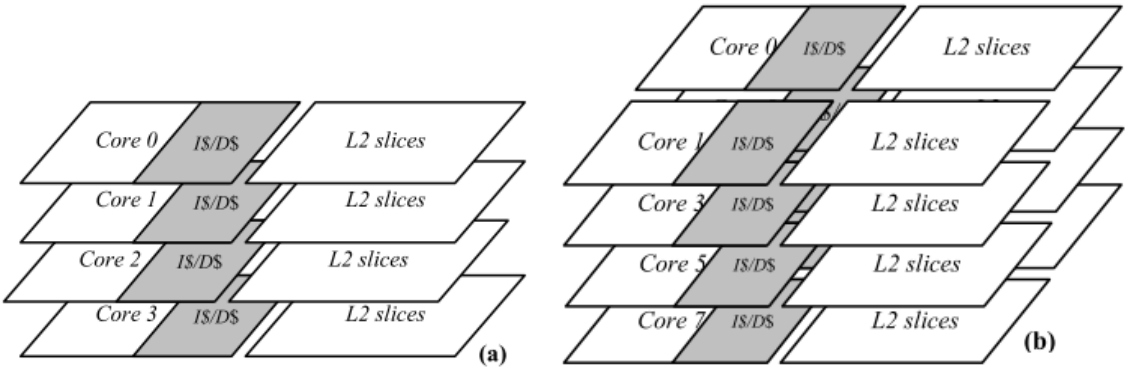

Figura 2 – Arquitectura: (a) *CMP* con 4 *cores*, (b) *CMP* con 8 *cores*.

3.4 Compartición de unidades funcionales

En [20] se realiza un análisis exhaustivo para determinar cómo la propuesta puede beneficiar al comportamiento del sistema, sin embargo no tienen en cuenta las unidades funcionales. Sí comparten estructuras que soportan la ejecución fuera de orden, como son la *LSQ*, *ROB*, *PRF* e *IQ*, pero no hacen lo propio con las unidades funcionales, con lo que se puede saturar el ancho de banda del *retire*.

En nuestra propuesta, sí se comparten las unidades funcionales sin modificar el *front-end* del procesador y por tanto el ancho de banda del *fetch* permanece fijo. Sin embargo, como bien es sabido, incrementar el número de vías de ejecución implica incrementar significativamente la lógica de *data-path*, de hecho, la red de *bypass* aumenta como el cuadrado del número de vías, lo que en un sistema 2D incrementaría la longitud de los cables y la complejidad del *routing*, lo que seguramente incrementaría la longitud del *pipeline*. Por el contrario, la tercera dimensión que aporta el apilamiento en 3D alivia el problema.

La Figura 3 muestra cómo, utilizando cuatro capas apiladas con *cores* de dos vías, sería posible conectar los multiplexores usando *through-silicon vías* (*TSV*) para permitir el *bypass* entre las salidas de las unidades funcionales de cualquiera de las capas sin incrementar la propia red de *bypass*. El *delay* de peor caso, introducido por las *TSVs*, es dos órdenes de magnitud menor que el ciclo de reloj y como no hay un *overhead* en los cables no es necesario incrementar la longitud del *pipeline*.

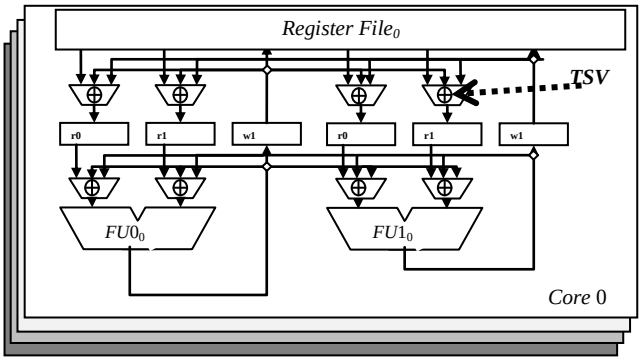

Figura 3 – Red de *bypass* en 3D.

Otra cuestión con las unidades de ejecución es el número de puertos del banco de registros. Para no incrementar el número de puertos de escritura restringimos a las instrucciones a una

determinada capa desde el *dispatch* hasta el *retire*, cuando una instrucción hace el *dispatch* reservamos todas las estructuras fuera de orden en la misma capa.

La instrucción solo podrá utilizar la unidad funcional de la capa en la que está alojada. Intuitivamente, la utilización de la unidad funcional en cada capa será proporcional al número de entradas reservadas por *thread* en la capa en la que reside la unidad funcional. Por lo tanto, no se observará una degradación con motivo de esta restricción.

En lo que respeta al número de puertos de lectura del banco de registros, de acuerdo con [23], el apilamiento en 3D permite incrementar el número de puertos sin incrementar significativamente el coste del banco de registros. Se pueden utilizar otras técnicas como [24].

3.5 Compartición de recursos

Para maximizar el rendimiento, la compartición de recursos no es ilimitada, al contrario que en [20]. Además hay que controlar qué es lo que está haciendo cada *core* para que no se produzcan situaciones indeseables.

Concretamente, durante la sincronización los *cores*, pueden robar recursos valiosos que otros *threads* están utilizando para realizar un trabajo útil y además esos recursos se van a desperdiciar ya que, durante la sincronización, el trabajo no es útil. El código de sincronización se ejecuta con un comportamiento casi perfecto de la *cache* y de la lógica de predicción de saltos lo que, en la mayoría de los casos, implica un *ratio* del *fetch* más grande con respecto a los *threads* que están realizando trabajo útil. Ese comportamiento se produce de igual modo en el *idle-loop* del sistema operativo cuando los *cores* no están ejecutando trabajo útil, y en ambos casos puede llevar a problemas de inanición con el resto de los *cores* ya que los que están dentro de un *spin-loop* se comportan de una manera muy voraz con los recursos compartidos.

Para evitar esto, proponemos detectar el *spinning* y regular su ejecución reduciendo al mínimo los recursos asignados a los *threads* que se encuentren dentro de un *spin-loop*, dejándolos con una única instrucción en vuelo, favoreciendo a los *threads* que están realizando un trabajo útil. De manera indirecta, estaremos acelerando los procesos de sincronización ofreciendo más recursos a los *threads* que, por ejemplo, están dentro de una sección crítica.

Por otra parte, es necesario determinar un umbral por debajo del cual los recursos no se comparten para así evitar problemas de inanición ya que, por ejemplo, cuando un *thread* tiene un fallo en la *cache* de instrucciones, el resto de *threads* podrían arrebatarle la totalidad de los recursos, de tal modo que cuando el *thread* que ha fallado intentase reanudar la ejecución no podría ejecutar y podría llevarle un tiempo significativo volver a conseguir los recursos que tenía antes de fallar en la cache.

Para prevenir este tipo de situaciones, se establece un mínimo para que el resto de *cores* no puedan reservar recursos de manera remota de los recursos de un determinado *core*. Esta regla se combina con la anterior, combinando trabajo útil y evitación de la inanición.

Veamos un ejemplo (Figura 4) sobre cómo funcionan las restricciones anteriores en el que interviene una sección crítica. En el ejemplo aparecen dos *threads* (T0 y T1), que forman parte de la misma aplicación, y además hay un *thread* (T2) perteneciente a otra aplicación distinta. Inicialmente los tres *threads* ejecutan de un modo normal y, en un determinado momento, T0 toma el *lock* para acceder a la sección crítica. Momentos después, T1 intenta acceder a la misma sección crítica, pero como el *lock* está tomado entonces se inicia un *spin-loop*. Progresivamente

los recursos de T1 le van siendo arrebatados por T0 de tal modo que P0 incrementa el número de instrucciones en vuelo y por tanto la ejecución de la sección crítica se acelera. Cuando T0 termina la ejecución de la sección crítica y abre el *lock*, T1 lo toma, entra en la sección crítica y los recursos que le habían sido arrebatados mientras se encontraba en el *spin-loop* los va recuperando hasta que se equilibra la asignación de recursos entre los dos *cores*. La ejecución continúa de un modo normal, hasta que T0 llega a una barrera donde, de nuevo, se entra en un *spin-loop* y los recursos de P0 pasan a P1. Al mismo tiempo, el *thread* independiente T2 finaliza su ejecución y comienza el *idle-loop* del sistema operativo, entonces gran parte de los recursos de P2 pasan a P1 acelerando la ejecución de T1. En cierto momento, T1 llega a la barrera donde se encuentra T0 y ambos la cruzan, a partir de ese punto la asignación de recursos vuelve a equilibrarse, si bien pueden continuar con los recursos de P2 y por lo tanto la ejecución para T0 y T1 es más rápida que al inicio.

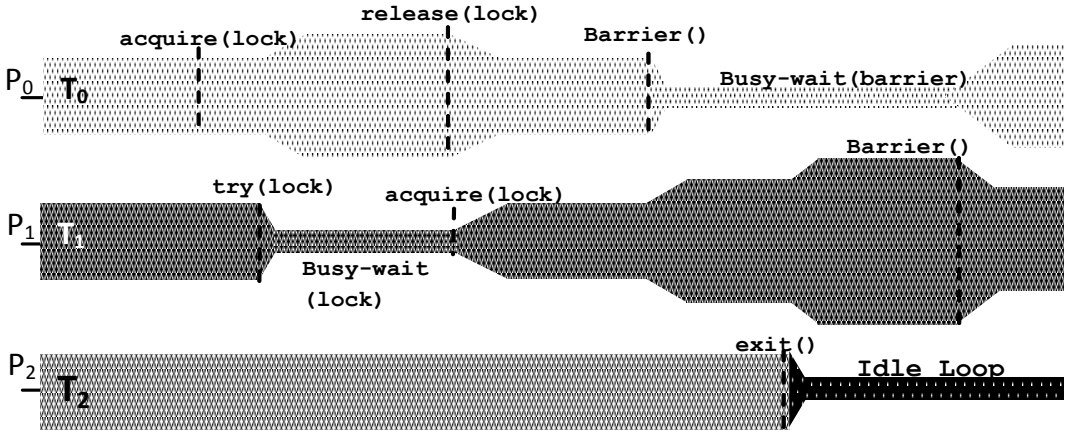

Figura 4 – Ejemplo de la evolución de las instrucciones en vuelo en cada *core*.

3.6 *Hardware* de detección de *spinning*

Los procesos de sincronización son bastante complejos y dependen de un gran número de factores [25]. Sin embargo, desde la perspectiva del *hardware*, puede ser relativamente sencillo usando, por ejemplo, la metodología propuesta por [26].

Esta metodología afirma que si entre dos ejecuciones del mismo *Backward Control Transfer* (*BCT*), un salto hacia atrás en el flujo de ejecución, el estado arquitectural del procesador y la memoria no varían entonces significa que no se está realizando trabajo útil y por tanto se está en una fase de *spinning*.

Se propone un módulo de detección de *spinning* (*SDM*) el cual se encarga de identificar los *spinning BCT* del código regular. Cuando el contador del programa alcanza un *BCT*, el módulo *SDM* comienza a controlar a las instrucciones en la etapa del *retire* que modifican el banco de registros. El valor inicial del registro se almacena en el *SDM* y se marca el registro como modificado. Si otra instrucción dentro del mismo *BCT* escribe en el mismo registro antes de la próxima iteración del *BCT* con un valor idéntico al inicial entonces la entrada del *SDM* se borra. Si no hay entradas en el *SDM* cuando se ejecuta de nuevo el *BCT* entonces el estado del procesador no ha variado. Ante la ejecución de un *non-silent store* se considera que el código está realizando trabajo útil y la detección de *spinning* se cancela.

En resumen, si el estado de los registros y de la memoria no ha cambiado entre dos *BCTs* consecutivos, entonces el *SDM* determina que la ejecución de un *thread* está es una fase de

spinning. El *SDM* requiere una cantidad pequeña de almacenamiento para poder controlar el estado de los registros y trabaja de forma paralela en la etapa *commit*.

Sin embargo, el alcance limitado del análisis de las operaciones de memoria restringe su efectividad con los algoritmos de sincronización que utilizan la memoria para almacenar cierta información auxiliar. Esta situación ocurre con los algoritmos de sincronización en dos fases, pero está también presente en otros algoritmos de sincronización donde se requiere la coordinación de tres o más *threads*. Los autores [26] proponen la ayuda del *software* para solventar este problema con los algoritmos en dos fases pero no indican cómo resolverlo para los algoritmos avanzados de sincronización en una fase. Una vez probado esta metodología podemos afirmar que no es capaz de capturar nada de *spinning* con nuestra configuración SPARCv9/Solaris 10 [27] debido a que los algoritmos de sincronización y el *idle-loop* del sistema operativo requieren cambios en memoria, lo que invalida la metodología propuesta por [26]; esta observación es corroborada también por [28].

Para controlar los cambios en memoria optamos por la propuesta heurística de [29]. Esta propuesta establece que el procesador está en una fase de *spinning* si el número de *stores* (en modo privilegiado y en modo usuario) y/o el número de *loads* (solo en modo usuario, por ejemplo, para detectar las búsquedas iterativas dentro de un *array*) sobre direcciones únicas está por debajo de un determinado umbral. La unicidad de las operaciones en memoria viene dada tanto por la dirección como por los valores para los *stores*, mientas que para los *loads* basta con la dirección. Después de un exhaustivo estudio del espacio de diseño, [28] afirman que la mejor opción es al menos 8 *loads/stores* únicos cada 1024 instrucciones. El uso aislado de esta segunda propuesta crea falso *spinning* en aplicaciones dominadas por sincronización en modo usuario como por ejemplo en las *TRANS*.

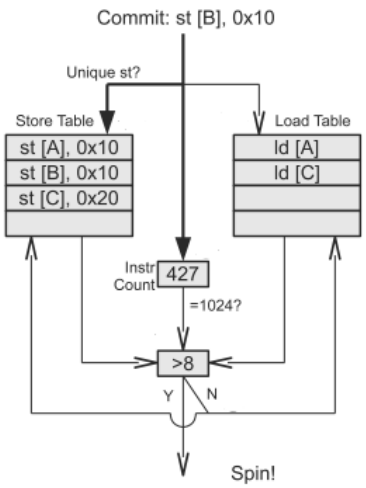

Figura 5 – *Hardware* para la detección de *spinning*.

Decidimos utilizar una combinación de las dos propuestas explicadas: [26] para el estado de los registros y [29] para el estado de la memoria. La condición para determinar que un *thread* está en una fase de *spinning* es que el estado de los registros del procesador, entre la ejecución de dos *BCTs* consecutivos, permanece sin cambios y que la frecuencia de *stores* y *loads* únicos cae por debajo de 8.

3.7 Algoritmo para la asignación de recursos

Resumiendo la discusión anterior, proponemos el siguiente algoritmo para la asignación de los recursos compartidos por los *threads* que se están ejecutando.

Sea T_i un *thread* ejecutado por el procesador P_i y T_j ejecutado por P_j. Sea R_j un recurso de ejecución del procesador P_j.

El *dispatch* de una nueva instrucción de T_i se realizará sobre la entrada R_j si y solo si:

1. T_i no está en una fase de *spinning*.
2. Si T_j no está es una fase de *spinning*: los recursos disponibles de P_j son más de los necesarios para poder realizar el *dispatch* instrucciones en el *front-end* de P_j.
3. Si T_j está es una fase de *spinning*: los recursos disponibles en P_j son más de los requeridos para realizar el *dispatch* de dos instrucciones.

Siguiendo este conjunto de reglas, cuando un procesador inicia un *spin-loop*, en un periodo corto de tiempo, perderá la mayor parte de los recursos y se comportará como un procesador en orden. A través de esta aproximación, las principales fuentes de mala utilización de los recursos en las aplicaciones *multithreaded*, como la contención en las secciones críticas, las fases del código secuenciales o las cargas de trabajo mal balanceadas, son superadas de manera automática con unos resultados muy positivos.

Capítulo IV
Evaluación

En este capítulo se exponen y analizan los resultados de la evaluación de la propuesta.

4.1 Metodología

Para evaluar la eficacia de la propuesta, compararemos nuestra solución con diferentes diseños tanto estáticos como dinámicos. Para ello, emplearemos un simulador de sistema completo, capaz de involucrar todos los aspectos tales como la actividad del sistema operativo.

En cuanto a los *benchmarks* que se van a utilizar, como ya se ha comentado en el capítulo II, utilizamos tanto aplicaciones *multithreaded* como multiprogramadas. Las *suites* de aplicaciones son *NPB, PARSEC*, Wisconsin Server Suite y *SPEC* 2006. La combinación de estas cargas de trabajo evaluará nuestra propuesta desde múltiples enfoques y por tanto permitirá obtener unos datos más fiables, ya que, por ejemplo [20], solo utiliza aplicaciones multiprogramadas y además no utilizan un simulador de sistema completo, luego no tiene que enfrentarse a los problemas que provoca la voracidad del *idle-loop* del sistema operativo cuando un determinado *core* no está ejecutando ningún *thread;* tampoco somete su propuesta a aplicaciones *multithreaded* luego no tienen que enfrentarse a la problemática de la voracidad de los *spin-loops* de la sincronización.

Para realizar la evaluación, utilizamos el simulador *GEMS*, junto con *Simics*, ejecutando un sistema SPARCv9 sin modificar. El sistema operativo utilizado es *Solaris* 10. Las aplicaciones se ejecutan múltiples veces hasta alcanzar un intervalo de confianza del 95% (en muchos casos, las barras de error prácticamente no se aprecian). Las aplicaciones pasan por una fase de inicialización donde se "calientan" las *caches*, el *TLB*,…

Vamos a comparar nuestra propuesta 3D dinámica con *cores* estáticos con el mismo número de recursos (Medium-End). Un procesador que dobla el número de recursos (High-End) ya que posee 4 vías, el doble de unidades funciones y 64 entradas en la ventana de instrucciones. La propuesta de [20] con el mismo número de recursos (PLAIN3D). Nuestra propuesta se denota por (IMPROV3D). Todas las configuraciones, a excepción de High-End, ocupan la misma área, el cual representa un procesador con recursos sobredimensionados para no perder rendimiento en las cargas de trabajo *single-threaded*. Para facilitar la interpretación de los resultados, todas las configuraciones utilizan la misma jerarquía de memoria.

4.2 Cargas de trabajo multiprogramadas

En la primera parte de la evaluación nos centramos en las aplicaciones multiprogramadas (Tabla 1) utilizando aplicaciones pertenecientes a *SPEC* 2006.

4.2.1 HYBRID 4THR

Como ya se introdujo en el capítulo II, se utilizan 20 cargas de trabajo híbridas donde cada *core* ejecuta una aplicación diferente de *SpecINT* o *SpecFP*. De este modo, podemos comprobar al comportamiento de aplicaciones con saltos difíciles de predecir (*int*) junto con aplicaciones con una predicción de saltos más precisa (*fp*).

Los resultados (Figura 6) muestran un beneficio en el rendimiento de IMPROV3D modesto (cercano al 5%) ya que todos los *cores* están en uso. Sin embargo, vemos que PLAIN3D muestra un rendimiento por debajo de la configuración base (Medium-End).

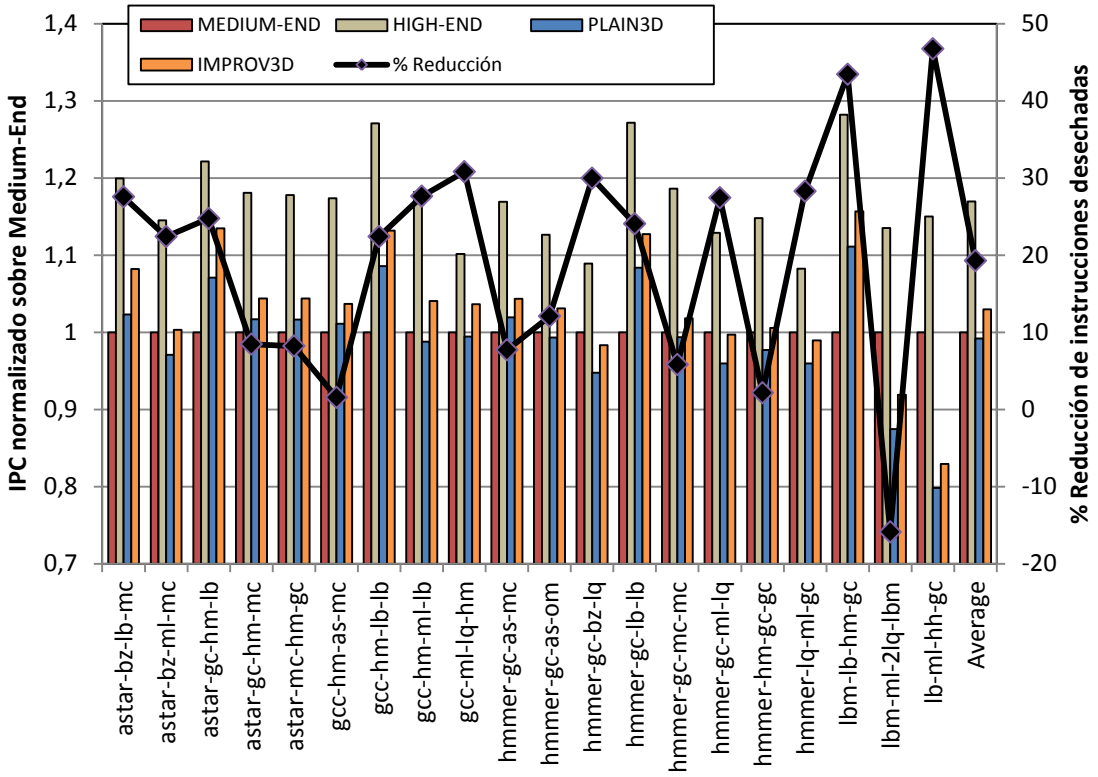

Figura 6 – IPC normalizado sobre Medium-End para HYBRID 4THR.

La figura también muestra la reducción de las instrucciones que se desechan, concretamente, el eje *y* secundario muestra el porcentaje de ahorro de instrucciones desechadas por parte de IMPROV3D sobre Medium-End. El número de instrucciones desechadas es la diferencia entre el número de instrucciones decodificadas menos el número de instrucciones que finalizan su ejecución llegando a la etapa de *commit*. En la mayoría de los casos, hay una correlación entre la mejora del rendimiento y el ahorro de instrucciones desechadas, lo que, al final, se traduce en un ahorro de energía.

En PLAIN3D, el número de instrucciones desechadas también se reduce, pero al no compartir las unidades funcionales no consigue sacar un beneficio significativo. Hay cargas de trabajo donde la compartición de recursos degrada el rendimiento como en *lbm-milc-libquantum-lbm* y en *libquantum-milc-hmmer-gcc;* éstos conjuntos representan un caso patológico pues ciertas aplicaciones tienen una localidad de memoria muy mala y otras muy buena, con lo que resulta que las aplicaciones con una buena localidad monopolizan los recursos y, como consecuencia, el rendimiento general baja.

En todo caso, el tamaño de la tabla del predictor de saltos está sobredimensionado, con tablas más pequeñas, el número de especulaciones erróneas aumentaría, luego IMPROV3D lograría aprovecharse aún más y por tanto la ventaja sería mayor. Aunque sería interesante investigar más a fondo la interacción entre la lógica de predicción de saltos y la propuesta y además

compartir dinámicamente la capacidad de almacenamiento, lo posponemos para un futuro trabajo.

Una vez visto el rendimiento y el ahorro en las instrucciones desechadas, vamos a mostrar de dónde proviene el beneficio para este tipo de cargas de trabajo en el que todos los *cores* están ocupados y además no hay fases de sincronización. Vamos a centrarnos en la carga de trabajo *astar-mcf-hmmer-gcc*, ya que muestra una gran disparidad en la precisión de la predicción de saltos, y analizar la asignación de los recursos a cada *core*.

En este caso, el porcentaje de predicciones mal hechas por *core* son 17%, 6%, 10% y 2%. Para los *cores* Medium-End, el porcentaje de instrucciones no *committeadas* son 40%, 22%, 17% y 10% respectivamente, vemos que hay un gran porcentaje de instrucciones que no completan su ejecución para el *core* 0. Si comparamos este porcentaje con IMPROV3D, los resultados son 35%, 23%, 16% y 7%. En definitiva, vemos que el número de instrucciones que se desechan baja y esto se debe a que cuando un *thread* muestra un número considerable de especulaciones mal realizadas se producen muchos *rollbacks* y por tanto el resto de *cores*, con un porcentaje de menor de especulaciones mal hechas, tienen más oportunidades de arrebatarle recursos.

Como consecuencia, en esta carga de trabajo en particular, el *core* que ejecuta *astar* tiene un número menor de instrucciones que no completan su ejecución. La Figura 7 muestra la evolución de los recursos asignados en cada *core* a lo largo de la ejecución para IMPROV3D. Podemos ver que *astar* tiene un porcentaje de recursos menor al 25% en gran parte de la ejecución, y en las fases donde la especulación de saltos ofrece una precisión menor, los recursos se sitúan un poco por encima del 10%.

Por último, en la arquitectura High-End, el porcentaje de instrucciones mal especuladas es 47%, 28%, 20% y 21% respectivamente, como consecuencia de mantener el mismo predictor de saltos, ya que es habitual incrementar la capacidad de la lógica de predicción de saltos cuando el número de instrucciones en vuelo es mayor.

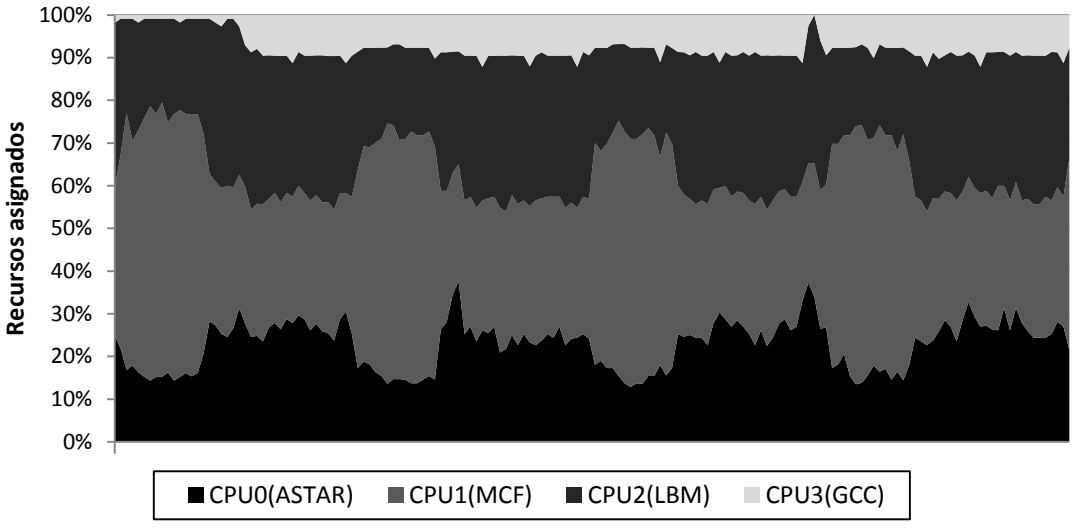

Figura 7 – Evolución de los recursos asignados por *core* para *astar-mcf-hmmer-gcc* en IMPROV3D.

4.2.2 RATE 3THR

Las RATE 3THR se componen de cuatro cargas de trabajo, donde los tres *cores* (*core* 0 – *core* 2) ejecutan la misma aplicación y un *core* (*core* 3) queda ocioso ejecutando el *idle-loop* del sistema operativo.

Los resultados (Figura 8), muestran un beneficio medio cercano al 10% debido a que los recursos del *core* ocioso se reparten entre los otros tres *cores* incrementando el rendimiento de éstos. Mientras que PLAIN3D muestra un rendimiento muy por debajo del Medium-End debido a la voracidad del *core* ocioso como ya se ha mencionado anteriormente.

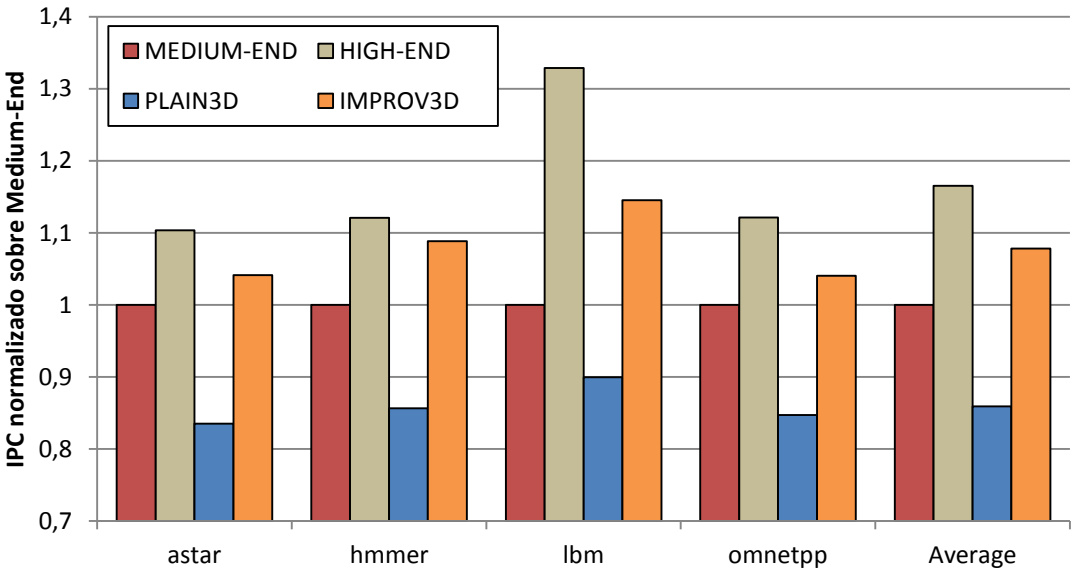

Figura 8 – IPC normalizado sobre Medium-End para RATE 3THR.

Vamos a analizar el porcentaje de recursos asignados a cada *core* para PLAIN3D e IMPROV3D (Figura 9). Vemos que para PLAIN3D el *core* que ejecuta el *idle-loop* monopoliza los recursos acaparando el 80%, a pesar de no estar realizando trabajo útil, por lo que el rendimiento de las tres instancias de *astar*, corriendo en los *cores* restantes, decae de manera abrupta. Sin embargo, esta situación no sucede en IMPROV3D puesto que la lógica de detección de *spinning* detecta que el *core* 3 no está realizando trabajo útil y reduce sus recursos al mínimo, por tanto el resto de los *cores* se benefician de ello, en concreto vemos que el *core* 3 tiene asignados el mínimo de recursos durante toda la ejecución de la carga de trabajo. Este mismo comportamiento se observa de manera idéntica en las tres cargas de trabajo restantes.

Este resultado no se aprecia en [20] debido a que en ese trabajo no se utiliza un simulador de sistema completo por lo que el *idle-loop* del sistema operativo no entra en escena.

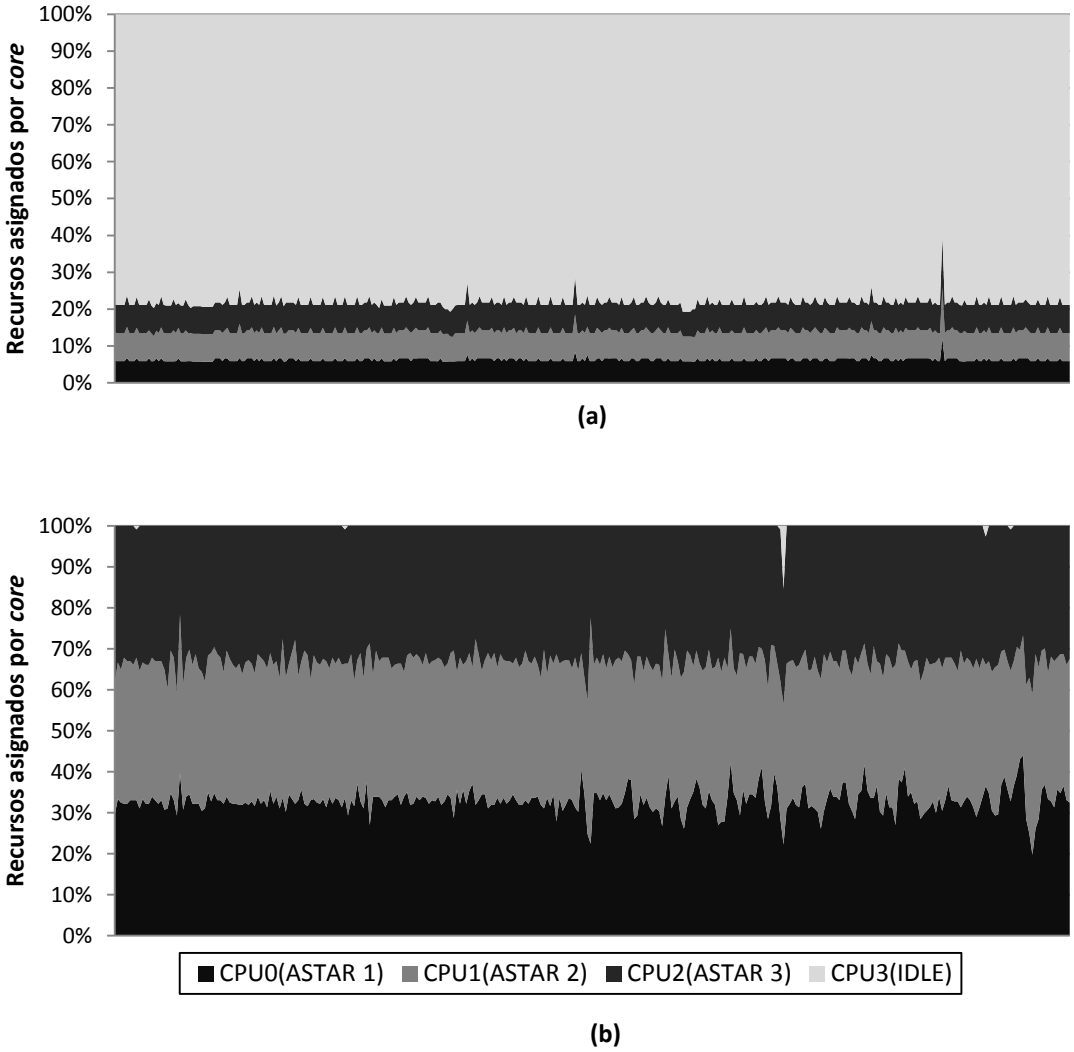

Figura 9 – Evolución de la asignación de recursos por *core* ejecutando tres instancias de *astar* para: (a) PLAIN 3D y (b) IMPROV3D.

4.2.3 SINGLE 1THR

El último tipo de cargas de trabajo multiprogramadas, donde solamente uno de los cuatro *cores* ejecuta una aplicación, quedando los otros tres ociosos.

El rendimiento ofrecido por IMPROV3D (Figura 10) no solo mejora el propio de Medium-End sino que incluso mejora a High-End debido a que, en este caso, los recursos de los tres *cores* ociosos pasan al *core* que está ejecutando la aplicación, incrementando su rendimiento a la vez que el número de instrucciones ejecutadas por los *cores* que ejecutan el *idle-loop* disminuye notablemente ya que, como sucede con las RATE 3THR, los recursos de los *cores* ociosos se mantienen al mínimo.

De nuevo, PLAIN3D muestra un rendimiento medio muy por debajo del que ofrece Medium-End y se debe a la voracidad de los tres *cores* ociosos como sucedía con RATE 3THR los cuales dejan sin recursos a la único *thread* útil del sistema.

La Figura 11 muestra la evolución del IPC de High-End e IMPROV3D durante los primeros 200 millones de instrucciones ejecutadas tras el *checkpoint*. IMPROV3D muestra un beneficio en el tiempo de ejecución cercano al 30% gracias a que puede beneficiarse de los recursos de los tres *cores* ociosos llegando a emplear una ventana de instrucciones de 128 entradas, si bien el ancho de banda del *fetch* limita el rendimiento ya que se mantiene constante.

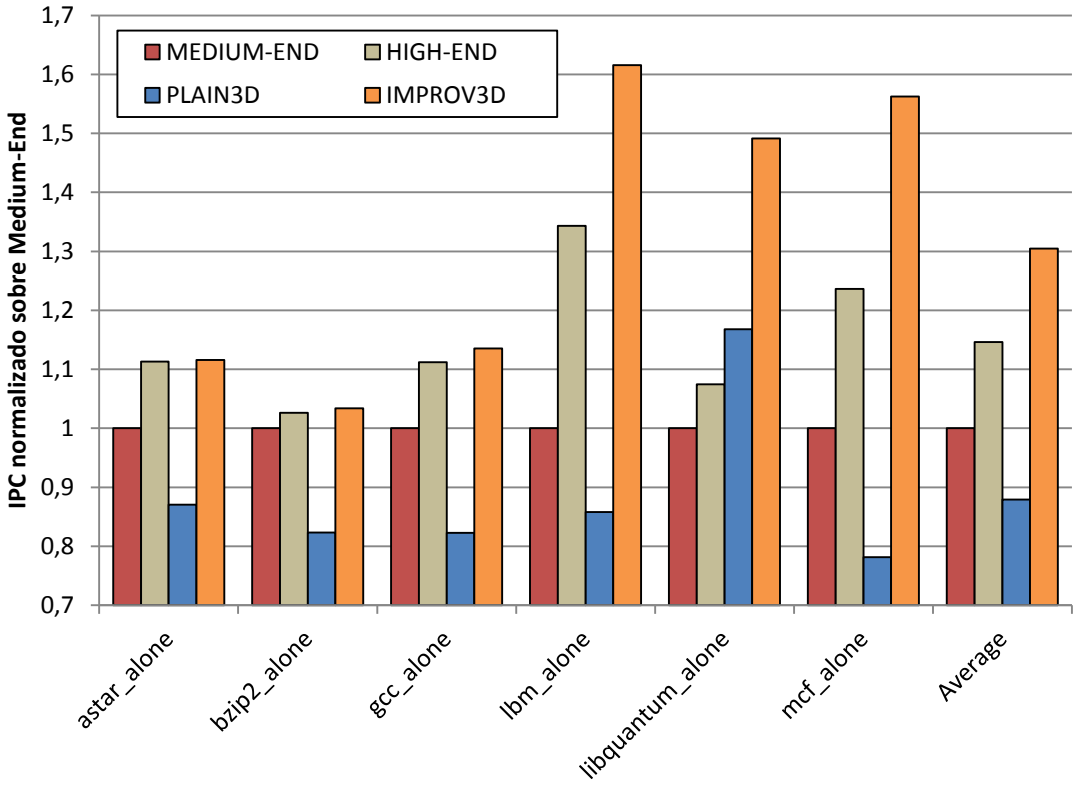

Figura 10 – IPC normalizado sobre Medium-End para SINGLE 1THR.

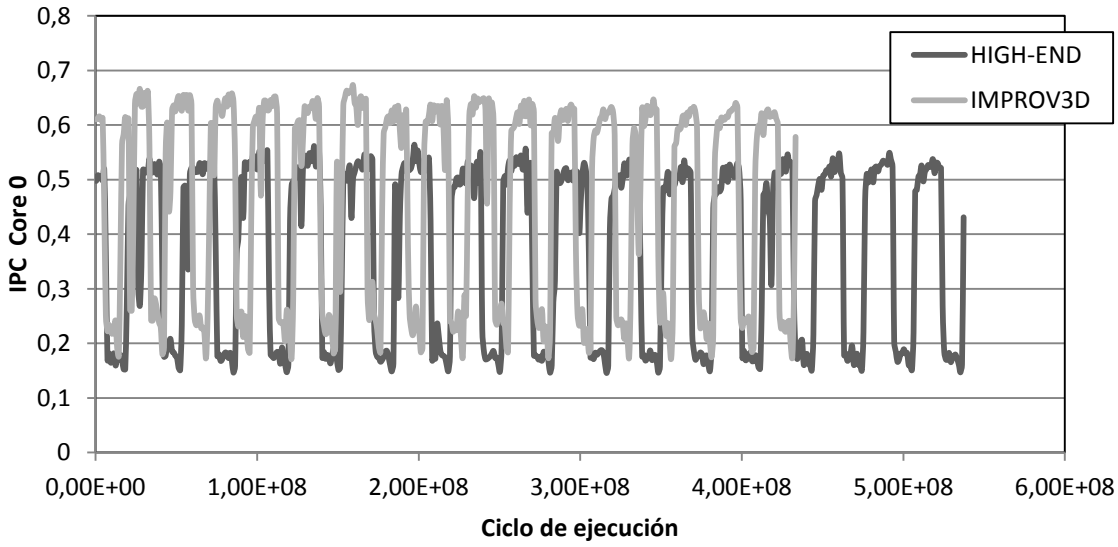

Figura 11 – Evolución del IPC del *core* 0 ejecutando *mcf*.

4.2.4 Resumen de los resultados

Hemos visto el rendimiento de las cuatro configuraciones en tres escenarios distintos:

- Todos los *cores* ejecutando diferentes aplicaciones.
- Tres *cores* ejecutando la misma instancia de la aplicación y un solo *core* ocioso.
- Un *core* ejecutando una aplicación y tres *cores* ociosos.

IMPROV3D (Figura 12) es capaz de superar el rendimiento de Medium-End para los tres tipos de cargas de trabajo holgadamente y es, incluso, capaz de superar a High-End en SINGLE 1THR; este beneficio se debe, en gran parte, a la habilidad de la lógica de detección de *spinning* para detectar que el *idle-loop* del sistema operativo, que se ejecuta en los *cores* ociosos, no es trabajo útil, de tal forma que los *cores* que están ejecutando la aplicación pueden aprovechar estos recursos y así incrementar el rendimiento.

PLAIN3D (Figura 12) no incorpora una lógica de detección de *spinning* puesto que [20] no utiliza un simulador de sistema completo y por tanto no se enfrenta a la problemática de la voracidad el *idle-loop* , el cual presenta un comportamiento en memoria casi ideal. Como consecuencia, el rendimiento que ofrece es peor que el propio de Medium-End para los tres tipos de cargas de trabajo mostrando un rendimiento significativamente negativo con RATE 3THR y SINGLE 1THR debido precisamente a la aparición en escena del *idle-loop*.

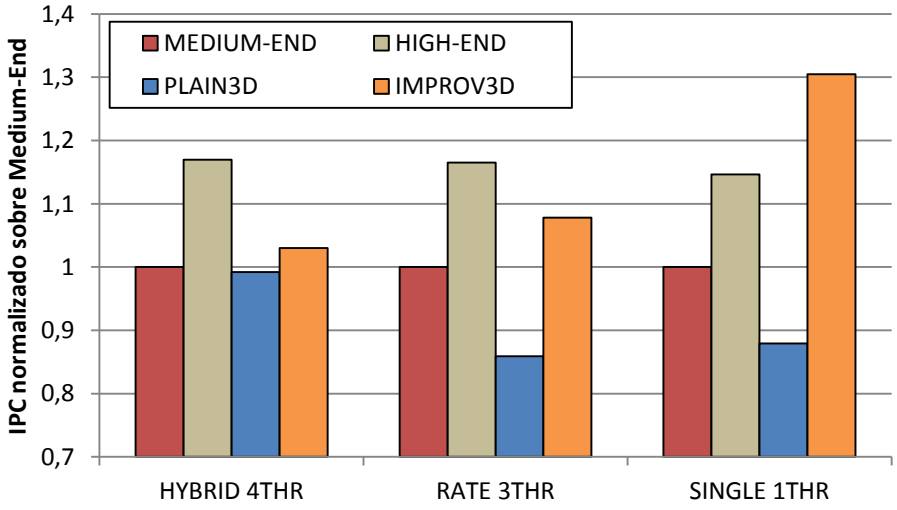

Figura 12 – IPC medio de los *cores* activos normalizado sobre Medium-End.

4.3 – Cargas de trabajo *multithreaded*

Una vez analizado el rendimiento de la propuesta con cargas de trabajo multiprogramadas, donde no hay interacción entre los distintos *threads*, pasamos a analizar el comportamiento con cargas de trabajo *multithreaded*, donde sí hay interacción entre los distintos *threads*. Se van a utilizar catorce aplicaciones distintas descritas en el Capítulo II.

De nuevo, vamos a utilizar IMPROV3D, PLAIN3D, Medium-End y High-End para nuestro análisis. No se incluye la comparación con un *CMP* asimétrico como *ACS* [19] puesto que esta propuesta requiere modificar el código fuente de las aplicaciones. Además, tenemos acceso al código fuente de determinadas aplicaciones pero un porcentaje significativo de la sincronización

proviene del *kernel* del sistema operativo. Se ha intentado inferir cuándo un *thread* entra en una sección crítica analizando el código ejecutado por cada *core* en el simulador y aunque en algunos casos es sencillo determinarlo, en otros casos es extremadamente complejo.

IMRPOV3D (Figura 13) en un *CMP* con 4 *cores* muestra un rendimiento superior al de la arquitectura High-End en todos los casos, salvo en la aplicación *canneal*. Más que los resultados, lo más importante es que el conjunto de mecanismos propuestos, con la mitad de área del procesador, es capaz de mejorar la arquitectura estática de High-End. Como determina [20], el área de las *TSV* por *core* es alrededor de 0,0004mm^2 utilizando una tecnología de 32nm lo cual es insignificante comparado con el resto del área del *core*. Otro *hardware* adicional, como la lógica de detección de *spinning*, es también despreciable.

PLAIN3D también consigue mejorar el rendimiento de la arquitectura estática Medium-End, pero esta mejora es cercana al 2% solamente. Al contrario que IMPROV3D, cuando un *core* entra en una fase de *spinning*, PLAIN3D no obtiene ningún beneficio, ya que en esos casos los recursos de los *threads* que están realizando trabajo útil son asignados a los *threads* en *spinning* debido a su comportamiento prácticamente perfecto en memoria y a que la lógica de predicción de saltos captura a la perfección su comportamiento. Lo anterior lleva a que muestre un tiempo de ejecución superior al Medium-End con las *PARSEC*. La compensación de lo anterior con otras fases de ejecución parece compensar el rendimiento, de tal modo que, prácticamente, iguala el rendimiento del Medium-End en el resto de aplicaciones.

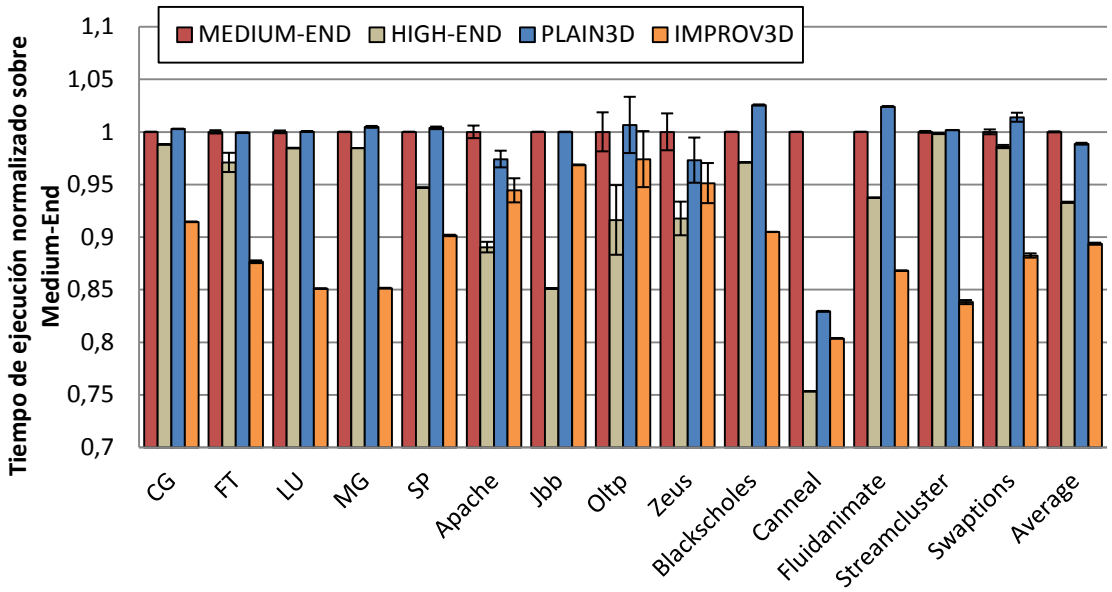

Figura 13 – Tiempo de ejecución normalizado de cargas de trabajo *multithreaded* en un *CMP* con 4 *cores*.

Aunque cuando el número de *cores* es mayor (Figura 14), la contención en la sincronización podría ser mayor, no se observa un rendimiento superior al mostrado en un *CMP* con cuatro *cores*. Esto está relacionado con el hecho de que el *CMP* de ocho *cores* tiene una pareja de cuatro *cores* apilados (Figura 2b), ya que si los *threads* están en el otro subconjunto no es posible utilizar estos recursos. Con un número mayor de capas, es decir, de *cores* apilados se podría solucionar esto, sin embargo implicaría ampliar el análisis físico y de coste de [20] a ocho *cores* apilados.

Para explicar los resultados anteriores, vamos a centrarnos en la aplicación *MG* sobre un *CMP* de cuatro *cores* ya que IMPROV3D es prácticamente un 15% más rápido que Medium-End en este caso particular. Atendiendo a la evolución del IPC útil (sin tener en cuenta las instrucciones de sincronización) (Figura 15) y a la evolución de la fracción de sincronización (Figura 16), vemos como todos los *cores* tienen que esperar al *core* 0 en los instantes correspondientes a 40 y 80 millones de ciclos.

Durante estos instantes, los recursos de los *cores* que están esperando se asignan al *core* 0 y éste aumenta su IPC de manera muy significativa de tal forma que se reduce el tiempo de espera de los otros *threads*.

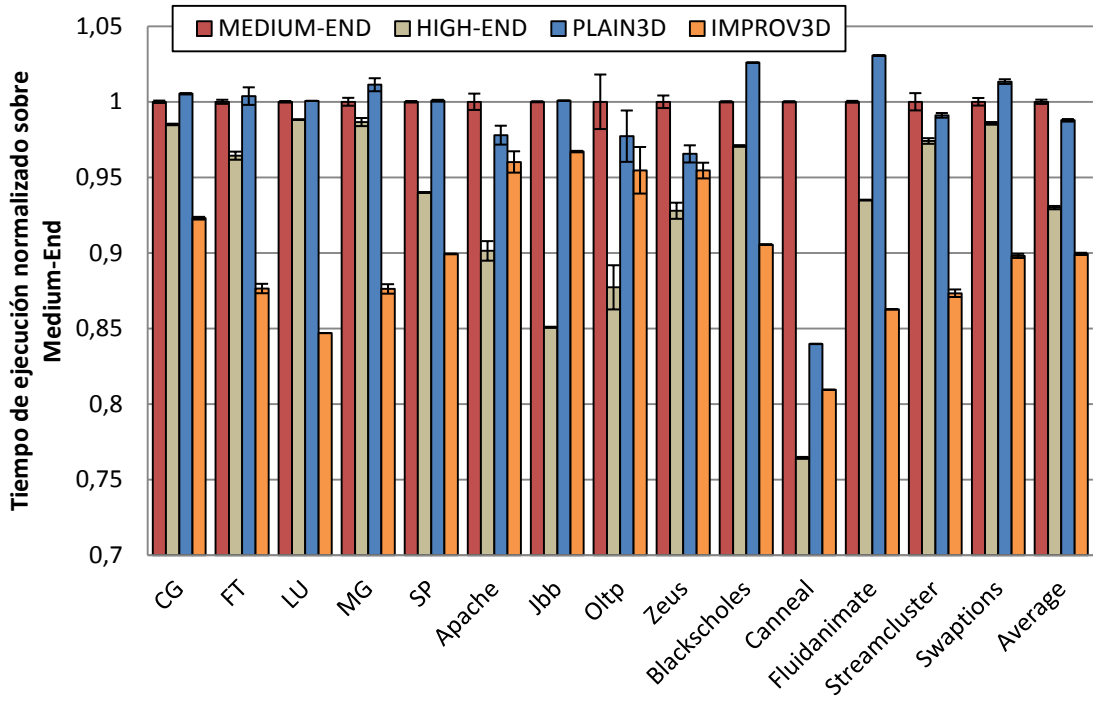

Figura 14 – Tiempo de ejecución normalizado de cargas de trabajo *multithreaded* en un *CMP* con 8 *cores*.

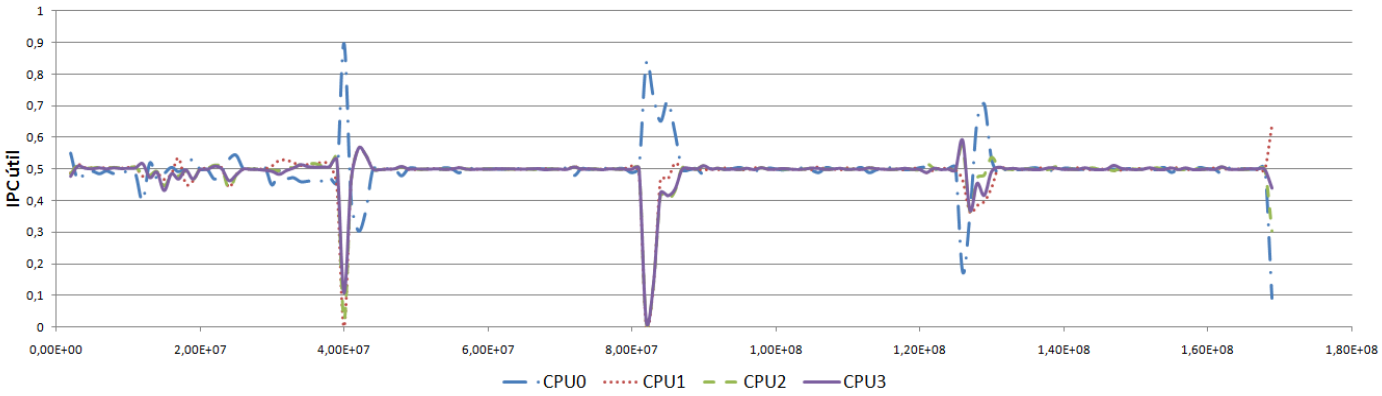

Figura 15 – Evolución del IPC útil para una iteración de *MG* en un *CMP* con 4 *cores*.

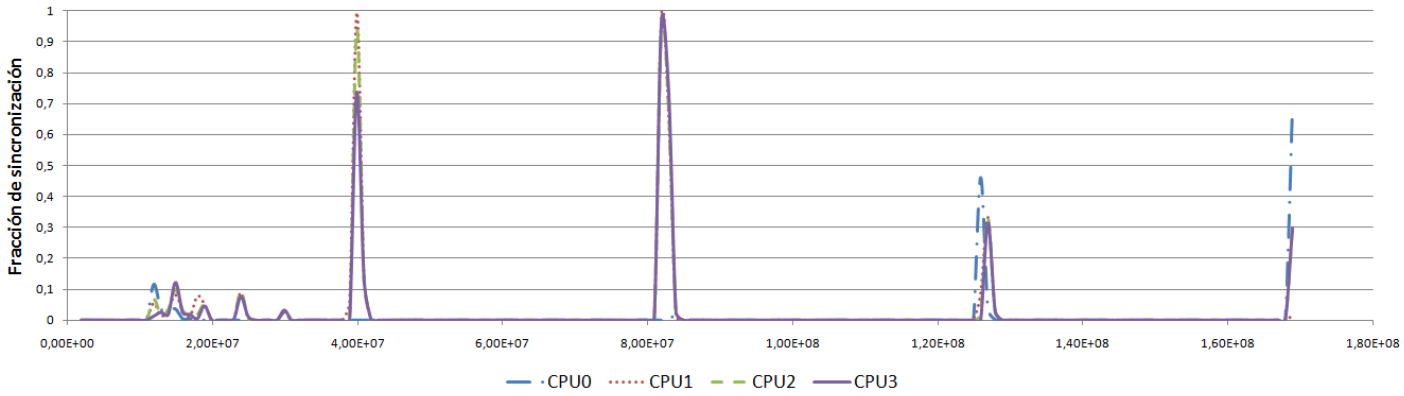

Figura 16 – Evolución de la fracción de sincronización para una iteración de *MG* en un *CMP* con 4 *cores*.

La eficiencia depende del mecanismo simple para la detección de *spinning* propuesto; para comprobarlo podemos comparar el número de instrucciones ejecutadas. Como podemos ver en la Figura 17, en la mayoría de los casos, hay una reducción en el número de instrucciones que finalizan su ejecución, lo que indica que la lógica toma buenas decisiones teniendo en cuenta que las instrucciones útiles tienen que ejecutarse de todas formas ya que medimos el tiempo requerido para realizar la misma cantidad de trabajo (el número de iteraciones en las aplicaciones numéricas y el número de transacciones en las aplicaciones transaccionales). La variabilidad en los resultados es diversa, con un rango que va desde valores cercanos al 30% en las *NPB* hasta el 0% en algunas aplicaciones transaccionales.

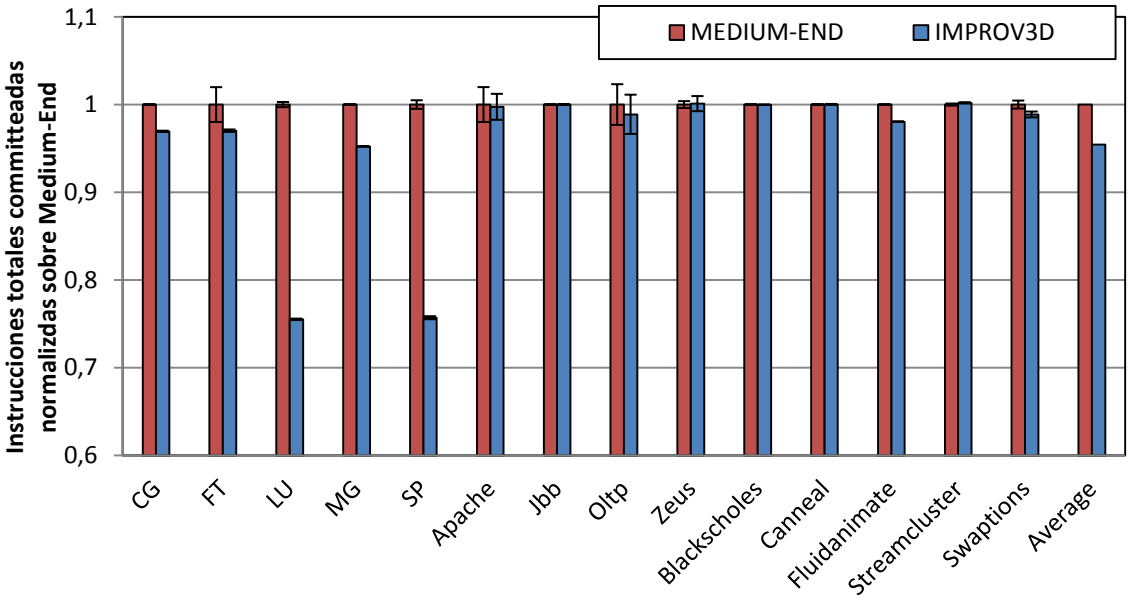

Figura 17 – Instrucciones totales *committeadas* para la arquitectura Medium-End.

4.4 – *Cores* simples

Por último, vamos a estudiar el rendimiento de nuestra arquitectura dinámica con apilamiento en 3D, reduciendo los recursos de la arquitectura Medium-End. Reducimos los recursos a la mitad, soportando 16 instrucciones en vuelo, reduciendo el número de vías a uno y reduciendo a la mitad el número de unidades funcionales. De manera pesimista, conservamos el número de

etapas del *pipeline*, un modelo más realista reduciría el número de etapas, y mantenemos la jerarquía de memoria. Denominamos esta arquitectura 1W-PROV3D.

Reproducimos los experimentos llevados a cabo en las subsecciones anteriores, simulando 1W-PROV3D tanto con cargas de trabajo multiprogramadas como con cargas de trabajo *multithreaded*.

El rendimiento en las cargas de trabajo multiprogramadas decae (Figura 18) con respecto al rendimiento mostrado por Medium-End, especialmente cuando todos los *cores* están ocupados (HYBRID 4THR). La diferencia se reduce cuando un *core* queda ocioso (RATE 3THR). Por último, cuando tres *cores* quedan ociosos (SINGLE 1THR), el rendimiento supera casi en un 10% al propio de Medium-End ya que, este caso, la arquitectura dinámica es, a efectos prácticos, la arquitectura High-End (64 instrucciones en vuelo, cuatro vías de ejecución,…) puesto que los recursos de los tres *cores* ociosos son utilizados por el único *thread* realizando trabajo útil. Las diferencias entre el rendimiento de 1W-PROV3D y High-End (ver Figura 12) tienen como causa las limitaciones de la propuesta, como es el *front-end* estático.

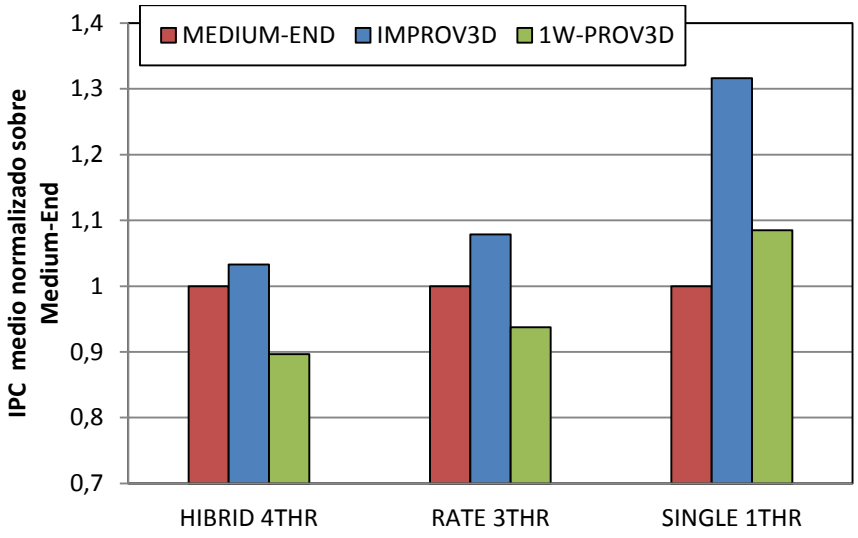

Figura 18 – IPC medio, normalizado sobre la arquitectura Medium-End, con *cores* simples.

Simulamos de nuevo las cargas de trabajo *multithreaded*, tanto utilizando un *CMP* de 4 *cores* (Figura 19) como con un *CMP* de 8 *cores* (Figura 20), al igual que hicimos con IMPROV3D.

Por un lado, en la mayoría de las aplicaciones donde la arquitectura High-End no obtiene un beneficio relevante como en las *NPB* o en algunas aplicaciones de *PARSEC*, 1W-PROV3D obtiene un mayor beneficio, lo que indica que es más importante acelerar los procesos de sincronización que sobredimensionar los *cores*. Por otro lado, en las cargas de trabajo de servidores como las *TRANS*, el rendimiento de 1W-PROV3D es significativamente peor donde High-End obtiene un buen rendimiento, lo que indica que en, este tipo de aplicaciones, sobredimensionar los *cores* es más beneficioso que acelerar los procesos de sincronización.

En resumen, 1W-PROV3D mejora ligeramente el rendimiento de *Medium-End*, con la mitad del área del procesador.

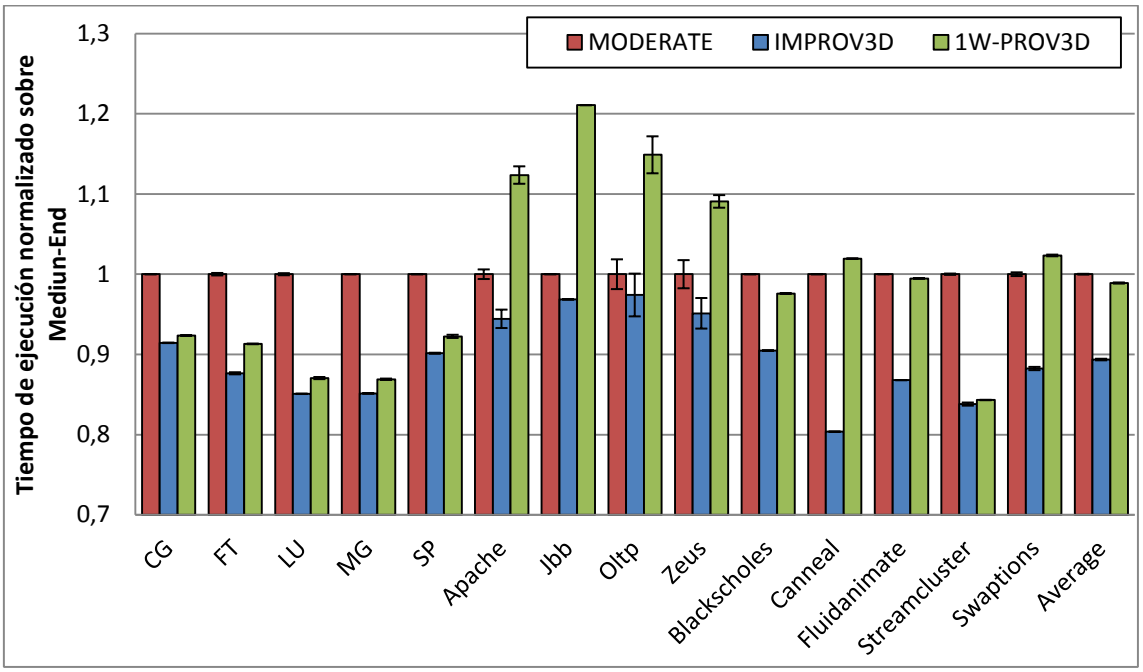

Figura 19 – Tiempo de ejecución, normalizado sobre Medium-End, con *cores* simples, en un *CMP* con 4 *cores*.

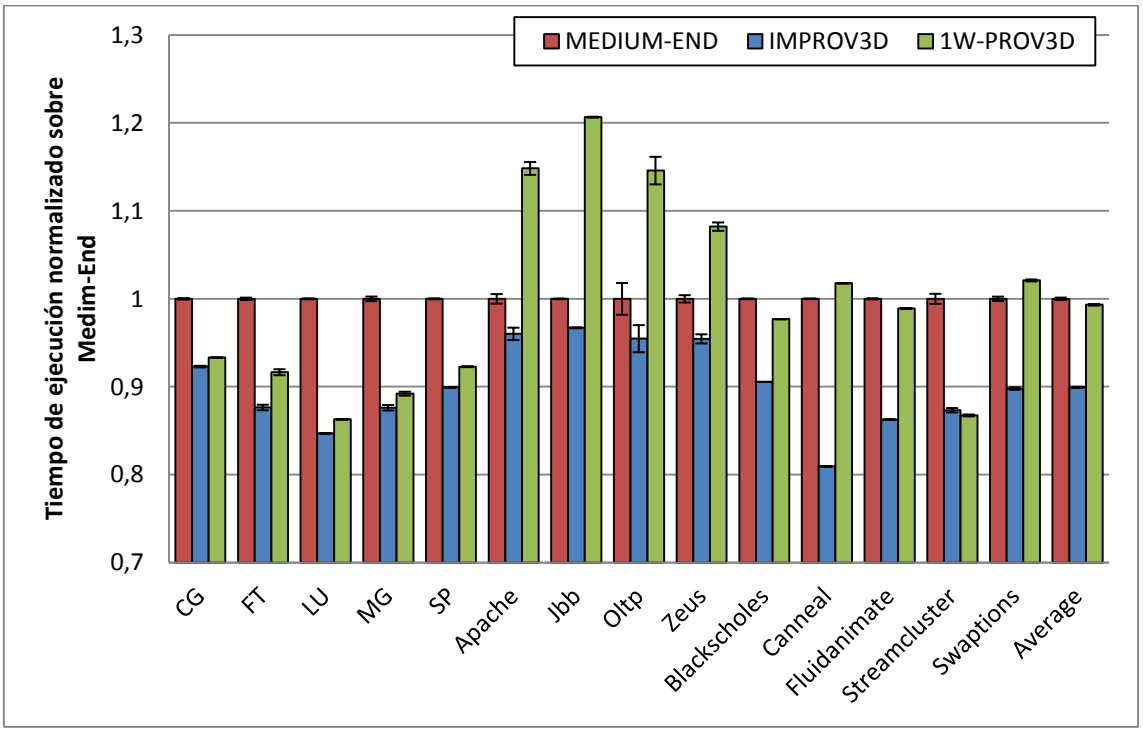

Figura 20 – Tiempo de ejecución normalizado sobre Medium-End con *cores* simples en un *CMP* con 8 *cores*.

Por último, resulta interesante comparar el comportamiento de 1W-PROV3D en un *CMP* con ocho *cores* con un *CMP* con cuatro *cores* Medium-End bajo área constante, esto es, doblamos el número de *cores* simples pero mantenemos constante el área de la jerarquía de *cache* (en ambos

casos 4MB para la *cache* L2, un solo controlador de memoria y dividimos por dos la *cache* de L1 de 1W-PROV3D), en resumen, ambos *CMPs* requieren el mismo área.

Los resultados (Figura 21) son positivos en la mayoría de los casos. En general, resulta más provechoso utilizar el doble de *cores* sencillos y la misma *cache*, la mayoría de las aplicaciones escalan bien, salvo las *PARSEC* donde algunas de las aplicaciones muestran un tiempo de ejecución superior al obtenido con el *CMP* de cuatro *cores*.

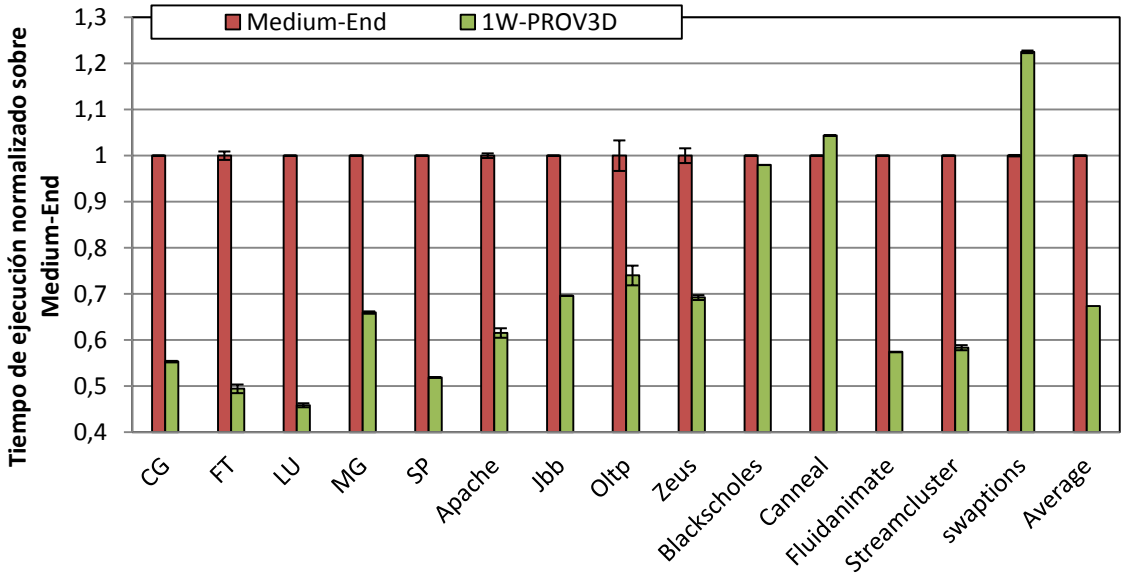

Figura 21 – Tiempo de ejecución normalizado de un *CMP* con 8 *cores* 1W-PROV3D sobre un *CMP* con 4 *cores* Medium-End bajo área constante.

Capítulo V
Conclusiones

En los últimos años los procesadores han sufrido un gran cambio, pasando de tener un solo *core* por chip a poseer varios replicados dentro del mismo. Asociado a este cambio ha surgido la ejecución paralela como método para obtener el máximo rendimiento dentro de este nuevo tipo de procesadores, sin embargo, no está exenta de problemas que limitan su rendimiento con las regiones secuenciales y la sincronización.

Con el objetivo de superar estos problemas se ha propuesto un procesador con microarquitectura dinámica, apilado verticalmente, que permite aprovechar mejor los recursos disponibles, ejecutando una mayor fracción de trabajo útil y como consecuencia mostrar un mejor rendimiento que la misma arquitectura estática.

La propuesta se ha evaluado con un número elevado cargas de trabajo de distinta índole, como aplicaciones multiprogramadas, donde no hay una interacción entre los distintos *threads* del sistema y aplicaciones *multithreaded* donde sí que hay una interacción en los *threads*. IMPROV3D ha mostrado un mayor rendimiento en ambos tipos de cargas de trabajo con rangos cercanos al Medium-End en algunos casos y llegando al 15% en otros casos.

Por último, se ha evaluado la misma propuesta pero utilizando *cores* más simples que utilizan la mitad del área de Medium-End e IMPROV3D y los resultados han sido positivos ya que, de manera general, supera ligeramente el rendimiento de Medium-End utilizando la mitad del área. Por último, al comparar un *CMP* con ocho cores sencillos 1W-PROV3D contra un *CMP* con cuatro *cores* Medium-End bajo al mismo área, se ha observado una mejora sustancial en el tiempo de ejecución.

Capítulo VI
Trabajos Futuros

El proyecto sigue el camino comenzado por [20] obteniendo mayores beneficios. Queremos seguir explorando las excitantes oportunidades que brinda el 3D *stacking*. Concretamente, sería interesante estudiar sistemas con más de cuatro capas apiladas e incluir la compartición dinámica del *front-end* del procesador ya que en este proyecto solo se comparte el *back-end* y puede ser una limitación importante.

Adicionalmente, el 3D *stacking* podría tener un mayor impacto en la jerarquía de memoria si la arquitectura de la *cache*, el protocolo de coherencia y la red de interconexión *on-chip* fuesen concebidas con la latencia y el ancho de banda acorde a las características del 3D *stacking*. Agrupando ambas cosas se podrían incrementar el rendimiento de la propuesta.

Adicionalmente, los resultados mostrados por los *cores* simples 1W-PROVED son muy interesantes ya que el rendimiento general es superior al mostrado por el caso base Medium-End con la mitad del área. Del mismo modo, la última prueba ha revelado que manteniendo el área constante y doblando el número de *cores* sencillos pueden obtenerse unos resultados muy significativos.

En definitiva, hay múltiples y prometedores caminos a seguir.

Bibliografía

[1] G. E. Moore, "Cramming More Components Onto Integrated Circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82-85, Jan. 1998.

[2] K. Olukotun and L. Hammond, "The future of microprocessors," *Queue*, vol. 3, no. 7, p. 26, Sep. 2005.

[3] W. Davis, J. Wilson, S. Mick, and J. Xu, "Demystifying 3D ICs: The pros and cons of going vertical," *Design & Test of Computers, IEEE*, pp. 498-510, 2005.

[4] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, "Core fusion," in *Proceedings of the 34th annual international symposium on Computer architecture - ISCA '07*, 2007, pp. 186-196.

[5] D. Gibson and D. A. Wood, "Forwardflow: a scalable core for power-constrained CMPs," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010, pp. 14–25.

[6] C. Kim et al., "Composable Lightweight Processors," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007, pp. 381-394.

[7] G. H. Loh and Y. Xie, "3D Stacked Microprocessor: Are We There Yet?," *IEEE Micro*, vol. 30, no. 3, pp. 60-64, May 2010.

[8] M. M. K. Martin et al., "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 99-107, 2005.

[9] P. S. Magnusson et al., "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50-58, 2002.

[10] A. R. Alameldeen et al., "Simulating a $2M commercial server on a $2K PC," *Computer*, vol. 36, no. 2, pp. 50-57, Feb. 2003.

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, no. January, pp. 72–81.

[12] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance," *NASA Ames Research Center," Technical Report NAS-99-011*, Citeseer, 1999.

[13] V. Standard Performance Evaluation Corporation, SPEC*, http://www.spec.org, Warrenton, "SPEC 2006." .

[14] M. A. Ramirez, A. Cristal, A. V. Veidenbaum, L. Villa, and M. Valero, "A new pointer-based instruction queue design and its power-performance evaluation," in *2005 International Conference on Computer Design*, pp. 647-653.

[15] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, “A NUCA substrate for flexible CMP cache sharing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 8, pp. 1028-1040, 2007.

[16] M. Martin and M. Hill, “Token Coherence: a New Framework for Shared-Memory Multiprocessors,” *Micro, IEEE*, pp. 108-116, 2004.

[17] A. N. Eden and T. Mudge, “The YAGS branch prediction scheme,” *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 69-77.

[18] J. Kim et al., “A novel dimensionally-decomposed router for on-chip communication in 3D architectures,” in *Proceedings of the 34th annual international symposium on Computer architecture - ISCA ’07*, 2007, vol. 35, no. 2, p. 138.

[19] M. Suleman, O. Mutlu, M. Qureshi, and YN, “Accelerating critical section execution with asymmetric multi-core architectures,” in *Proceeding of the 14th*, 2009, pp. 253-264.

[20] H. Homayoun, T.-wei Lin, D. M. Tullsen, V. Kontorinis, and A. Shayan, “Dynamically heterogeneous cores through 3D resource pooling,” in *IEEE International Symposium on High-Performance Comp Architecture*, 2012, pp. 1-12.

[21] D. Ponomarev, G. Kucuk, and K. Ghose, “Dynamic resizing of superscalar datapath components for energy efficiency,” *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 199-213, Feb. 2006.

[22] A. Karandikar and K. K. Parhi, “Low power SRAM design using hierarchical divided bit-line approach,” in *Computer Design: VLSI in Computers and Processors, 1998. ICCD’98. Proceedings. International Conference on*, 1998, pp. 82–88.

[23] K. Puttaswamy and G. H. Loh, “Implementing Register Files for High-Performance Microprocessors in a Die-Stacked (3D) Technology,” in *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI’06)*, 2006, vol. 00, pp. 384-392.

[24] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi, “Reducing the complexity of the register file in dynamic superscalar processors,” in *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, pp. 237-248.

[25] C. Tian, V. Nagarajan, R. Gupta, and S. Tallam, “Automated dynamic detection of busy-wait synchronizations,” *Software: Practice and Experience*, vol. 39, no. 11, pp. 947-972, Aug. 2009.

[26] T. Li, A. Lebeck, and D. J. Sorin, “Spin detection hardware for improved management of multithreaded systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 6, pp. 508-521, Jun. 2006.

[27] M. Richard and M. Jim, *Solaris Internals: Solaris 10 and Open Solaris Kernel Architecture*. Prentice Hall, 2006.

[28] K. Chakraborty, P. M. Wells, and G. S. Sohi, “Supporting Overcommitted Virtual Machines through Hardware Spin Detection,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 2, pp. 353-366, Feb. 2012.

[29] P. M. Wells, K. Chakraborty, and G. S. Sohi, “Hardware support for spin management in overcommitted virtual machines,” in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques - PACT ’06*, 2006, pp. 124-132.

Acrónimos

3DDM 3D Stacked Dynamic Micro-architecture

ACS Accelerating Critical Sections

API Application Programming Interface

BCT Backward Control Transfer

CMP Chip multiprocessor

CPU Central Processing Unit

FP Floating Point

FU Functional Unit

ILP Instruction Level Parallelism

INT Integer

IPC Instrucciones por ciclo

IQ Instruction queue

LLC Last-level cache

LSQ Load-Store Queue

PARSEC Princeton Application Repository for Shared-Memory Computers

PRF Physical Register File

ROB Reorder buffer

SDM Spinning Detection Module

SPEC Standard Performance Evaluation Corporation

SPH Smoothed Particle Hydrodynamics

TLB Translation Lookaside Buffer

TSV Through silicon via