Proyecto Fin de Máster en
Ingeniería de Computadores.
Curso 2009-2010.

Estimación de Factores de Ganancia en Sistemas Asimétricos.

Autor:
Fermín Ayuso Márquez

Director:
Manuel Prieto Matías

Director externo de dirección:
Juan Carlos Sáez Alcaide

Máster en Investigación en Informática.
Facultad de Informática.
Universidad Complutense de Madrid.

A mis padres y hermanos.
A todos los que me han acompañado en este camino.

Agradecimientos

Quisiera agradecer al director del proyecto, Manuel Prieto Matías, y a Juan Carlos Sáez Alcaide, del Departamento de Arquitectura de Computadores y Automática, su dedicación, atención y colaboración para que este proyecto haya podido salir adelante. Sin su inestimable ayuda no hubiera sido posible.

Autorización

El abajo firmante, matriculado en el Máster en Investigación Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: *Estimación de Factores de Ganancia en Sistemas Asimétricos*, realizado durante el curso académico 2009-2010 bajo la dirección de Manuel Prieto Matías y con la colaboración externa de dirección de Juan Carlos Sáez Alcaide en el Departamento de Arquitectura de Computadores Y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo

Fermín Ayuso Márquez

Resumen

Los procesadores multicore asimétricos con repertorio de instrucciones común (ASISA: *asymmetric single-ISA multicore processors*), han sido propuestos como una alternativa más eficiente en términos de consumo-rendimiento a las arquitecturas multicore homogéneas convencionales. Un procesador ASISA consta de varios *cores* con diferentes capacidades, pero que a diferencia de otras arquitecturas heterogéneas como el CELL BE, son capaces de ejecutar todos ellos el mismo repertorio de instrucciones, o al menos un conjunto común amplio, simplificándose con ello el desarrollo software, uno de los principales obstáculos para la adopción de arquitecturas heterogéneas a una escala más global.

No existen todavía procesadores comerciales con una arquitectura ASISA y las propuestas que han ido surgiendo han sido diversas. Muchas de estas se han inspirado en el análisis de cargas de trabajo multiprogramadas compuestas por aplicaciones de naturaleza diversa. Con dicha diversidad, un sistema multicore asimétrico puede proporcionar potencialmente un mayor rendimiento por vatio que uno homogéneo. La clave está en asignar cada proceso (o thread) al core donde globalmente se obtiene el mejor balance consumo-rendimiento.

En este trabajo nos hemos centrado en estudiar como es posible inferir el Factor de Ganancia, es decir, el beneficio o deterioro relativo que puede obtener una aplicación al ejecutarse en un determinado tipo de *core* (lento o rápido), a partir de medidas de rendimiento de esa aplicación obtenidas con contadores hardware en el core en el que se está ejecutando actualmente.

Palabras clave: Procesadores multicore asimétricos, arquitecturas heterogéneas, Factor de Ganancia, contadores hardware, k-means.

Abstract

Asymmetric multicore processors with a common instruction set (ASISA: *asymmetric single-ISA multicore processors*), have been proposed as a more efficient alternative in terms of consumption-performance against conventional homogeneous multicore architectures. An ASISA processor consists of multiple cores with different capacities, but unlike other heterogeneous architectures such as the CELL BE, are capable of performing all the same instruction set, or at least a wide common set, simplifying the software development, one of the main obstacles to the adoption of heterogeneous architectures to a more global scale.

There are still no commercial processors with an architecture ASISA and proposals that have emerged have been different. Many of these proposals have been inspired by the analysis of different kinds of multi-program workloads applications. With such diversity, an asymmetric multicore system can potentially provide higher performance per watt than a homogeneous one. The key is to assign each process (or thread) to the core where you get the best overall consumption/performance balance.

In this work we have focused on studying how is it possible to estimate the speedup factor, the improvement or slowdown that a given application may experience when runnin on a certain core type, based on performance measures of that application obtained with hardware counters on the core type the application currently running on.

Key words: Asymetric Multicore Processor, speedup factor, hardware counters, k-means, heterogeneous architectures.

Índice

1. Introducción **12**
1.1. Planificador PA (Parallelism-Aware) en Multicore Asimétricos 13
1.2. Planificación HASS (Het-Aware Signature-Supported) en Multicore Asimétricos 15

2. Estimación de Factores de Ganancia en Sistemas Asimétricos **18**
2.1. Contadores Hardware 18
2.2. SPEC 2006 20
2.2.1. Descripción de los *benchmark* de CPU2006 20
2.3. Clustering con K-means 24
2.3.1. K-means y MATLAB® 25

3. Resultados **31**
3.1. Resultado de las medidas obtenidas a partir de los contadores hardware . . . 32
3.1.1. Medidas obtenidas para CINT2006 32
3.1.2. Medidas obtenidas para CFP2006 40
3.2. Clasificación realizada con *k-means* 48
3.2.1. Clasificación con *k-means* para CINT2006 haciendo uso de todas las medidas 49
3.2.2. Clasificación con *k-means* para CINT2006 con un conjunto reducido de medidas 53
3.2.3. Clasificación con *k-means* para CFP2006 con todas las medidas . . . 56
3.2.4. Clasificación con *k-means* para CFP2006 con un conjunto de las medidas obtenidas 60

4. Conclusiones y Trabajo Futuro **64**

Índice de figuras

1. Arquitectura homogénea (izquierda) frente a arquitectura asimétrica (derecha). 12
2. Clases de aplicaciones en el planificador PA. 14
3. Factores de ganancia observados frente a factores de ganancia estimados con las firmas estáticas en dos plataformas asimétricas emuladas con procesadores de Intel (izquierda) y AMD (derecha) utilizando los mecanismos DVFS para fijar los cores a distinta frecuencia. Aparecen etiquetados algunos benchmarks en los que existe discrepancia. Si existiese una precisión perfecta en la estimación todos los puntos aparecerían en la diagonal. 16
4. Contadores hardware para diferentes CPU's 18
5. 1) los centroides iniciales (para este caso, k=3) se seleccionan de forma aleatoria para el conjunto de datos. 2) Los k clusters se crean asociando cada dato del conjunto con su centroide más cercano. Las particiones en este caso representan el diagrama de Voronoi generado por los centroides. 3) El centroide de cada cluster se convierte en el nuevo medio. 4) Los pasos 2 y 3 se repiten hasta que el algoritmo converge . . 25
6. Representación de una matriz con $n=2$ y $N=7$ con dos clusters representativos 26
7. Representación de la matriz X en Matlab 27
8. Significado de cada componente de la matriz d anteriormente descrita 27
9. Resultado en Matlab para una matriz de datos z con sus centroides. Los triángulos representan los datos y los cuadrados los centroides 30
10. Mejora al ejecutar los *benchmark* de enteros en las arquitecturas Intel Atom e Intel Core2 con distintas frecuencias 31
11. Mejora al ejecutar los *benchmark* de punto flotante en las arquitecturas Intel Core2 con diferentes frecuencias e Intel Atom 32
12. valores obtenidos para el IPC en el Intel Atom y en el Core2 con diferentes frecuencias para el conjunto de benchmarks pertenecientes a CINT2006. . . . 33
13. valores obtenidos para el número de instrucciones de salto por cada mil instrucciones en el Intel Atom y en el Core2 con diferentes frecuencias para el conjunto de benchmarks que pertenecen a CINT2006. 34
14. valores obtenidos para el número de accesos al último nivel de cache por cada mil instrucciones en el Intel Atom y en el Core2 con diferentes frecuencias para el conjunto de benchmarks que pertenecen a CINT2006. 35
15. valores obtenidos para el número de fallos en el último nivel de caché por cada mil instrucciones en el Intel Atom y en el Core2 con diferentes frecuencias para el conjunto de benchmarks que pertenecen a CINT2006. 36
16. valores para el número de saltos mal predichos por cada mil instrucciones en el Intel Atom y en el Core2 con diferentes frecuencias para el conjunto de benchmarks que pertenecen a CINT2006. 37
17. valores obtenidos para el número de accesos al último nivel de cache por ciclo el Intel Atom y en el Core2 con diferentes frecuencias para el conjunto de benchmarks que pertenecen a CINT2006. 37

18. valores obtenidos para el número de fallos en el último nivel de caché por ciclo en el Intel Atom y en el Core2 con diferentes frecuencias para el conjunto de benchmarks que pertenecen a CINT2006. 38
19. valores obtenidos para el número de fallos en ITLB por millón de instrucciones en el Intel Atom y en el Core2 con diferentes frecuencias para el conjunto de benchmarks que pertenecen a CINT2006. 39
20. valores obtenidos para el número de fallos en DTLB por millón de instrucciones en el Intel Atom y en el Core2 con diferentes frecuencias para el conjunto de benchmarks que pertenecen a CINT2006. 39
21. porcentaje de instrucciones en punto flotante para el Intel Atom y en el Core2 con diferentes frecuencias para el conjunto de benchmarks que pertenecen a CINT2006. 40
22. Instrucciones por ciclo para los *benchmarks* de punto flotante. 41
23. Valores obtenidos para *benchmarks* de punto flotante teniendo en cuenta el número de instrucciones de salto por cada mil instrucciones. 41
24. Número de accesos al último nivel de cache para los *benchmarks* de punto flotante pertenecientes a CFP2006. 42
25. Número de fallos en el último nivel de cache para los *benchmarks* de punto flotante pertenecientes a CFP2006. 43
26. Valores obtenidos al calcular el número de saltos mal predichos por cada mil instrucciones en el Atom y en el Core2 para los *benchmarks* de punto flotante pertenecientes a CFP2006. 44
27. Número de accesos a LLC por ciclo en el Atom y en el Core2 para los *benchmarks* de punto flotante pertenecientes a CFP2006. 45
28. Número de fallos en LLC por ciclo en el Atom y en el Core2 para los *benchmarks* de punto flotante pertenecientes a CFP2006. 45
29. Número de fallos en ITLB por millón de instrucciones en el Atom y en el Core2 para los *benchmarks* de punto flotante pertenecientes a CFP2006. . . 46
30. Número de fallos en DTLB por millón de instrucciones en el Atom y en el Core2 para los *benchmarks* de punto flotante pertenecientes a CFP2006. . . 47
31. Porcentaje de instrucciones en punto flotante para los *benchmarks* pertenecientes a CFP2006 en el Atom y en el Core2. 48
32. Resultados obtenidos al comparar los grupos creados con las medidas tomadas en el Atom y con los del Core2 a 1.86Ghz para un valor de k=3 utilizando todas las medidas descritas en la sección 2.1. Entre paréntesis se muestran las discrepancias entre grupos, que para este caso, no existen. En la figura de la derecha, el eje de abcisas representa los grupos y el de ordenadas el speedup de cada uno de ellos. 49
33. Resultados obtenidos al comparar los grupos creados con las medidas obtenidas a partir de los contadores hardware del Core2 a 1.86Ghz y del Atom para un valor de k=4 50
34. Resultados obtenidos al comparar los grupos creados con las medidas obtenidas en el Atom y con los del Core2 a 1.86Ghz para CINT2006 y un valor de k=5 y donde se pueden observar las discrepancias 51

35. Resultados obtenidos al comparar los grupos creados con las medidas obtenidas en el Atom y con los del Core2 a 1.60Ghz para un valor de k=3 y donde se pueden observar que no existen discrepancias 52
36. Resultados obtenidos al comparar los grupos creados con las medidas para el Atom y el Core2 a 1.60Ghz para valores de k=4 y k=5 y donde se pueden observar que empiezan a existir discrepancias entre los grupos 53
37. Grupos creados para el Atom y el Core2 a 1.86Ghz y los speedup medios de cada uno de estos grupos con un valor de k=3. 54
38. Grupos creados por *k-means* con k=4 para el Atom y el Core2 a 1.86Ghz y los speedup medios de cada uno de estos grupos. 54
39. Grupos creados para el Atom y el Core2 a 1.86Ghz y los speedup medios de cada uno de estos grupos considerando un valor de k=5. 55
40. Grupos creados para el Atom y el Core2 a 1.6Ghz y los speedup medios de cada uno de estos grupos considerando un valor de k=3. 55
41. Grupos creados para el Atom y el Core2 a 1.6Ghz y los speedup medios de cada uno de estos grupos con un valor de k=4. 56
42. Grupos creados por *k-means* con k=5 para el Atom y el Core2 a 1.6Ghz y los speedup medios de cada uno de estos grupos. 56
43. Grupos creados por *k-means* con k=3 para el Atom y el Core2 a 1.86Ghz y los speedup medios de cada uno de estos grupos para los *benchmarks* pertenecientes a CFP2006. 57
44. Grupos creados por *k-means* con k=3 para el Atom y el Core2 a 1.6Ghz y los speedup medios de cada uno de estos grupos para los *benchmarks* pertenecientes a CFP2006. 58
45. Grupos creados por *k-means* con k=4 para el Atom y el Core2 a 1.86Ghz y los speedup medios de cada uno de estos grupos para los *benchmarks* pertenecientes a CFP2006. 58
46. Grupos creados para el Atom y el Core2 a 1.6Ghz y los speedup medios de cada uno de estos grupos con un valor de k=4 para los *benchmarks* pertenecientes a CFP2006. 59
47. Grupos creados por *k-means* con k=5 para el Atom y el Core2 a 1.86Ghz y los speedup medios de cada uno de estos grupos para los *benchmarks* pertenecientes a CFP2006. 59
48. Grupos creados por *k-means* con k=5 para el Atom y el Core2 a 1.60Ghz y los speedup medios de cada uno de estos grupos para los *benchmarks* pertenecientes a CFP2006. 60
49. Grupos creados para el Atom y el Core2 a 1.86Ghz y los speedup medios de cada uno de estos grupos con un valor de k=3 para los *benchmarks* pertenecientes a CFP2006. 61
50. Grupos creados para el Atom y el Core2 a 1.60Ghz y los speedup medios de cada uno de estos grupos con un valor de k=3 para los *benchmarks* pertenecientes a CFP2006. 61

51. Resultados obtenidos al comparar los grupos creados con el conjunto de medidas obtenidas en el Core2 a 1.86Ghz y con los del Atom para un valor de k=4 para CFP2006 62
52. Resultados obtenidos al comparar los grupos creados en el Core2 a 1.60Ghz y con los del Atom para un valor de k=4 para CFP2006 62
53. Resultados obtenidos al comparar los grupos creados en el Core2 a 1.86Ghz y con los del Atom para un valor de k=5 para CFP2006 63
54. Resultados obtenidos al comparar los grupos creados en el Core2 a 1.60Ghz y con los del Atom para un valor de k=5 para CFP2006 63

1. Introducción

Los procesadores multicore asimétricos con repertorio de instrucciones común (ASISA: *asymmetric single-ISA multicore processors*) [22], han sido propuestos como una alternativa más eficiente en términos de consumo-rendimiento a las arquitecturas multicore homogéneas convencionales. Un procesador ASISA consta de varios *cores* con diferentes capacidades, pero que a diferencia de otras arquitecturas heterogéneas como el CELL BE [14], son capaces de ejecutar todos ellos el mismo repertorio de instrucciones, o al menos un conjunto común amplio [24], simplificándose con ello el desarrollo software, uno de los principales obstáculos para la adopción de arquitecturas heterogéneas a una escala más global.
No existe todavía ningún procesador comercial con una arquitectura ASISA y las propuestas que han ido surgiendo han sido diversas. Los *cores* podrían diferir en diversas características microarquitectónicas, desde la frecuencia de los *cores* a cambios más profundos en las capacidades de las jerarquías de memoria interna o los mecanismos de planificación y ejecución de instrucciones. En cualquier caso, la característica esencial de este tipo de arquitecturas es la posibilidad de disponer, en un mismo procesador, de *cores* con distintos balances consumo-rendimiento en función del tipo de aplicaciones que ejecuten.
Muchas de las propuestas se han inspirado en el análisis de cargas de trabajo multiprogramadas compuestas por aplicaciones de naturaleza diversa. Por ejemplo, podría tratarse de aplicaciones con distinta intensidad de acceso a memoria o distinto grado de paralelismo a nivel de instrucción. Con dicha diversidad, un sistema multicore asimétrico puede proporcionar potencialmente un mayor rendimiento por vatio que uno homogéneo. La clave está en asignar cada proceso (o thread) al core donde globalmente se obtiene el mejor balance consumo-rendimiento. Consideremos como modelo un sistema asimétrico con dos tipos de *cores*: *rápidos*, con una frecuencia de reloj elevada, lanzamiento superescalar, ejecución fuera de orden y *lentos* con una frecuencia de reloj menor y lanzamiento escalar en orden. En este sistema, los procesos que ejecuten códigos cuyo rendimiento se vea limitado por operaciones de acceso a memoria debería asignarse típicamente a los *cores* lentos, ya que la ganancia (el speedup) que puede obtenerse ejecutando dichos procesos en los *cores* rápidos es relativamente menor que el consumo de energía adicional que supondría. Esta eficiencia energética de los sistemas heterogéneos ha quedado demostrada en numerosos estudios [7, 22, 21, 23, 27].

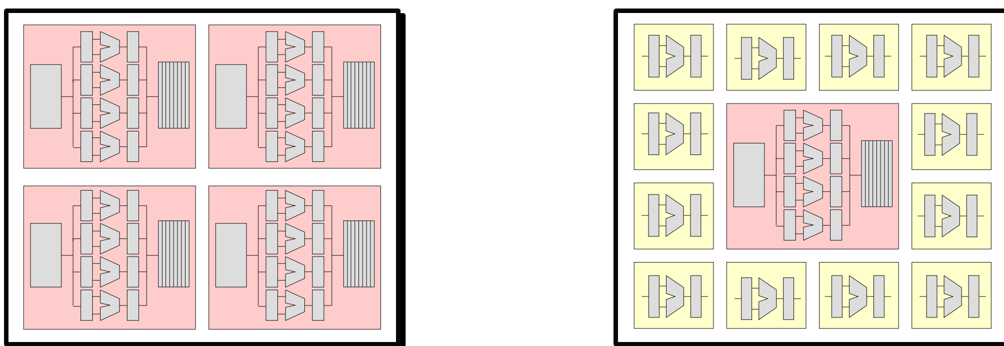


Figura 1: Arquitectura homogénea (izquierda) frente a arquitectura asimétrica (derecha).

Otra potencial ventaja de las arquitecturas ASISA la encontramos con cargas de trabajo donde se combinan aplicaciones escalables con otras en las que existen problemas para explotar paralelismo a nivel de thread. Actualmente están en pleno auge las arquitecturas multicore construidas mediante la integración de múltiples *cores* simples de consumo reducido, estimándose que el número de *cores* por chip se incrementará en los próximos años de forma sostenida con cada nueva generación tecnológica. Estas arquitecturas ofrecen una gran oportunidad para las aplicaciones paralelas, ya que con cada nueva generación su rendimiento se incrementará proporcionalmente con el número de *cores*. Sin embargo, las aplicaciones secuenciales o las aplicaciones paralelas que sólo pueden escalar hasta un determinado número de *cores* no obtendrán beneficios significativos[1]. Los procesadores asimétricos integrados por un número considerable de *cores* simples, potencialmente lentos pero con un consumo reducido, y por unos pocos *cores* más complejos, más rápidos aunque con mayor consumo energético, como el que se ilustra a la derecha de la Figura 1, pueden ayudar a mitigar estos problemas. Potencialmente, en un procesador ASISA sería posible la ejecución de las aplicaciones secuenciales, o las fases secuenciales de las aplicaciones paralelas, en los *cores* rápidos y las paralelas en los lentos. Se podría ofrecer así, de una forma transparente a los programadores siempre que el software del sistema sea capaz de gestionarlo automáticamente, lo mejor de ambos mundos: escalabilidad y bajo consumo de energía para aplicaciones paralelas, así como un excepcional rendimiento para aplicaciones secuenciales [16].
En ambos escenarios, la eficiencia de los sistemas asimétricos se maximiza claramente cuando las aplicaciones se asignan a los *cores* en función de las propiedades de ambos. Para evitar complejidad en el desarrollo del software, dicha asignación debería realizarse por el planificador del sistema operativo, utilizando algoritmos que sean conscientes de la heterogeneidad. En este tema ha venido trabajando nuestro grupo de investigación durante los últimos años. Entre las propuestas que hemos presentando están los planificadores HASS [12] y PA [32], que describimos brevemente en las siguientes subsecciones.

1.1. Planificador PA (Parallelism-Aware) en Multicore Asimétricos

Este planificador busca repartir los *cores* rápidos y lentos entre las aplicaciones basándose en su grado de paralelismo a a nivel de thread. Para ello se intenta [32]:

1. Distribuir la carga de trabajo de modo que los threads de las aplicaciones secuenciales inviertan más tiempo en los *cores* rápidos que los de las aplicaciones paralelas.

2. Migrar dinámicamente a los *cores* rápidos aquellos threads de aplicaciones paralelas que entren en una fase secuencial, acelerándolas de forma efectiva y mitigando en cierta medida los cuellos de botella de escalabilidad que presenten las aplicaciones.

3. Ofrecer buena escalabilidad para aplicaciones que hagan uso de muchos *cores*.

[1] A pesar de los avances en los distintos ámbitos del paralelismo, muchas aplicaciones seguirán siendo poco escalables por distintos motivos. El alto coste asociado a la paralelización de software [10] sigue siendo elevado, muchos algoritmos son de naturaleza puramente secuencial, y técnicas para la aceleración de aplicaciones mono-hilo en procesadores *multicore* como la especulación multi-hilo [35] tiene limitaciones.

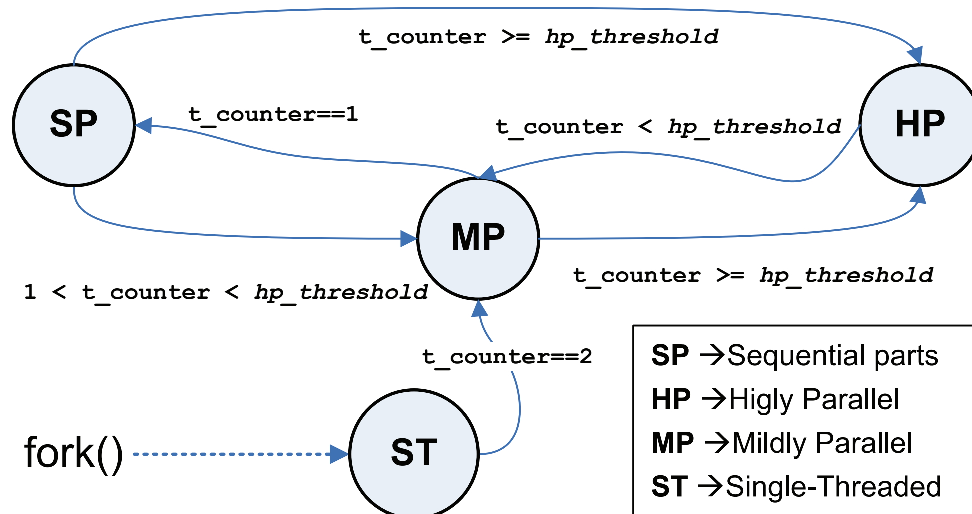


Figura 2: Clases de aplicaciones en el planificador PA.

Para alcanzar estos objetivos, el planificador divide explícitamente las colas de planificación en dos **particiones** o conjuntos de cores, FAST y SLOW, en las que se incluyen los cores rápidos y lentos presentes en la plataforma, respectivamente. La partición FAST se usa principalmente para ejecutar aplicaciones secuenciales y con nivel de paralelismo medio, así como las fases secuenciales de las aplicaciones paralelas. Por el contrario, la partición SLOW se destina a la ejecución de aplicaciones que posean un nivel de paralelismo elevado.
Del mismo modo, las aplicaciones y sus respectivos threads se asignan a **clases** que determinan la partición donde el thread se ejecutará. El procedimiento de asignación de clases se ilustra en el diagrama de transición de la Figura 2. Una aplicación recién creada se asigna a la clase ST (*Single-Threaded*) tras crear su primer thread, pero si el número de threads activos crece transitará a la clase MP (*Mildly Parallel*) – clase destinada a aplicaciones con nivel de paralelismo intermedio –. Si el número de thread activos de una aplicación supera un cierto umbral configurable – *hp_threshold* – , la aplicación se asignará a la clase HP (*Highly Parallel*). De forma análoga, cuando el número de threads activos de una aplicación HP disminuye por debajo de dicho umbral transita a la clase MP. Para las aplicación paralelas que se quedan con un único thread activo se reserva la clase SP (*Secuential Part*) –clase que representa las fases secuenciales de las aplicaciones paralelas–
La **asignación inicial** de threads a cores se efectúa en PA manteniendo un equilibrio en la carga del sistema y priorizando el *llenado* de la partición FAST con respecto a la SLOW. Adicionalmente se efectúan migraciones posteriores entre ambas particiones para asignar los cores a aplicaciones teniendo en cuenta su grado de paralelismo. Dichas migraciones se realizan cuidadosamente para evitar el desequilibrio en la carga. Supongamos que el thread T debe migrar de una partición a otra. Una solución simple es insertar el thread en una cola en ejecución de cualquier *core* de la partición objetivo. Esto, sin embargo, puede producir

desequilibrios si hay un gran número de migraciones yendo en una única dirección. Para evitarlo, nunca se hace una migración de la partición SLOW a la FAST a menos que haya un **thread candidato** para migrar de la partición FAST a la SLOW (o viceversa). Con estos **intercambios** es evidente que se preserva el equilibrio. No obstante, pueden ser necesarias migraciones adicionales cuando el cambio en el número de threads activos de una aplicación implique un desequilibrio en la carga de las particiones.
Una cuestión abierta del planificador PA es como puede detectar el sistema operativo fases secuenciales (o con un grado bajo de paralelismo) en las aplicaciones. En muchas ocasiones los threads no usados se bloquean durante dichas fases y el sistema operativo puede detectarlas simplemente monitorizando el número de threads activos de la aplicación. Sin embargo, en otras ocasiones, los threads no usados permanecen realizando algún tipo de espera activa y la monitorización del número de threads activos no permite la detección. Por lo tanto, para exponer estas fases de poco paralelismo al sistema operativo, la única alternativa viable que consideramos es aumentar la coordinación entre el sistema operativo y *runtime* de la infraestructura de paralelización utilizada (librería de threads, librería de paso de mensajes,...), para que se notifique al sistema operativo cuando un thread esta realizando algún tipo de espera activa.

1.2. Planificación HASS (Het-Aware Signature-Supported) en Multicore Asimétricos

Este planificador está basado en la idea de utilizar firmas arquitectónicas para caracterizar aplicaciones. Una firma arquitectónica (*architectural signature*) es un resumen compacto de las propiedades particulares de una aplicación que pueda interpretarse de forma rápida por el planificador para cuantificar como se adapta una aplicación a un determinado tipo de *core*. Puede contener por ejemplo información sobre el patrón de acceso a memoria utilizado, el grado de paralelismo a nivel de instrucción que podría explotarse o información sobre la sensibilidad de la aplicación a variaciones en la frecuencia de reloj del procesador. Con esta información se estiman los beneficios relativos que puede obtener las distintas aplicaciones si se asignan a los *cores* rápidos en lugar de a los lentos y que cuantificamos en lo que denominamos **Factor de Ganancia (Speedup Factor)**. En función de los Factores de Ganancia de todas las aplicaciones listas para ejecución el planificador realiza la asignación a *cores* rápidos o lentos que mayores beneficios potenciales ofrece a nivel global [34, 12, 33].
En [33] se evaluó HASS utilizando firmas arquitectónicas diseñadas específicamente para sistemas asimétricos cuyos *cores* diferían exclusivamente en su frecuencia. Se limitó el estudio a este tipo de sistemas ya que son fáciles de emular con procesadores multicore actuales fijando explícitamente las frecuencia de cada tipo de core gracias a los mecanismos DVFS disponibles en los procesadores actuales. Para capturar la sensibilidad a la frecuencia de las aplicaciones, las firmas utilizadas en estos sistemas caracterizan la *intensidad en memoria* [13] de las aplicaciones, ya que intuitivamente, una aplicación con un elevado ratio de accesos a memoria es probable que se vea afectada por frecuentes paradas del pipeline y la frecuencia del *core* no tenga un efecto significativo en el rendimiento. La intensidad en memoria puede aproximarse por la tasa de fallos de cache [37], y a partir de esta medida se puede estimar los Factores de Ganancia utilizando un modelo de rendimiento sencillo.

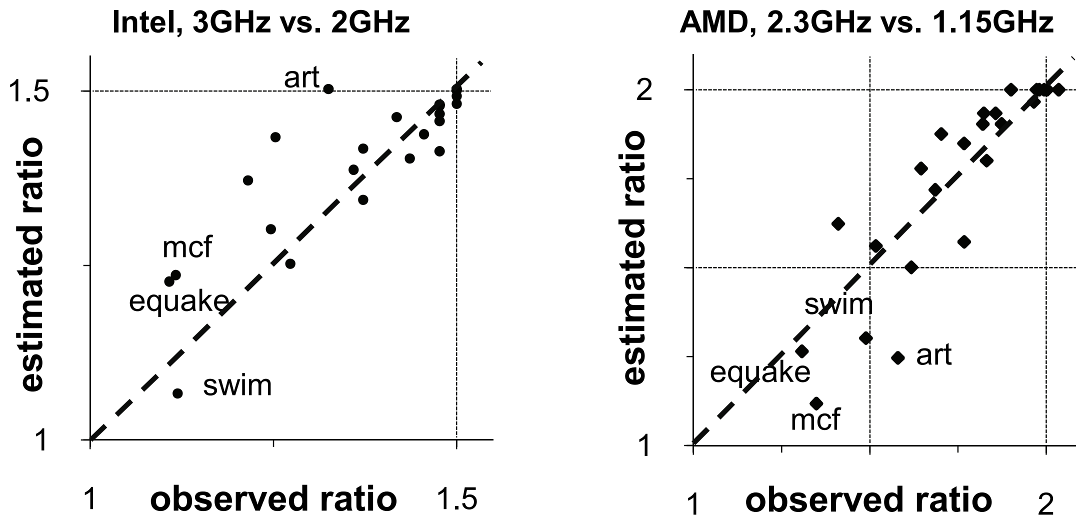


Figura 3: Factores de ganancia observados frente a factores de ganancia estimados con las firmas estáticas en dos plataformas asimétricas emuladas con procesadores de Intel (izquierda) y AMD (derecha) utilizando los mecanismos DVFS para fijar los cores a distinta frecuencia. Aparecen etiquetados algunos benchmarks en los que existe discrepancia. Si existiese una precisión perfecta en la estimación todos los puntos aparecerían en la diagonal.

Inicialmente se implementó una primera versión estática de HASS, a la que denominamos **HASS-S**, en la que las firmas se construyen en tiempo de compilación a partir de perfiles de la distancia de reuso [8] (reuse-distance profiles) con los que se estiman la tasa de fallos de cache para varias configuraciones de cache posibles (se pueden modelar distintos tamaños y distintas asociatividades). Esta versión es más apropiada para sistemas empotrados, ya que en dichos entornos puede conocerse a priori las entradas de las aplicaciones. En la Figura 3 se ilustra la precisión de las firmas estáticas para estimar los Factores de Ganancia en dos configuraciones asimétricas distintas utilizando aplicaciones del conjunto de *benchmarks* SPEC CPU2006. Como se observa en la figura, el método de estimación es suficientemente preciso para separar las aplicaciones intensivas en memoria, que son menos sensibles a la frecuencia del *core* y se concentran en la parte inferior izquierda, de las intensivas en CPU, concentradas en la parte superior derecha. Este grado de precisión es suficiente para el propósito de la planificación.
Posteriormente se implementó una versión dinámica, a la que denominamos **HASS-D**, en la que se utiliza directamente la tasa de fallos de cache obtenida en tiempo de ejecución monitorizando contadores hardware. La gestión de los contadores añade cierta complejidad al sistema operativo y también cierta sobrecarga, aunque eligiendo periodos de muestreo razonables es despreciable. Como contrapartida se evitan dos de las limitaciones más importantes que plantea la versión estática. En primer lugar se pueden distinguir distintas fases en una aplicación de manera natural, consiguiéndose así una mejor caracterización de aplicaciones muy dinámicas. Además, no es necesario contar con un profile de la aplicación, lo cual suele plantear problemas cuando no se conocen a priori los datos de entrada de la aplicación o cuando el tipo de comportamiento depende fuertemente de dichos datos de entrada.

Uno de los objetivos que nos hemos marcado en este trabajo es extender nuestros estudios sobre HASS-D a sistemas multicore asimétricos en los cuales los *cores* difieran no sólo en su frecuencia, sino también en otras características micro-arquitectónicas. Dado que no existe aún en el mercado una plataforma de este tipo, hemos utilizado como modelo procesadores de Intel de gama alta para emular los *cores* rápidos (Intel Core2) y de gama baja (Intel Atom) para emular los *cores* lentos. Nuestro objetivo se ha centrado en estudiar como es posible inferir el Factor de Ganancia, es decir, el beneficio o deterioro relativo que puede obtener una aplicación al ejecutarse en un determinado tipo de *cores* (rápido o lento), a partir de medidas de rendimiento de esa aplicación obtenidas con contadores hardware en *cores* de distinto tipo. Se trata por tanto de una extensión de HASS-D a un contexto asimétrico más general.

2. Estimación de Factores de Ganancia en Sistemas Asimétricos

2.1. Contadores Hardware

Los contadores hardware son un conjunto de registros de propósito específico que utilizan los microprocesadores para almacenar información acerca de su actividad. El número de estos contadores es limitado y varía notablemente entre CPU's. Cada contador se puede programar con el índice de un tipo de evento para ser monitorizados, como *L1 cache miss* o *branch mispredict*. Uno de los primeros procesadores en implementar estos contadores fue el *Intel Pentium*, pero no fueron documentados por Terje Mathisen hasta julio de 1994 [26]. La tabla de la figura 4 muestra algunos ejemplos de CPU's y el número de los contadores hardware de los que disponen. En comparación con los analizadores software, los contadores hardware pro-

Processor	available HW counters
UltraSparc II	2
Pentium III	2
AMD Athlon	4
IA-64	4
POWER4	8
Pentium 4	18

Figura 4: Contadores hardware para diferentes CPU's

porcionan una sobrecarga mínima en el acceso a una gran cantidad de información detallada relacionada con el rendimiento de la CPU, unidades funcionales, caches, memoria principal, etc. Otra ventaja de utilizar contadores hardware es que no se necesitan modificaciones de código fuente en general. Sin embargo, los tipos y significados de estos contadores varían de un tipo de arquitectura a otra, debido a la variación en las organizaciones hardware.
Puede haber dificultades para correlacionar la métrica de rendimiento a bajo nivel con código fuente. El número limitado de registros para almacenar los contadores a menudo obligan a los usuarios a realizar múltiples mediciones para recoger todas las métricas de rendimiento deseado. Por otra parte, los procesadores superescalares modernos planifican y ejecutan múltiples instrucciones al mismo tiempo. Estas instrucciones "*in-flight*" pueden ser retiradas en cualquier momento. Esto puede causar que los eventos de un contador se atribuyan a instrucciones equivocadas, haciendo que la precisión del análisis sea difícil o imposible.
Los nuevos procesadores han introducido métodos para mitigar algunos de estos inconvenientes. Por ejemplo, los procesadores *AMD Opteron* aplican una técnica conocida como base de muestreo de instrucciones (*Instruction Based Sampling* o IBS). La implementación de AMD de IBS proporciona contadores de hardware, tanto para recoger muestras (la parte frontal del *pipeline* superescalar) como para tomar muestras op (la parte posterior del *pipeline*).

A partir de los contadores hardware disponibles en las arquitecturas *Intel Atom* e *Intel Core2*,

hemos obtenido las medidas que se describen a continuación.

- *Intructions per cycle*(IPC). Esta medida describe el promedio de instrucciones ejecutadas en cada ciclo de reloj. El número de instrucciones por segundo se puede obtener multiplicando las instrucciones por ciclo y la velocidad del reloj del procesador en cuestión. Un cierto nivel de instrucciones por segundo puede obtenerse con un alto IPC y una velocidad de reloj baja (como en los *AMD Athlon* y en los *Intel Core2*), al contrario, un bajo IPC y alta velocidad de reloj (como en los *Intel Pentium 4*). Ambas opciones son válidas para el diseño de procesadores y la elección de una de estas dos suele venir dictada por la historia, los límites de la ingeniería o de la demanda del mercado.

- *Branch instructions per thousand instructions*. Las instrucciones de salto realizan una evaluación de una condición lógica, si se trata de un salto condicional, o realizarse siempre, en el caso de un salto incondicional. Con esta medida se obtiene el número de instrucciones de salto por cada mil instrucciones.

- *Misspredicted branch instructions per thousand instructions*. Indica el número de malas predicciones de salto realizadas debido a que el procesador predijo un camino incorrecto. Este tipo de saltos afecta al rendimiento porque el procesador tiene que descartar todo el trabajo hecho y empezar de nuevo por el camino correcto.

- *LLC request per thousand instructions*. Almacena el número de peticiones al último nivel de cache (*Last Level Cache*) por cada mil instrucciones.

- *LLC miss per thousand instructions*. Esta medida acumula la cantidad fallos en el último nivel de cache se han producido por cada mil instrucciones.

- *LLC request per cycle*. Acumula el número de peticiones que se realizan al último nivel de cache cada ciclo.

- *LLC miss per cycle*. Indica cuantos fallos se han producido en el último nivel de cache por ciclo.

- *Instruction Translation Look-aside Buffer misses per M-instructions*. Esta medida sirve para indicar el numero de instrucciones que resultan en un fallo en la *Translation Look-aside Buffer* (TLB).

- *Data Translation Look-aside Buffer misses per M-instructions*. Indica el número de peticiones que han producido un fallo en la *Data Translation Look-aside Buffer* (DTLB).

- *Floating point instruction percentage*. Esta medida contiene el porcentaje de instrucciones en punto flotante.

2.2. SPEC 2006

Standard Performance Evaluation Corporation (SPEC), es un consorcio sin ánimo de lucro que incluye a vendedores de computadoras, integradores de sistemas, universidades, grupos de investigación, publicadores y consultores de todo el mundo. Tiene dos objetivos: crear un *benchmark* estándar para medir el rendimiento de computadoras y controlar y publicar los resultados de estos tests. Para llevar a cabo estos propósitos, SPEC publica una serie de programas de prueba (o *benchmark*) para permitir una comprobación transparente, rigurosa y que se pueda reproducir de elementos computacionales, teorías científicas u otras nuevas tecnologías. Los *benchmark* utilizados en este trabajo son los SPEC 2006, diseñados para proporcionar medidas de rendimiento que pueden ser usadas para comparar cargas de trabajo computacionalmente intensivas en diferentes sistemas. SPEC CPU2006 contiene dos grandes grupos de *benchmark*: CINT2006 para medir y comparar el rendimiento de programas de enteros, y CFP2006 para compararlo y medirlo para los que utilizan programas de punto flotante. La descripción de cada uno de ellos se realiza en las siguientes subsecciones.

2.2.1. Descripción de los benchmark de CPU2006

SPEC2006 está compuesto por 29 *benchmark* agrupados en dos categorías, CINT2006, que consta de 12 programas de prueba, y CFP2006, que incluye 27. Los *benchmark* de CINT2006 se describen siguiendo el orden aparecido en [2].

- **400.perlbench**: versión reducida de Perl V5.8.7. Incluye MHonArc v2.6.8 (indexador de e-mail), SpamAssassin v2.61, MailTools v1.60 y specdiff (parte de las herramientas de SPEC) entre otros. Puede englobarse dentro del tipo de lenguajes de programación y está escrito en código C.

- **401.bzip2**: está basado en la versión de Julian Seward de bzip2 v1.0.3. La única diferencia con esta es que la versión de SPEC no realiza ninguna I/O distinta de leer el fichero de entrada. Todas las compresiones y descompresiones ocurren en memoria, aislando el trabajo que se realiza solo en la CPU del subsistema de memoria. Se engloba como programa de compresión y está escrito en lenguaje C.

- **403.gcc**: está basado en la versión v3.2 de gcc. Genera código para un procesador AMD Opteron. El programa se ejecuta como un compilador con muchos de sus *flags* activados. Está escrito en lenguaje C y pertenece al grupo de *benchmarks* de compilador de C.

- **429.mcf**: deriva de MCF, un programa que se utiliza para planificar vehículos de transporte público masivo. Esta versión utiliza exclusivamente aritmética con números enteros. Pertenece al grupo de optimización combinatorial y está escrito en C.

- **445.gobmk**: el programa inicia el juego Go y ejecuta un conjunto de comandos para analizar las posiciones dentro del mismo. Un test típico involucra lecturas de un cierto punto y después ejecuta un comando para analizar dicha posición. La categoría a la que pertenece este *benchmark* es la de inteligencia artificial y está descrito en C.

- **456.hmmer**: el perfil *Hidden Markov Models* (perfil HMMs) son modelos estadísticos de muchas secuencias de alineamiento, que se usan en computación biológica para buscar patrones en secuencias del ADN. La técnica se usa para hacer una búsqueda intensiva en una base de datos, usando descripciones estáticas. Se usa para el análisis de secuencia de proteínas. El código está escrito en C y pertenece a la categoría de búsqueda de secuencias genéticas.

- **458.sjeng**: está basado en la versión de Sjeng v11.2, que es un programa de ajedrez con varias variantes de este juego. Trata de encontrar el mejor movimiento a través de combinaciones *alpha-beta*, movimientos avanzados de ordenación, evaluación posicional y otras heurísticas. En la práctica, explora el árbol de variaciones resultantes de una posición dada. Pertenece a la categoría de los de inteligencia artificial y su código está escrito en C.

- **462.libquantum**: Es una librería para simular el comportamiento de un computador cuántico, ejecutando el algoritmo de factorización en tiempo polimonial de Shor. *libquantum* proporciona una estructura para representar un registro cuántico y algunas puertas elementales. Se puede englobar en el grupo de test físicos (computación cuántica) y está escrito en C.

- **464.h264ref**: basado en el compresor de vídeo H.264/AVC (*Advanced Video Coding*), en la versión v9.3 de h264avc. Este estándar de vídeo remplaza al actual MPEG-2, y será aplicado en la próxima generación de DVD's (Blu-ray y HD-DVD) y en *broadcasting*. Pertenece al conjunto de compresión de vídeo y su código está en lenguaje C.

- **471.omnetpp**: este *benchmark* realiza simulaciones de eventos en una red Ethernet grande. La simulación se basa en OMNeT++, un *framework* de simulación genérico y de abierto. Está escrito en C++ y pertenece a la categoría de simulación de eventos.

- **471.astar**: es una librería para el reconocimiento de caminos en dos dimensiones de mapas, que incluye el algoritmo A*. También incluye funciones pseudo-intelectuales para la determinación de regiones en el mapa. Descrito en código C++, pertenece a la categoría de inteligencia artificial.

- **483.xalancbmk**: es una modificación del programa Xalan-C++, un procesador XSLT. La versión de Xalan-C++ v1.8 es una implementación de las recomendaciones W3C para las transformaciones XSL (XSLT) y el lenguaje XPath.Su código está en C++ y se engloba en la categoría de procesamiento de código XML.

Una vez descritos los *benchmark* que componen CINT2006, nos disponemos a describir los que componen CFP2006, que como antes se ha descrito, son de punto flotante. Estos *benchmark* son los que se describen a continuación y siguiendo el orden que aparece en [1].

- **410.bwaves**: realiza cálculos en 3 dimensiones sobre un líquido transónico, laminar, transitorio y viscoso. El algoritmo implementa una forma de resolver ecuaciones de Navier-Stokes utilizando el algoritmo Bi-CGstab, que resuelve sistemas de ecuaciones

iterativas no simétricas y lineales. Está dentro de la categoría de computación para fluidos dinámicos y está escrito en Fortran.

- **416.gamess**: implementa un rango amplio de cálculos de química cuántica. Para el caso de SPEC y su carga de trabajo, hace uso de métodos como el de Hartree Fock. Su código está en Fortran y pertenece al grupo de computación de química cuántica.

- **433.milc**: es un conjunto de códigos escritos en C escritos por *MIMD Lattice Computation* (MILC) para hacer simulaciones de cuatro dimensiones SU(3) en máquinas paralelas MIMD. Pertenece a la categoría de física cromodinámica cuántica y está escrito en C.

- **434.zeusmp**: está basado en ZEUS-MP, un código para calcular fluidos dinámicos y desarrollado por el Laboratorio de Astrofísica Computacional de la Universidad de Illinois (NCSA, *University of Illinois at Urbana-Champaign*) para la simulación de fenómenos astrofísicos. Encaja en el grupo de magneto termodinámica y su código es Fortran.

- **435.gromacs**: deriva de GROMACS, un paquete que resuelve dinámicas moleculares, como por ejemplo, la simulación de ecuaciones *newtonianas* para el movimiento de cientos de millones de partículas. Esta versión analiza y hace uso de las proteínas Lysozyme en una solución de agua e iones.Está escrito tanto en C como en Fortran y se engloba en el conjunto de *benchmarks* de química/dinámica molecular.

- **436.cactusADM**: es una combinación de Cactus y BenchADM. Resuelve las ecuaciones de evolución de Einstein, que describen las curvas del espacio-tiempo como respuesta a su contenido de materia. Se incluye en la categoría de física/relatividad general y el código combina C y Fortran.

- **437.leslie3d**: deriva de LESlie3d (*Large-Eddy Simulations with Linear-Eddy Model in 3D*), que sirve para resolver código computacional de dinámica de fluidos (CFD). Hace uso de simulaciones Large-Eddy con el modelo Linear-Eddy en 3 dimensiones y utiliza un esquema de integración con predictor-corrector (*MacCormack Predictor-Corrector*). El código de este *benchmark* está en Fortran y se cataloga dentro del grupo de dinámica de fluidos.

- **444.namd**: simula sistemas de grandes biomoléculas. El test contiene 92224 átomos de apolipoproteína A-I. Pertenece al grupo de biología y dinámica molecular y su código está en C++.

- **447.dealII**: es un programa que usa deal.II, una librería de C++ dirigida a los elementos finitos adaptativos y a la estimación de errores. Incluye la librería Boost, que ofrece una interfaz moderna para estructuras de datos completas y algoritmos que requieren de adaptabilidad y permiten el uso de una variedad de elementos finitos en una, dos y tres dimensiones espaciales, así como problemas dependientes del tiempo. Pertenece al conjunto de soluciones de ecuaciones con derivadas parciales.

- **450.soplex**: está basado en la versión de SoPlex v1.2.1, que resuelve un programa linear usando el algoritmo Simplex y álgebra lineal dispersa, en particular, la factorización LU. Se engloba en el grupo de programación linear y optimización y está en lenguaje C++.
- **453.povray**: se encarga de renderizar imágenes mediante *Ray-tracing*, que es una técnica de renderización que calcula una escena de una imagen. Como entrada recibe una imagen sin aliasing de 1280x1024. Se cataloga dentro del conjunto de visualización por computador y, al igual que el anterior, su código también está en C++.
- **454.calculix**: está basado en CalculiX, un software libre para aplicaciones lineales y no lineales en tres dimensiones con elementos finitos. Hace uso de la librería SPOOLES y pertenece a la categoría de mecánica estructural. Su código es una mezcla de C y Fortran.
- **459.GemsFDTD**: resuelve las ecuaciones de Maxwell en 3D en el dominio del tiempo usando el método de dominio temporal de diferencias finitas (FDTD). El código se divide en tres pasos, inicialización, repetición del cálculo durante múltiples ciclos y el post-proceso. Más del 99 % del tiempo se emplea en el segundo paso. Su código está en Fortran y entra dentro del grupo de computación electromagnética (CEM).
- **465.tonto**: es un paquete de código abierto de química cuántica diseñado por Dylan Jayatilaka y Daniel J. Grimwood y utiliza un diseño orientado a objetos de Fortran en su versión Fortran95. El código es fácilmente extendible y es sencillo de entender y usar [20]. Pertenece a la categoría de cristalografía cuántica.
- **470.lbm**: este *benchmark* implementa el método conocido por el nombre de *Lattice Boltzmann Method* (LBM) para simular fluidos incomprensibles en 3 dimensiones como se describe en [30]. Está escrito en C y se engloba en la categoría de dinámica de fluidos.
- **481.wrf**: está basado en el modelo del tiempo *Weather Research and Forecasting* (WRF), en la versión 2.0.2. WRF es un sistema de predicción del tiempo diseñado para proporcionar la previsión con un rango de distancias que abarca desde metros a miles de kilómetros. Se incluye en el grupo de previsión del tiempo y combina tanto el lenguaje C como el Fortran.
- **482.sphinx3**: es un sistema de reconocimiento de voz de la Universidad de Carnegie Mellon. Pertenece a la categoría de reconocimiento de voz y el lenguaje de programación utilizado es C.

2.3. Clustering con K-means

El término *k-means* fue utilizado por primera vez por James MacQueen en 1967 [6] a partir de la idea de Hugo Steinhaus en 1956 [15]. El algoritmo estándar fue propuesto por primera vez por Stuart Lloyd en 1957 como una técnica para la modulación del pulso (*pulse-code modulation*), aunque no fue publicado hasta 1982 [28, 29]. *K-means clustering* es un método de análisis de clusters que tiene por objetivo dividir *n* observaciones en *k* grupos en la que cada observación pertenece al grupo más cercano a la media. Es similar al algoritmo de expectación-maximización (*expectation-maximization algorithm*) de expectativas para las mezclas *gaussianas* en que ambos pretenden encontrar los centros de agrupaciones naturales en los datos, así como en el enfoque de refinamiento iterativo empleado por ambos algoritmos.
Descripción: Dado un conjunto de observaciones (x_1, x_2,.....,x_n) donde cada observación es un vector real *d*-dimensional, *k-means clustering* tiene por objetivo dividir las *n* observaciones en *k* grupos (*k* < *n*) S=S_1, S_2,. . . .., S_n con el fin de minimizar la suma dentro de los grupos (*within-cluster sum of squares, WCSS*)

$$arg\,min \sum_{i=1}^{k} \sum_{x_j \epsilon S_i} \|x_j - \mu_i\|^2$$

donde μ_i es la media de puntos en S_i. De acuerdo con la complejidad computacional, el algoritmo de *k-means clustering* es:

- **NP-complejo** en general en espacios euclídeos incluso para 2 clusters [4] [31] y para un número general de *k* grupos incluso en el plano [25].
- Si *k* y *d* son fijos, el problema se puede resolver exactamente en tiempo *O*($n^{dk+1} log\, n$), donde *n* es el número de entidades que se agrupan [17].

En consecuencia, se suelen utilizar variedades de algoritmos heurísticos. El algoritmo estándar utiliza una técnica de refinamiento iterativa. Debido a su ubicuidad se suele llamar *k-means* a dicho algoritmo, aunque también se lo conoce como el algoritmo de Lloyd, particularmente en la comunidad informática. Dado un conjunto inicial de medias de *k* ($m_1^{(1)}, \ldots, m_k^{(1)}$), que puede ser especificado de forma aleatoria o mediante alguna heurística, el algoritmo actúa alternando entre dos pasos:

- **Asignación**: asigna cada observación con el grupo más cercano al medio:

$$S_i^{(t)} = \left\{ x_j : \left\| x_j - m_i^{(t)} \right\| \leq \left\| x_j - m_{i*}^{(t)} \right\| for\, all\, i^* = 1, ..., k \right\}$$

- **Actualización**: calcula los nuevos medios para ser el centroide de las observaciones en el grupo:

$$m_i^{(t+1)} = \frac{1}{\left| S_i^{(t)} \right|} \sum_{x_j \epsilon S_j^{(t)}} x_j$$

El paso de asignación también se conoce por el nombre de paso expectativo y el de actualización como el paso de maximización, haciendo de este algoritmo una variante del algoritmo general de expectación-maximización (*expectation-maximization algorithm*). Se considera que el algoritmo converge cuando las asignaciones ya no cambian. En la figura 5 se muestra y describe de forma gráfica los pasos seguidos por el algoritmo.

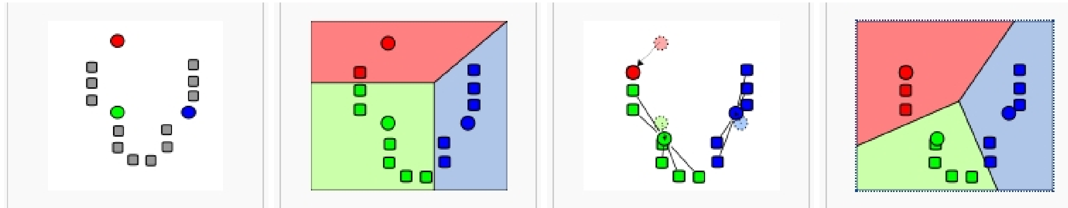


Figura 5: 1) los centroides iniciales (para este caso, k=3) se seleccionan de forma aleatoria para el conjunto de datos. 2) Los k clusters se crean asociando cada dato del conjunto con su centroide más cercano. Las particiones en este caso representan el diagrama de Voronoi generado por los centroides. 3) El centroide de cada cluster se convierte en el nuevo medio. 4) Los pasos 2 y 3 se repiten hasta que el algoritmo converge

Como es un algoritmo heurístico no hay garantías de que este converja a una solución óptima global, y el resultado depende de los grupos iniciales. Como el algoritmo normalmente es muy rápido, lo que se suele hacer es ejecutarlo múltiples veces con diferentes condiciones iniciales. No obstante, en el peor caso, *k-means* puede converger de manera muy lenta. En particular, se ha demostrado que existen ciertos grupos de puntos, incluso en 2 dimensiones, con los que *k-means* obtiene tiempos exponenciales, $2^{\Omega(n)}$, para converger [3, 9]. Sin embargo, este conjunto de puntos no parece que surja en la práctica [5].

2.3.1. K-means y MATLAB®

El algoritmo de *k-means clustering* es el referente principal entre los diversos métodos para seleccionar grupos representativos entre los datos. Existen una serie matrices que constituyen el fundamento para la implementación de este tipo de algoritmo, entre ellas:

- Matriz de datos
- Matriz de distancias
- Matriz de centroides
- Matriz de pertenencias

Sus diferentes variantes se basan fundamentalmente en la forma de medir distancias entre los datos y los grupos, el criterio para definir la pertenencia de los datos a cada grupo y la forma de actualizar dichos grupos.
Matriz de datos. Almacena el conjunto de muestras de las que se pretende obtener los

puntos significativos que representen los grupos que se van a clasificar. Esta matriz puede representarse como la siguiente estructura:

$$X = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1N} \\ x_{21} & x_{22} & \dots & x_{2N} \\ & & . & \\ & & . & \\ x_{n1} & x_{n2} & \dots & x_{nN} \end{bmatrix}$$

siendo su orden de n x N, donde N es el número de observaciones o muestras de las que se dispone y n son los rasgos que caracterizan a cada muestra. En la figura 6 se representa el caso de n=2 y N=7, con dos clusters o centros de grupos representativos. Por ejemplo, la

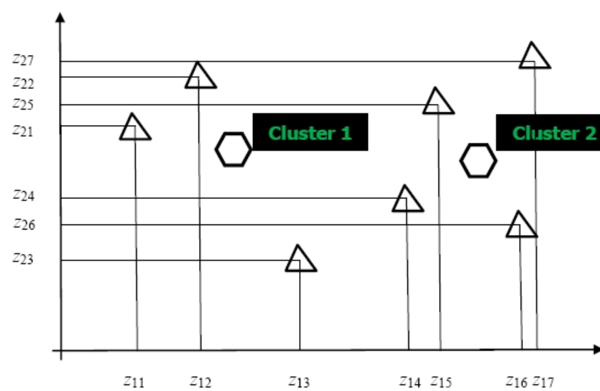


Figura 6: Representación de una matriz con n= 2 y N= 7 con dos clusters representativos [H]

representación de una matriz X definida en *matlab* por el siguiente código se muestra en la figura 7.

```
X=[0.5 0.5 0.5 1 1.5 2 2.5 2.5 2.5; 1 2 3 1 1 1 2 3 4];
plot(X(1,:),X(2,:),'vb')
```

Matriz de distancias. Almacena la distancia de cada punto de la matriz de datos a cada centro de grupo o centroide, cuyo tamaño es $c x$ N, siendo c el número de clusters.

$$d = \begin{bmatrix} d_{11} & d_{12} & \dots & d_{1N} \\ d_{21} & d_{22} & \dots & d_{2N} \\ & & . & \\ & & . & \\ d_{c1} & d_{c2} & \dots & d_{cN} \end{bmatrix}$$

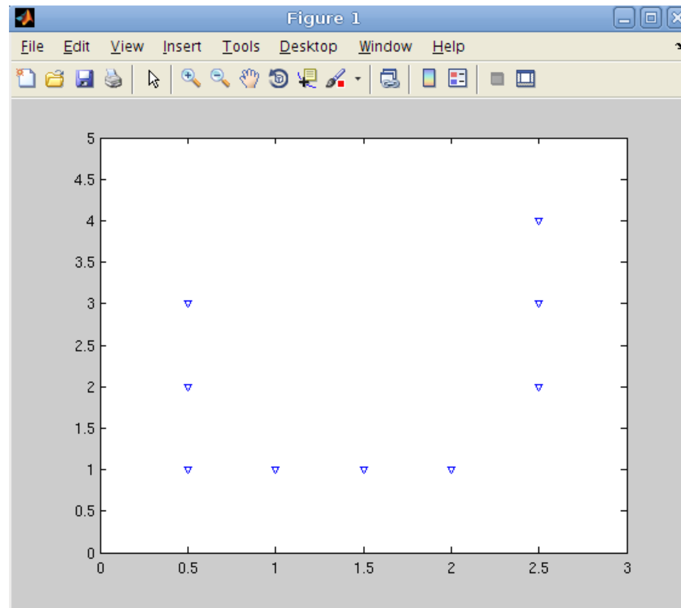


Figura 7: Representación de la matriz X en Matlab

donde cada componente d_{ij} representa la distancia de la muestra j (j=1:N) al centroide i (i=1:c). El significado de cada componente de la matriz de distancias se representa en la figura 8.

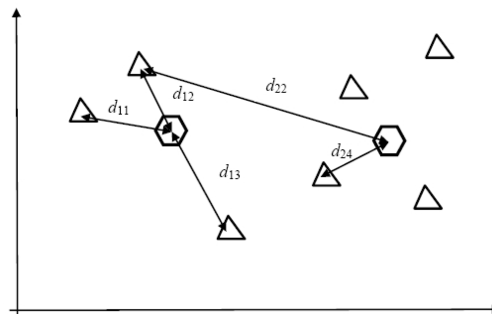


Figura 8: Significado de cada componente de la matriz d anteriormente descrita

Uno de los elementos que caracterizan al algoritmo *k-means clustering* es que debe definirse el número de clusters que se desean obtener. Supóngase que el objetivo es obtener dos clusters de la matriz de datos:

```
X=[0.5 0.5 0.5 1 1.5 2 2.5 2.5 2.5; 1 2 3 1 1 1 2 3 4];
```

Considerando que las condiciones iniciales de los clusters son el primer y último valor, se tiene:

```
v=[0 2.5; 1 4]
```

Si se utiliza la distancia euclídea como medida de distancias entre cada elemento de la muestra y los centroides, entonces se cumple:

$$d_{ij} = \sqrt{\sum_{k=1}^{n}(X_{kj} - v_{ki})^2}$$

Matriz de clusters o centroides. Los centroides que se generen durante la evolución del algoritmo de clustering van siendo almacenados en la matriz **v**(n x c), siendo n el número de rasgos que caracterizan a cada cluster y c el número de grupos que deben definirse antes de aplicar el algoritmo. La matriz de centroides se representa por:

$$v = \begin{bmatrix} v_{11} & v_{12} & \dots & v_{1c} \\ v_{21} & v_{22} & \dots & v_{2c} \\ & & . & \\ & & . & \\ v_{c1} & v_{c2} & \dots & v_{cc} \end{bmatrix}$$

Esta matriz se actualiza continuamente durante la evolución del algoritmo. Para ello, debe previamente definirse a que grupo pertenece cada muestra. Si se utiliza la mínima distancia como criterio para actualizar el valor de los centroides, entonces ha de definirse la matriz de pertenencias.

Matriz de pertenencias. En la matriz de pertenencias se define la pertenencia a uno u otro grupo (normalmente normalizado en el intervalo [0, 1]). Esta matriz define la eficiencia del algoritmo de clustering, pues será la base para actualizar los valores de los centroides. Si se define la matriz de pertenencias por:

$$u = \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1N} \\ u_{21} & u_{22} & \dots & u_{2N} \\ & & . & \\ & & . & \\ u_{c1} & u_{c2} & \dots & u_{cN} \end{bmatrix} \begin{matrix} Grupo\,1 \\ Grupo\,2 \\ \\ Grupo\,c \end{matrix}$$

El algoritmo de *k-means clustering* puede resumirse por los siguientes pasos:

1. Condiciones iniciales: Definir el número de centroides c y los centroides iniciales (**v**), basado en la matriz de datos (**X**).

2. Calcular la matriz de distancias (**d**)

3. Calcular la matriz de pertenencia (**u**)

4. Actualizar la matriz de centroides (**v**)

Los puntos 2, 3 y 4 se repiten hasta un criterio de parada definido previamente.
Como se puede apreciar en este algoritmo, existen tres factores de primordial importancia, algunos de ellos con criterios de selección heurística:

1. Definir cual es el número de clusters.

2. Seleccionar el criterio para medir distancias.

3. Definir el criterio para determinar el grado de pertenencia a cada grupo.

Función kmeans de Matlab. Matlab posee una función que realiza el algoritmo de *k-means clustering*. La sintaxis es:

```
[...] = kmeans(...,'param1',val1,'param2',val2,...)
```

donde, entre otras variables, devuelve:

(a). La matriz de pertenencia

(b). La matriz de centroides

(c). La matriz de distancias.

Con respecto a los parámetros de entrada, se puede definir:

(a). El criterio para medir la distancia entre la matriz de datos y los centroides (la medida implícita es la norma euclídea).

(b). La forma en que se establecen las condiciones iniciales de los centroides (de forma predeterminada escoge como valores iniciales a muestras pertenecientes a la matriz de datos).

(c). Número de iteraciones en la búsqueda de los centroides (el valor implícito es 100).

Veámos un ejemplo en Matlab con la siguiente matriz de datos.

```
>>x=[0 0 0 1 1 1 2 3 4 5 5 5 6 6 6 7 7 7 8 9 10 11 11 11 12 12 12];
>>y=[1 2 3 1.5 2 2.5 2 2 2 1.5 2 2.5 2 4 6 3 4 5 4 4 4 3 4 5 2 4 6];
>>z=[x; y]';
%Para obtener 5 centroides se aplica:
>> [U, v, sumd, D]=kmeans(z,5);
%Lo cual devuelve una matriz de pertenencias
>> U=U'
U = 1 1 1 1 1 1 1 4 4 4 4 4 4 2 2 3 3 2 3 3 5 5 5 5 5 5 5
%basado en una matriz de distancias
>>D=D'
D =
Columns 1 through 10
1.5102 0.5102 1.5102 0.3316 0.0816 0.3316 1.6531 5.2245 10.7959 18.6173
```

```
56.1111 49.1111 44.1111 40.6944 37.4444 34.6944 27.7778 20.1111 14.4444 14.0278
67.6250 63.1250 60.6250 50.6250 48.6250 47.1250 36.1250 25.6250 17.1250 12.6250
22.7778 21.7778 22.7778 13.6944 13.4444 13.6944 7.1111 2.7778 0.4444 0.3611
136.3673 131.3673 128.3673 112.0459 109.7959 108.0459 90.2245 72.6531 57.0816 45.7602
Columns 11 through 20
18.3673 18.6173 27.9388 31.9388 43.9388 40.5102 43.5102 48.5102 57.0816 72.6531
10.7778 8.0278 9.1111 1.1111 1.1111 4.4444 1.4444 0.4444 3.7778 8.1111
10.6250 9.1250 6.1250 3.1250 8.1250 1.1250 0.6250 2.1250 0.1250 1.6250
0.1111 0.3611 1.7778 5.7778 17.7778 6.4444 9.4444 14.4444 15.1111 22.7778
43.5102 41.7602 31.9388 27.9388 31.9388 19.3673 18.3673 19.3673 10.7959 5.2245
Columns 21 through 27
90.2245 106.7959 109.7959 114.7959 127.3673 131.3673 143.3673
14.4444 25.7778 22.7778 21.7778 41.1111 33.1111 33.1111
5.1250 11.1250 10.6250 12.1250 21.1250 18.1250 23.1250
32.4444 41.1111 44.1111 49.1111 53.7778 57.7778 69.7778
1.6531 1.0816 0.0816 1.0816 4.5102 0.5102 4.5102
%Siendo la matriz de centroides:
>> v
v = 0.7143      2.0000
6.3333      5.0000
7.7500      3.7500
4.6667      2.0000
11.2857    4.0000

>> plot(z(:,1),z(:,2),'v');
>> hold on
>> plot(v(:,1),v(:,2),'sr');
```

Con los tres últimos comandos puede visualizarse la figura 9 tal y como se muestra en *Matlab*.

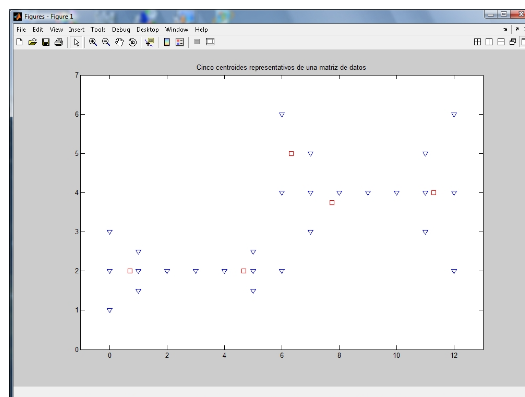


Figura 9: Resultado en Matlab para una matriz de datos z con sus centroides. Los triángulos representan los datos y los cuadrados los centroides

3. Resultados

Una vez descritas las bases teóricas de *k-means clustering*, los contadores hardware y las medidas obtenidas a partir de ellos, y los SPEC2006 utilizados en este trabajo, nos disponemos a mostrar y discutir los resultados del mismo. Los *benchmark* han sido compilados y ejecutados en dos tipos de procesadores (un *Intel Atom N270* de 32 bits y con una frecuencia de 1.60Ghz, y un *Intel Core2* de 64 bits) con el objetivo de estudiar el comportamiento de las medidas tomadas a partir de los contadores hardware descritos anteriormente. Con la medida del porcentaje de instrucciones en punto flotante (*floating point instructions percentage*) mostradas en la sección 3.1(figuras 21 y 31) dividimos los *benchmarks* en dos grupos, los de enteros por una parte y los de punto flotante por otra, con el objetivo de conseguir simular un sistema con procesadores asimétricos donde el *Atom* podría llevar a cabo el papel de *core* lento y el *Intel Core2* ejercer de *core* rápido. Como es posible modificar la frecuencia del *Core2*, las pruebas realizadas se han hecho cambiando esta entre 1.86Ghz y 1.60Ghz con el fin de obtener más medidas y así comparar el rendimiento de las ejecuciones con respecto al *Atom*.
Con el propósito de obtener unos resultados óptimos, separamos los programas de prueba en dos grupos, los pertenecientes a CFP2006 y los que pertenecen a CINT2006, y nos centramos en estos últimos. En este punto cabe destacar que se han encontrado problemas a la hora de compilar los *benchmark* **omnetpp** y **gamess**, correspondientes a los grupos CINT2006 y CFP2006 respectivamente, con lo que sus resultados han sido ignorados.
Con la finalidad de llevar a cabo el trabajo propuesto, el primer paso realizado, una vez compilados los SPEC, ha sido lanzarlos en cada una de las arquitecturas descritas anteriormente (Atom, Core2 a 1.86Ghz y Core2 a 1.60Ghz) para obtener sus tiempos de ejecución y los contadores hardware que se desean.

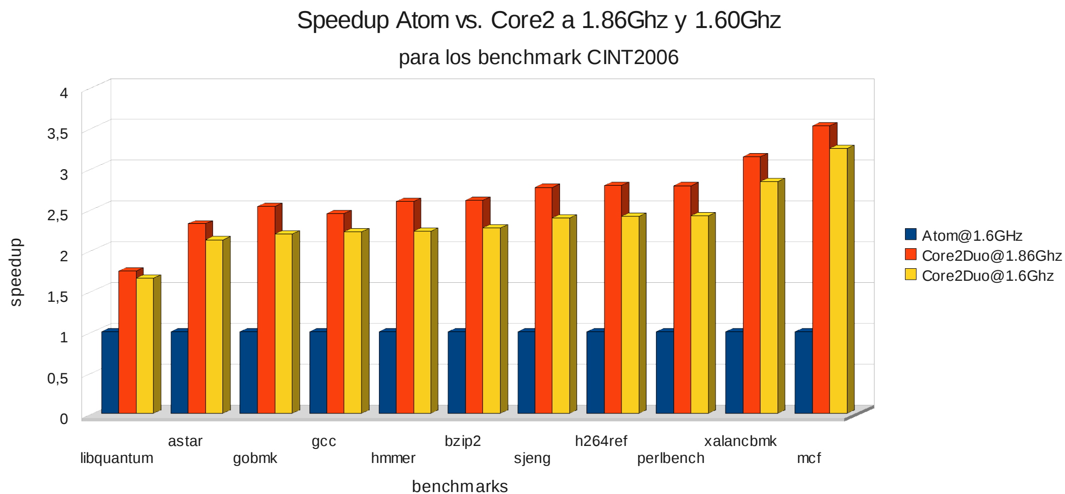


Figura 10: Mejora al ejecutar los *benchmark* de enteros en las arquitecturas Intel Atom e Intel Core2 con distintas frecuencias

La figura 10 muestra la mejora o speedup que supone ejecutar cada uno de los *benchmarks* en las diferentes arquitecturas tomando como base los tiempos de ejecución en el *core* lento, que para nuestro estudio es el Atom. Como puede observarse, la ejecución en el *Core2* es más rápida para todos los programas, ya esté configurado con una u otra frecuencia, lo que explica la elección de este como el *core* rápido. El proceso se repite, en este caso para los programas pertenecientes a CPF2006, obteniendo los resultados expuestos en la figura 11.

Figura 11: Mejora al ejecutar los *benchmark* de punto flotante en las arquitecturas Intel Core2 con diferentes frecuencias e Intel Atom

3.1. Resultado de las medidas obtenidas a partir de los contadores hardware

3.1.1. Medidas obtenidas para CINT2006

Los datos obtenidos al medir los contadores hardware de cada *benchmark* se presentan en las siguientes subsecciones, describiendo en cada caso cual es la medida que se ha tenido en cuenta y las características de estas. Nos centramos en primer lugar en los valores de dichas medidas para los SPEC que forman parte de CINT2006.
La figura 12 muestra los valores obtenidos para el IPC en cada una de las arquitecturas descritas con anterioridad para el grupo de SPEC's que forman parte de CINT2006. En ella podemos observar como, para todos los casos, el número de IPC es mayor cuando se ejecuta cada *benchmark* en la arquitectura del Core2, obteniendo resultados similares para esta arquitectura pero con diferentes frecuencias. También contemplamos como el *benchmark* **mcf** es el que menores valores para el IPC presenta, siendo el *benchmark* **hmmer** el que obtiene los mayores. Cabe destacar que para los casos de **libquantum** y **xalancbmk** el valor del IPC tomado en la arquitectura Core2 a 1.60Ghz es algo mayor que para la misma arquitectura pero con una frecuencia de 1.80Ghz.

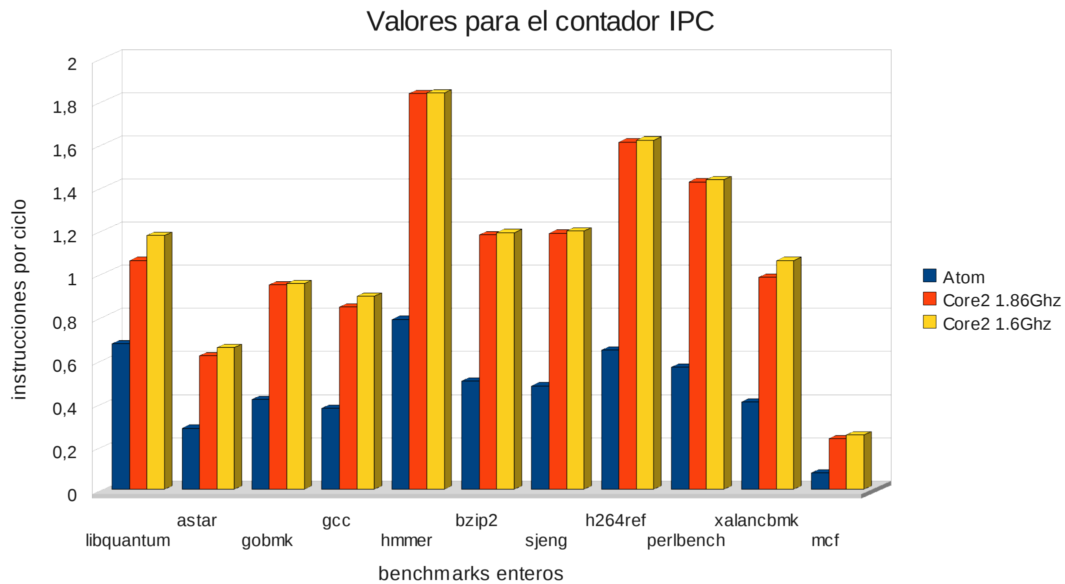


Figura 12: valores obtenidos para el IPC en el Intel Atom y en el Core2 con diferentes frecuencias para el conjunto de benchmarks pertenecientes a CINT2006.

Los datos adquiridos para la medida de instrucciones de salto por cada mil instrucciones (*branches per k-instructions*) se muestran en la figura 13, en la que se observa como el número de instrucciones de salto es prácticamente idéntico en las tres arquitecturas. Como se puede advertir en dicha imagen, el *benchmark* con menor número de instrucciones de salto por cada mil instrucciones es el **hmmer**, llegando a tener un valor por debajo de 25 instrucciones de este tipo. El *benchmark* con mayor valor para esta medida es **xalancbmk**.

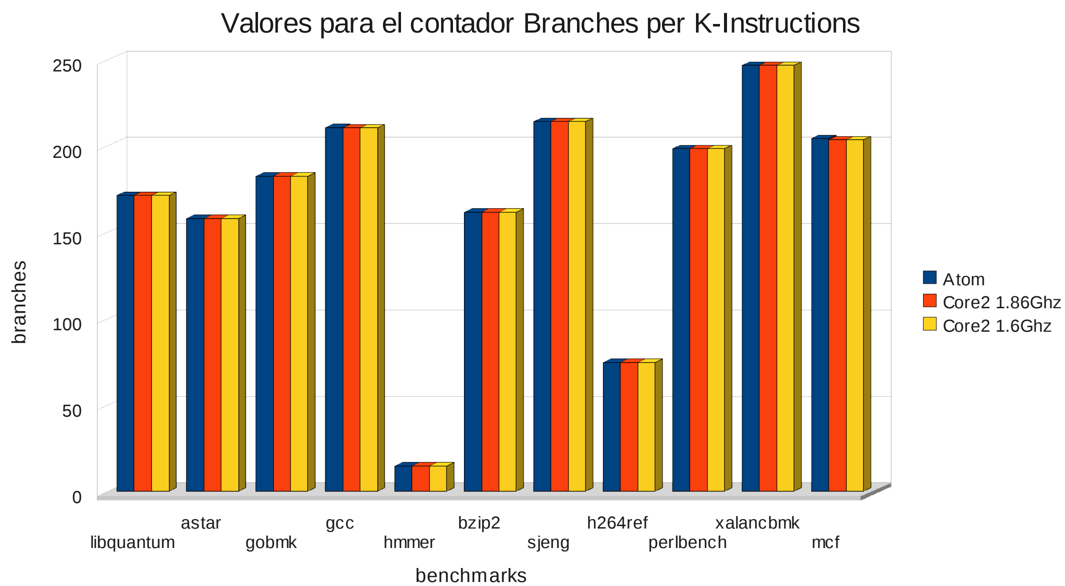


Figura 13: valores obtenidos para el número de instrucciones de salto por cada mil instrucciones en el Intel Atom y en el Core2 con diferentes frecuencias para el conjunto de benchmarks que pertenecen a CINT2006.

Otra de las medidas obtenidas para este trabajo ha sido el número de peticiones al último nivel de caché por cada mil instrucciones, que muestra la figura 14. En ella podemos observar como el *benchmark* con mayor valor es el **mcf**, siendo muy superior al del resto. En este caso, los valores obtenidos en el Atom son mayores que los obtenidos en el Core2, independientemente de la frecuencia de este. Solo en el caso de **gcc** los valores obtenidos en el Core2 son algo superiores a los obtenidos en el Atom.

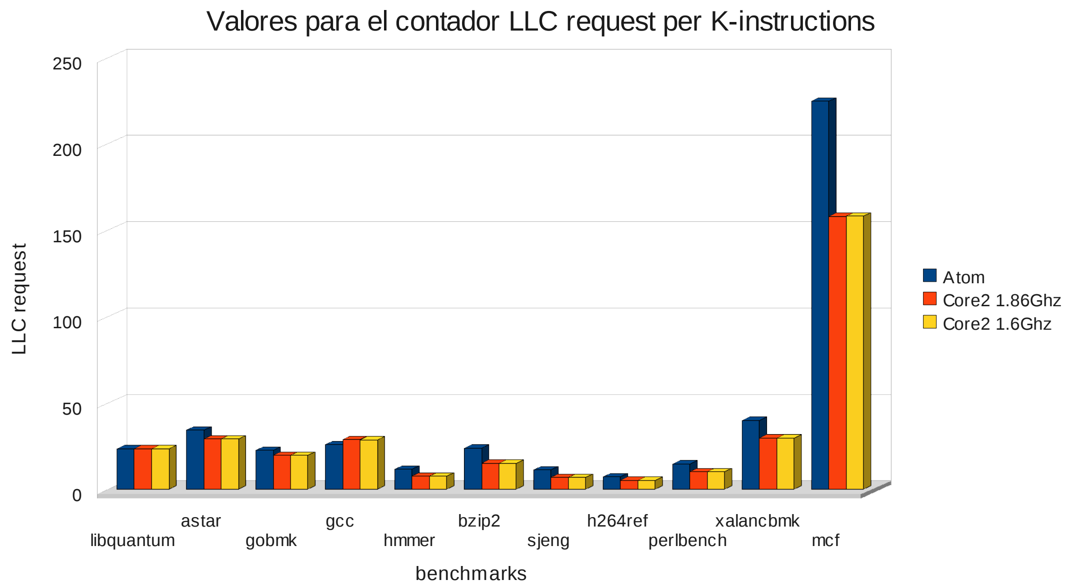


Figura 14: valores obtenidos para el número de accesos al último nivel de cache por cada mil instrucciones en el Intel Atom y en el Core2 con diferentes frecuencias para el conjunto de benchmarks que pertenecen a CINT2006.

Al igual que se ha tomado el número de peticiones al último nivel de cache, también se ha tenido en cuenta el número de fallos producidos en este nivel por cada mil instrucciones, lo que se muestra en la figura 15. De ella podemos deducir que, como en el caso anterior, el número de fallos en este nivel de cache es bajo en todos los *benchmark* salvo en el caso de **mcf**, donde se obtienen valores de 90 en el caso del Atom y de 40 para el Core2 con ambas frecuencias. Otro dato a destacar es que, como sucedía para el número de peticiones al último nivel de cache por cada mil instrucciones, para este caso el número de fallos es mayor en el Atom que en el Core2, sin tener en cuenta la frecuencia de este último.

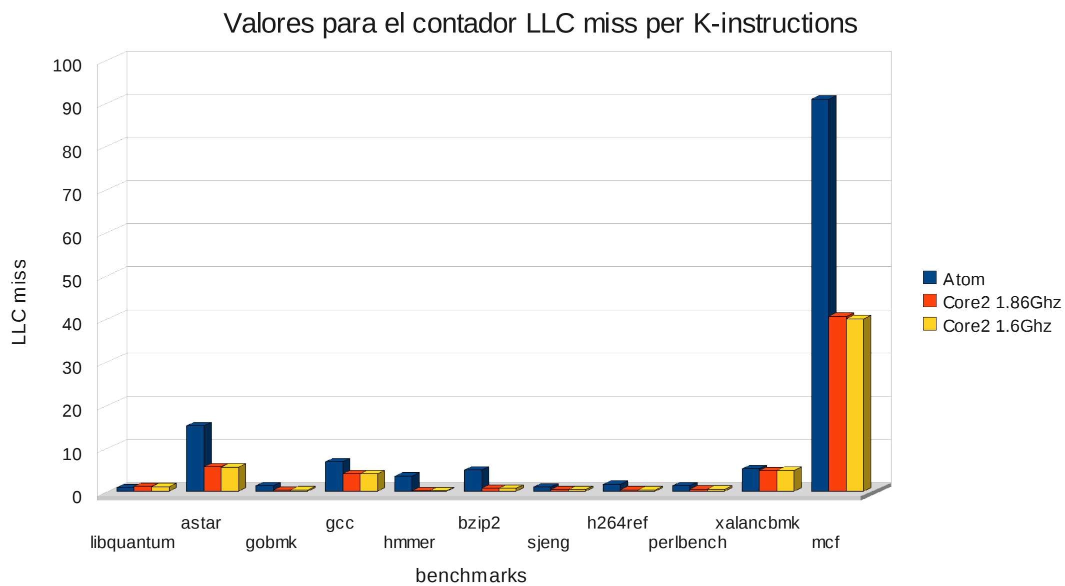


Figura 15: valores obtenidos para el número de fallos en el último nivel de caché por cada mil instrucciones en el Intel Atom y en el Core2 con diferentes frecuencias para el conjunto de benchmarks que pertenecen a CINT2006.

Continuando con el estudio de las medidas obtenidas a partir de los contadores hardware adquiridos en las arquitecturas descritas, la figura 16 muestra la cantidad de saltos mal predichos por cada mil instrucciones para cada uno de los *benchmarks* que aparecen reflejados. En el caso de **gobmk**, **sjeng**, **perlbench** y **gcc** la diferencia entre los datos tomados en el Atom y en el Core2 son considerables, llegando alcanzar en algún caso el doble. Para **hmmer** sin embargo, los valores son apenas perceptibles. En **libquantum** sucede que los valores obtenidos en el Core2 son superiores a los obtenidos en el Atom, al contrario que en el resto de *benchmarks*.

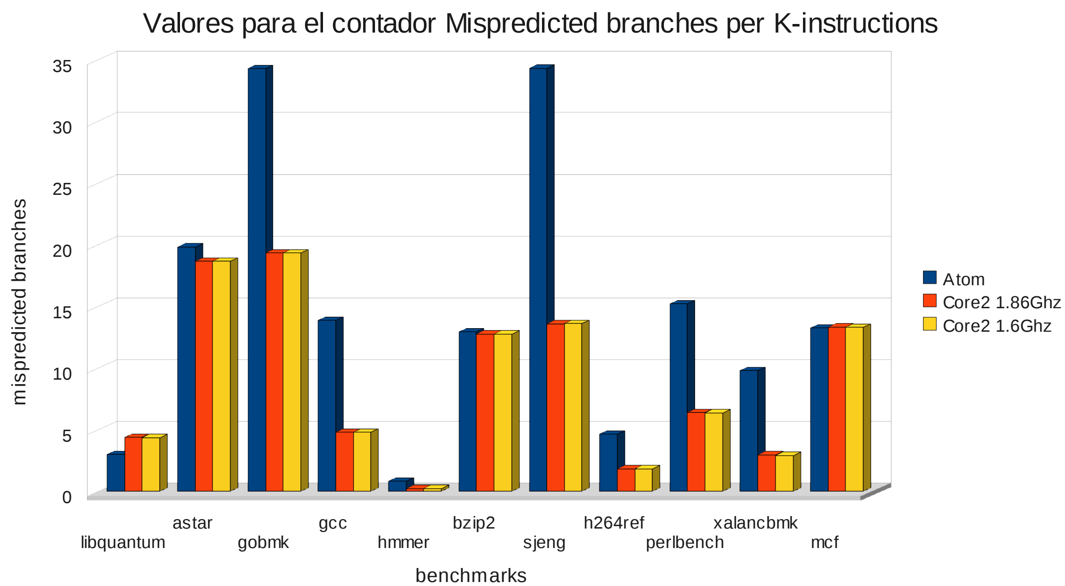


Figura 16: valores para el número de saltos mal predichos por cada mil instrucciones en el Intel Atom y en el Core2 con diferentes frecuencias para el conjunto de benchmarks que pertenecen a CINT2006.

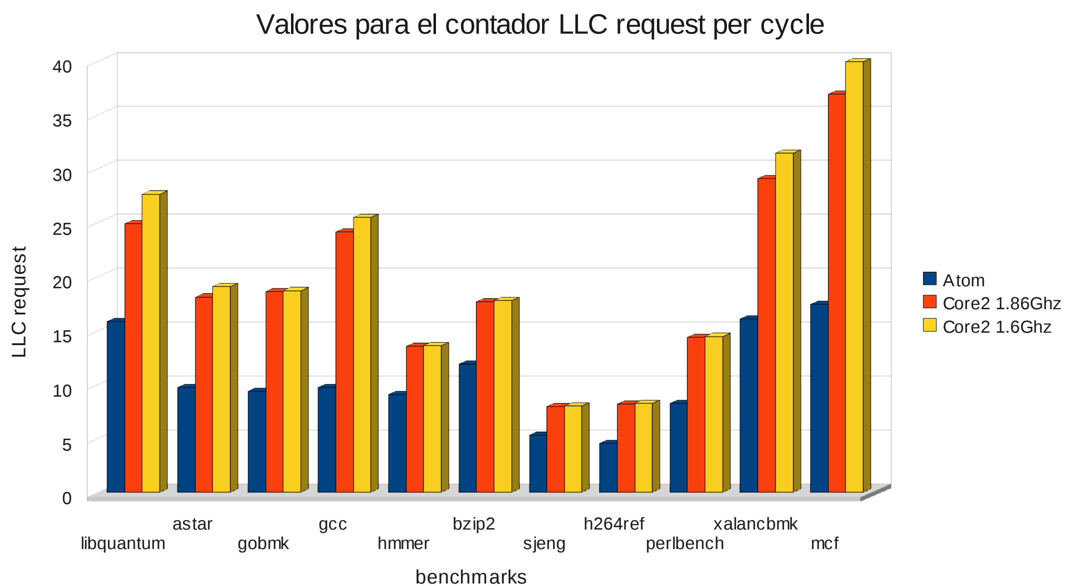


Figura 17: valores obtenidos para el número de accesos al último nivel de cache por ciclo el Intel Atom y en el Core2 con diferentes frecuencias para el conjunto de benchmarks que pertenecen a CINT2006.

Las figuras 17 y 18 hacen referencia al número de accesos y fallos al último nivel de cache

respectivamente, pero en este caso tomados por ciclo en lugar de por cada mil instrucciones. En estos casos obtenemos resultados dispares, siendo los valores para el número de accesos al último nivel de caché mayores en el Core2 que en el Atom en todos los *benchmarks* y destacando de estos el caso de **mcf**, en el que el valor llega hasta los 39 accesos por ciclo. Sin embargo, para el número de fallos cabe destacar que **astar**, **hmmer**, **bzip2** y **h264ref** tienen valores por encima en el Atom que en el Core2, siendo nuevamente **mcf** para el que más fallos se registran.

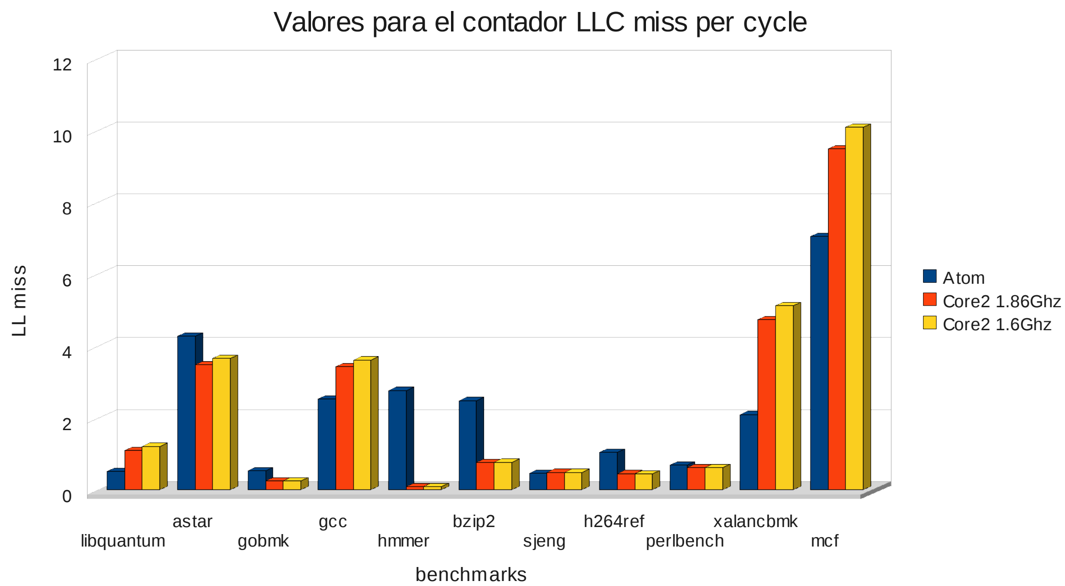


Figura 18: valores obtenidos para el número de fallos en el último nivel de caché por ciclo en el Intel Atom y en el Core2 con diferentes frecuencias para el conjunto de benchmarks que pertenecen a CINT2006.

Las medidas obtenidas para el número de fallos en ITLB y en DTLB por cada millón de instrucciones se representan en las figuras 19 y 20. En la primera (figura 19) podemos destacar como en Core2 apenas se producen fallos en ITLB siendo sorprendente el número de fallos obtenidos en el Atom, sobre todo para el caso de **xalancbmk**, **perlbench**, **sjeng**, **gcc**, **h264ref** y **gobmk**. Para el caso de fallos en DTLB se puede percibir como los valores son superiores en el Core2, destacando a **mcf**. En el caso de **libquantum** y **hmmer** apenas son apreciables.

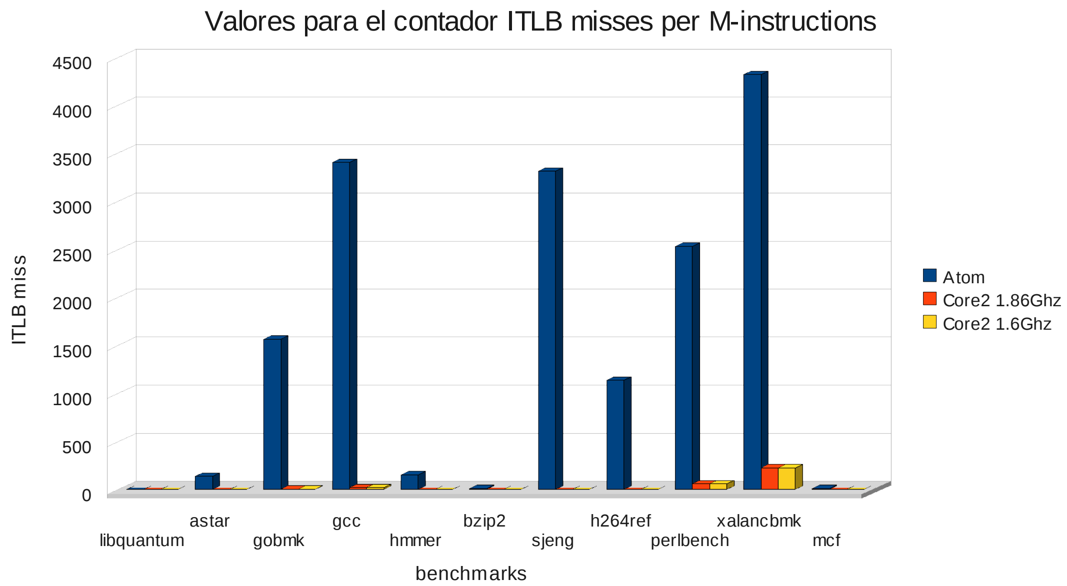


Figura 19: valores obtenidos para el número de fallos en ITLB por millón de instrucciones en el Intel Atom y en el Core2 con diferentes frecuencias para el conjunto de benchmarks que pertenecen a CINT2006.

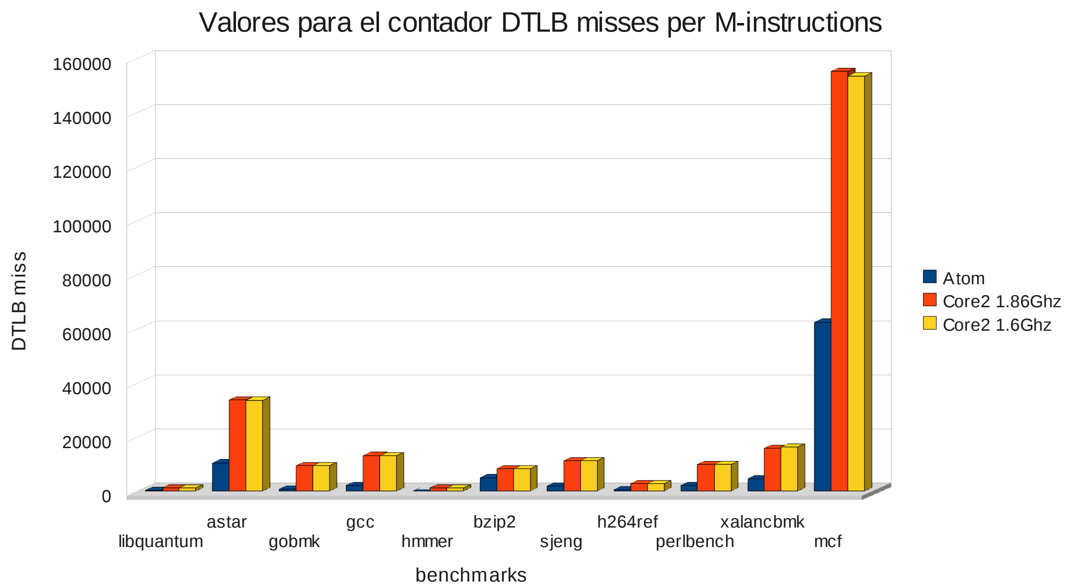


Figura 20: valores obtenidos para el número de fallos en DTLB por millón de instrucciones en el Intel Atom y en el Core2 con diferentes frecuencias para el conjunto de benchmarks que pertenecen a CINT2006.

Para terminar con esta sección, mostramos en la figura 21 el porcentaje de instrucciones en

punto flotante para cada uno de los *benchmarks* de enteros. Cabe destacar el *benchmark* **gcc**, para el cual se obtiene en el Atom un porcentaje mucho mayor que en el Core2. Para el resto, salvo para **gcc**, el porcentaje de estas instrucciones es imperceptible.

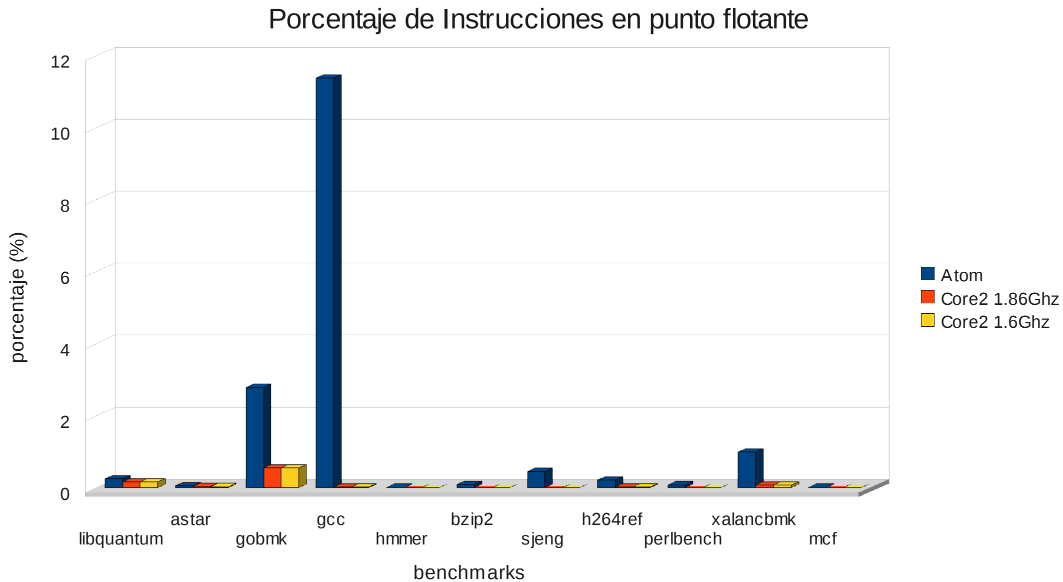


Figura 21: porcentaje de instrucciones en punto flotante para el Intel Atom y en el Core2 con diferentes frecuencias para el conjunto de benchmarks que pertenecen a CINT2006.

3.1.2. Medidas obtenidas para CFP2006

Una vez terminada la exposición de los resultados obtenidos a la hora de calcular las medidas asociadas a los contadores hardware para los *benchmarks* de enteros, nos disponemos a mostrar los valores adquiridos para los de punto flotante, pertenecientes a CFP2006. Al igual que para el caso de los SPEC pertenecientes a CINT2006, la primera medida que se muestra en la figura 22 es el número de instrucciones por ciclo. En ella advertimos que el valor del IPC obtenido en todos los casos es mayor en las ejecuciones en el Core2 , obteniendo valores muy similares con frecuencias distintas, y valores mucho menores en el caso de ejecutar los *benchmarks* en el Atom. Destacan los valores para **calculix**, siendo su IPC cercano a 2.

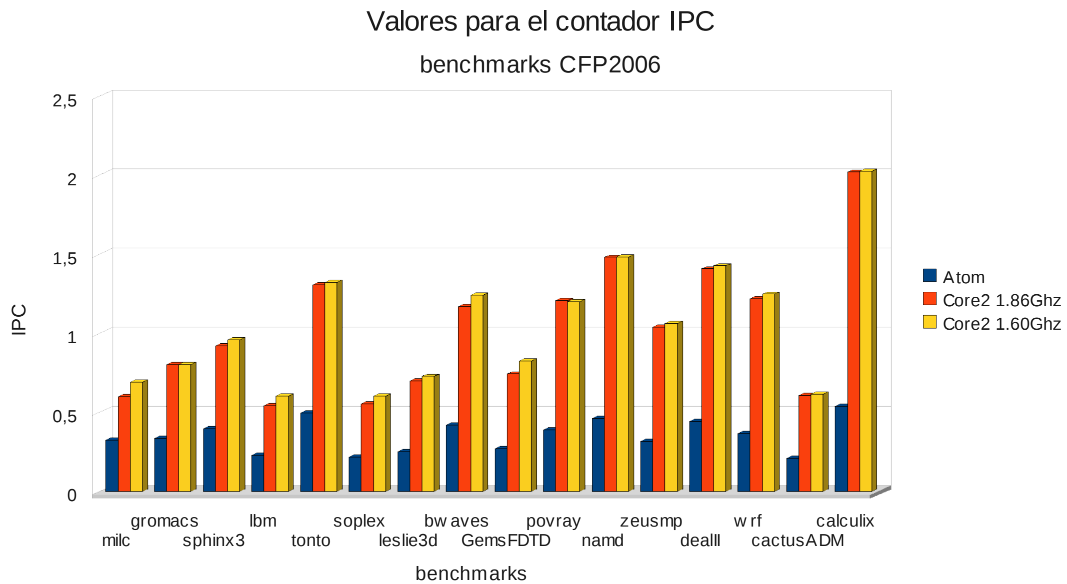


Figura 22: Instrucciones por ciclo para los *benchmarks* de punto flotante.

La figura 23 expone los resultados obtenidos para el número de instrucciones de salto por cada mil instrucciones, en la que se puede observar como los *benchmarks* **soplex**, **povray** y **dealII** son los que más instrucciones de este tipo poseen. Además, como puede percibirse, para cada uno de los programas, las ejecuciones en el Atom y en el Core2 muestran resultados idénticos.

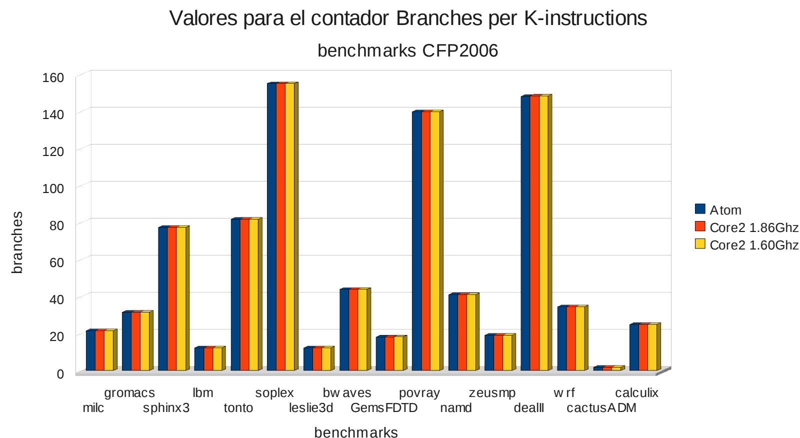


Figura 23: Valores obtenidos para *benchmarks* de punto flotante teniendo en cuenta el número de instrucciones de salto por cada mil instrucciones.

Si nos referimos al número de accesos al último nivel de cache por cada mil instrucciones,

la figura 24 expone los resultados adquiridos. Como puede observarse, en el caso de **bwaves** los valores obtenidos en el Atom y en el Core2 son similares. Para el resto de *benchmarks* se puede contemplar como son mayores los valores obtenidos en el Atom que en el Core2. En el caso de **zeusmp** se puede ver como el número de accesos es muy superior en el Atom, llegando a ser casi el doble que en el Core2, sin embargo, para **calculix** los resultados obtenidos son los menores de los recolectados, obteniendo valores inferiores a 10 en los tres casos (Atom, Core2 a 1.86Ghz y a 1.60Ghz).

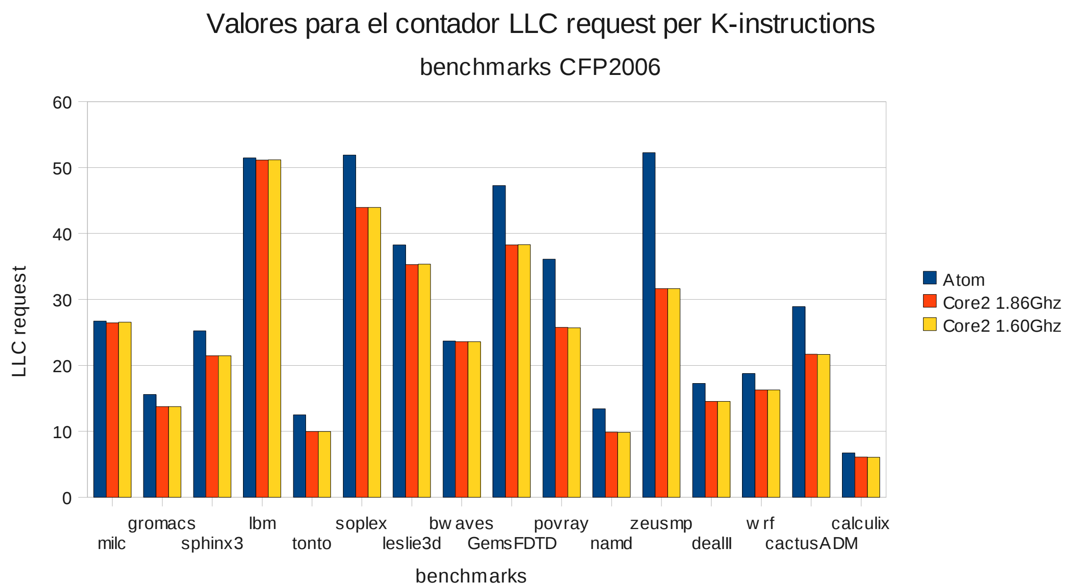


Figura 24: Número de accesos al último nivel de cache para los *benchmarks* de punto flotante pertenecientes a CFP2006.

Como hemos estudiado para el caso de los CINT2006, otra de las medidas que se han tenido en cuenta para los CFP2006 ha sido el número de fallos en el último nivel de cache por cada mil instrucciones de cada uno de los *benchmarks* pertenecientes a este grupo. Como puede apreciarse en la figura 25, para los *benchmarks* **povray** y **namd** los valores son apenas apreciables. En el caso de **lbm** vemos como el número de fallos en LLC es mucho mayor cuando se ejecuta en el Core2 que cuando su ejecución se realiza en el Atom. Para el resto de casos, las ejecuciones en el Core2 resultan con menos fallos en este nivel de la cache que las realizadas en el Atom.

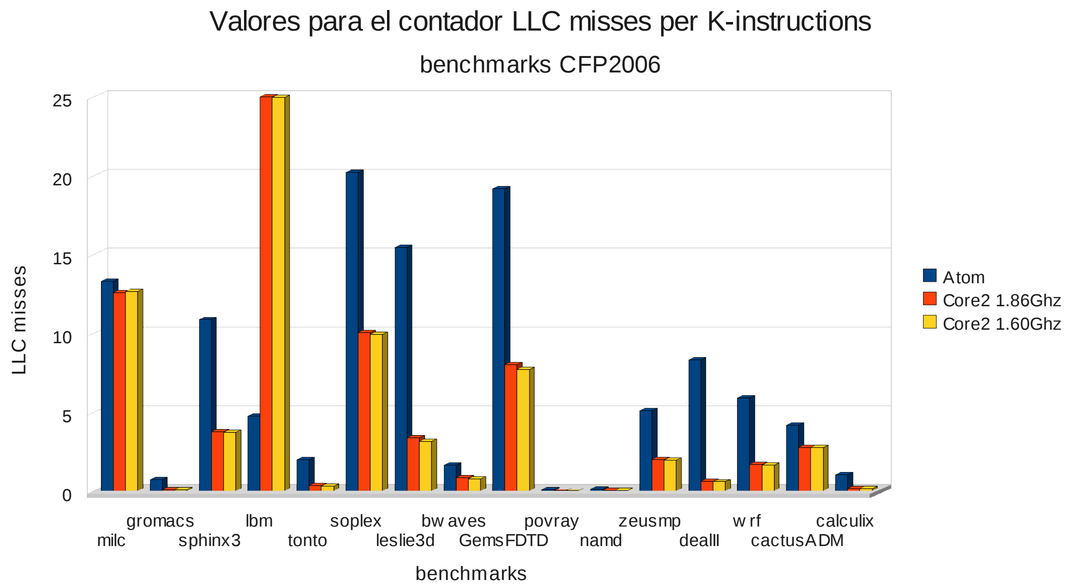


Figura 25: Número de fallos en el último nivel de cache para los *benchmarks* de punto flotante pertenecientes a CFP2006.

La figura 26 expone los valores para el número de saltos mal predichos por cada mil instrucciones, en la que se puede observar como los valores obtenidos para este grupo de *benchmarks* en el caso de ejecutarlos en un Core2 son bajos, llegando a ser en algunos casos (**milc**, **lbm**, **cactusADM**) casi inapreciables. El factor común notado en todos los casos es que se obtienen valores superiores para esta medida en las ejecuciones realizadas en el Atom.

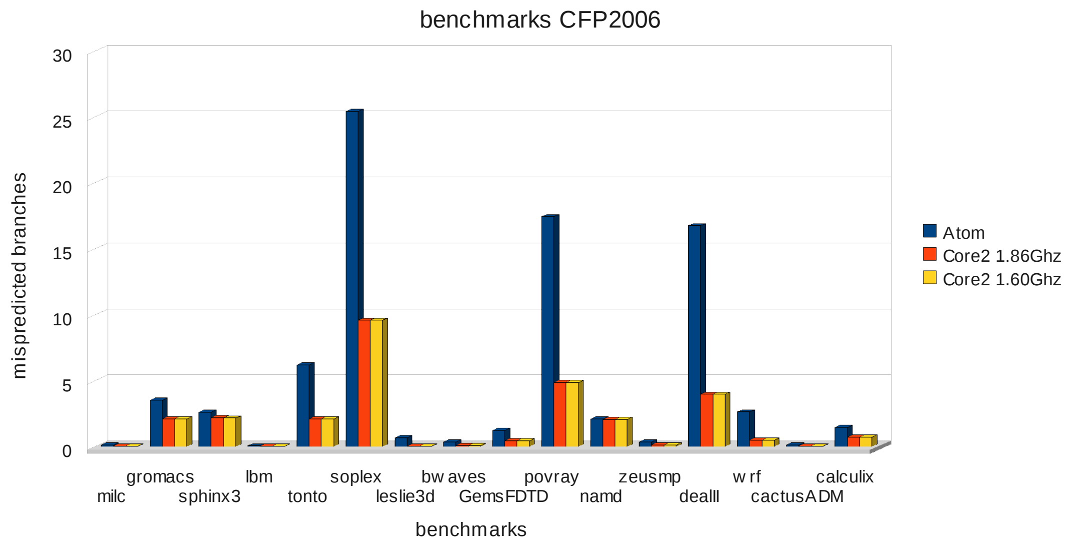


Figura 26: Valores obtenidos al calcular el número de saltos mal predichos por cada mil instrucciones en el Atom y en el Core2 para los *benchmarks* de punto flotante pertenecientes a CFP2006.

Si en vez de observar el número de accesos y fallos al último nivel de cache por cada mil instrucciones lo hacemos por cada ciclo, los resultados obtenidos se pueden contemplar en las figuras 27 y 28. En la primera de ellas (figura 27) notamos como los valores obtenidos para el número de accesos a LLC por ciclo es superior en las ejecuciones realizadas en el Core2 (independientemente de la frecuencia a la que ponemos este) que en las efectuadas en el Atom. En la mayoría de los casos, estos valores son algo superiores cuando el Core2 tiene una frecuencia de 1.60Ghz. En la figura 28, que muestra los fallos a LLC por ciclo, podemos ver que en el caso de **dealII**, **leslie3d** y **sphinx3** el número de fallos es mayor en el Atom que en el Core2, al contrario que en el resto de *benchmarks* que aparecen en la gráfica. Destaca el caso de **lbm**, en el que como podemos observar, el número de fallos al ejecutarlo en el Core2 supera con creces al obtenido en el Atom. En el caso de los *benchmarks* **gromacs**, **povray** y **namd** los valores obtenidos son muy pequeños, estando estos por debajo de 1.

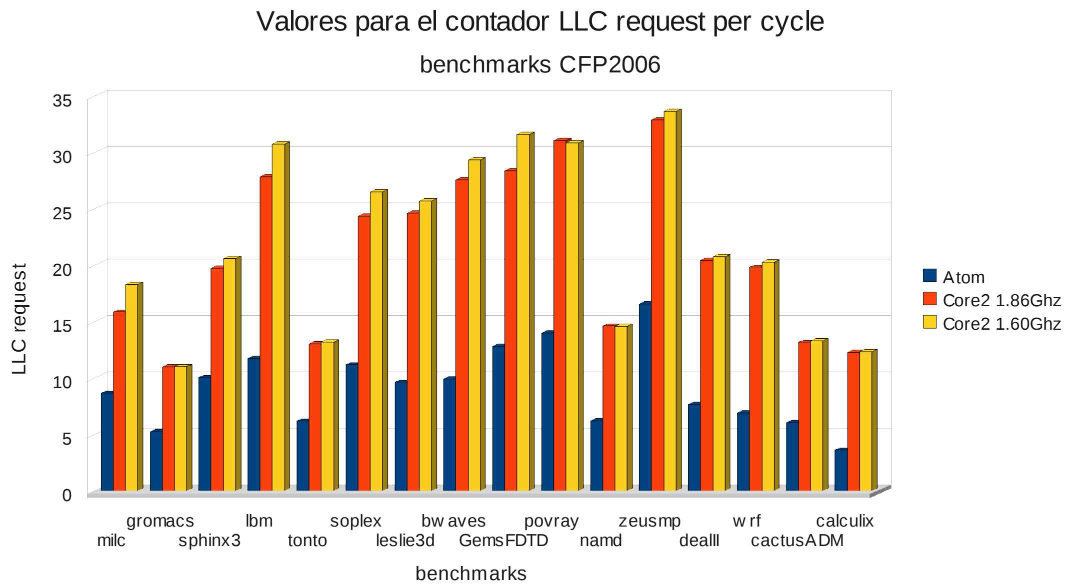


Figura 27: Número de accesos a LLC por ciclo en el Atom y en el Core2 para los *benchmarks* de punto flotante pertenecientes a CFP2006.

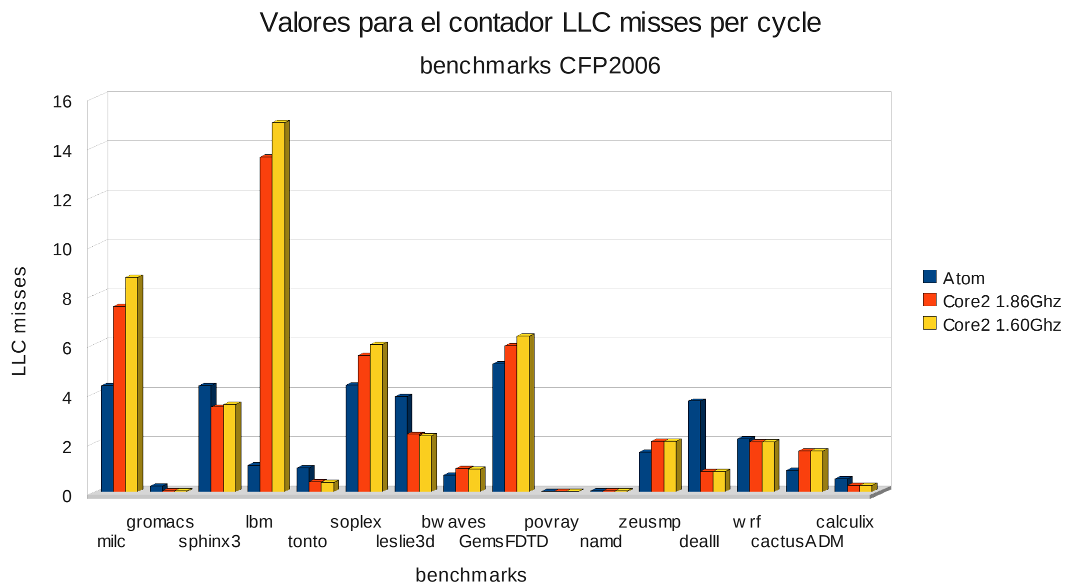


Figura 28: Número de fallos en LLC por ciclo en el Atom y en el Core2 para los *benchmarks* de punto flotante pertenecientes a CFP2006.

Para terminar con el estudio de las medidas obtenidas a partir de los contadores hardware

para el grupo de *benchmarks*, pertenecientes a CFP2006, las figuras 29 y 30 muestran los valores para los fallos en ITLB y DTLB por cada millón de instrucciones respectivamente y la figura 31 muestra el porcentaje de instrucciones en punto flotante que poseen cada uno de los programas. En la primera de este grupo de gráficas (figura 29) podemos observar como el número de fallos en ITLB es inexistente o inapreciable para las ejecuciones en el Core2. La figura 30, que mostraba el número de fallos en DTLB por cada millón de instrucciones, deja patente que para las ejecuciones en el Atom se producen muchos menos fallos que para las realizadas en el Core2 para todos los casos, siendo muy grande la diferencia en muchos de estos como por ejemplo en el **cactusADM** o **povray**.

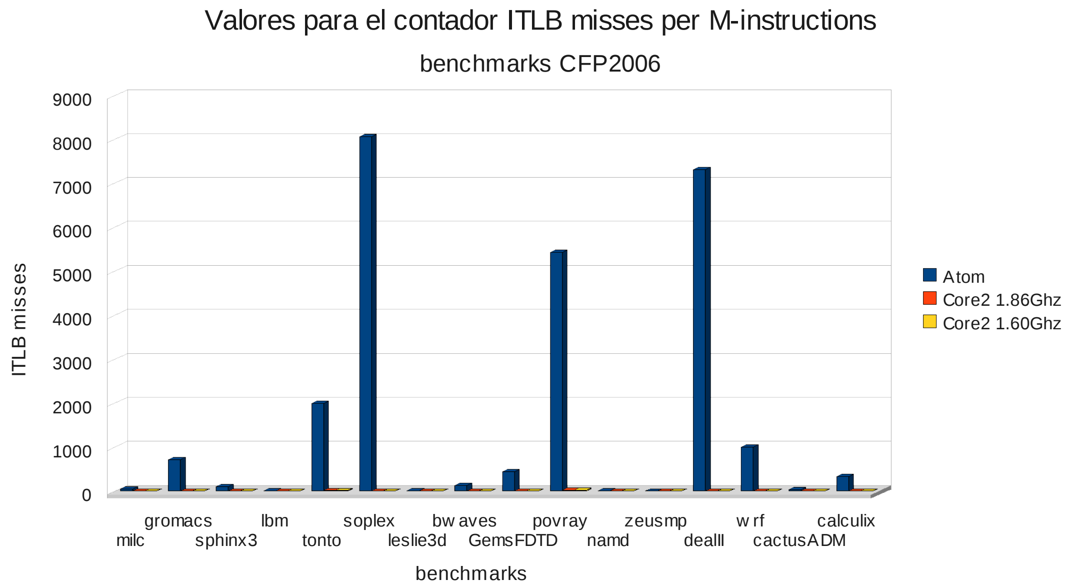


Figura 29: Número de fallos en ITLB por millón de instrucciones en el Atom y en el Core2 para los *benchmarks* de punto flotante pertenecientes a CFP2006.

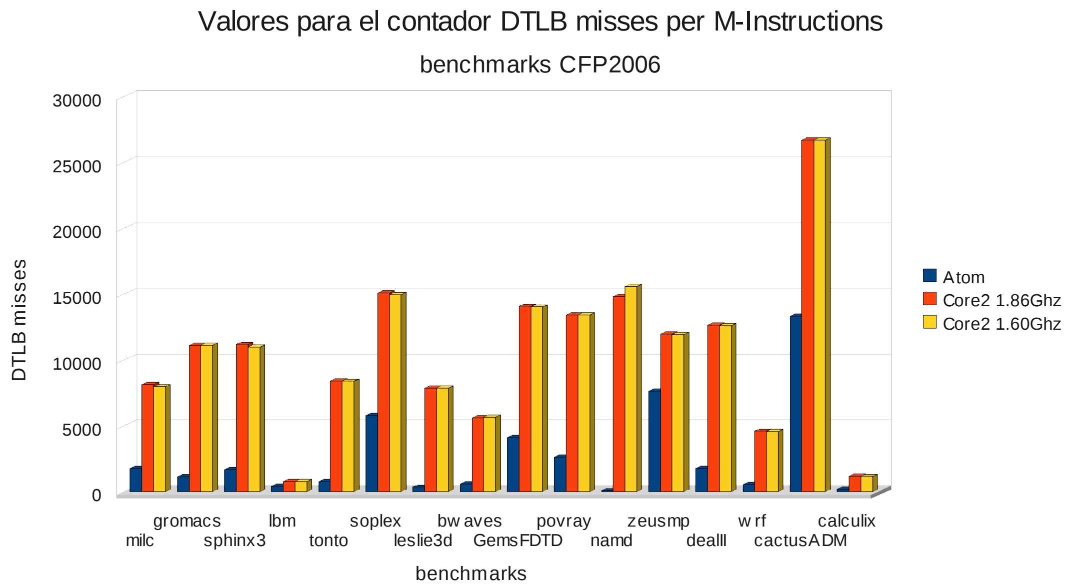


Figura 30: Número de fallos en DTLB por millón de instrucciones en el Atom y en el Core2 para los *benchmarks* de punto flotante pertenecientes a CFP2006.

Para el porcentaje de instrucciones en punto flotante, la figura 31 evidencia los datos obtenidos tanto en el Atom como en el Core2 con frecuencias igual a 1.86Ghz y 1.60Ghz. En ella puede observarse que este porcentaje es mayor en las ejecuciones realizadas en Atom que las llevadas a cabo en el Core2 salvo para el *benchmark* **wrf** donde dicho porcentaje es muy superior cuando se ejecuta en el Core2.

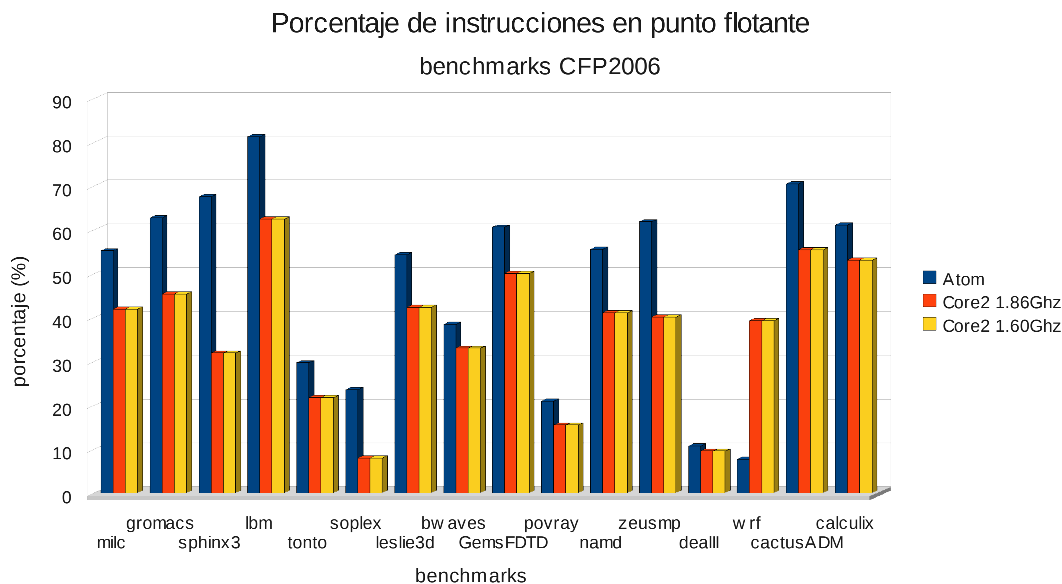


Figura 31: Porcentaje de instrucciones en punto flotante para los *benchmarks* pertenecientes a CFP2006 en el Atom y en el Core2.

Después de exponer los resultados obtenidos para cada una de las medidas obtenidas a partir de los contadores hardware utilizados en este trabajo, la siguiente sección describe y muestra los resultados obtenidos al aplicar el algoritmo de *clustering k-means*.

3.2. Clasificación realizada con k-means

Como el objetivo de este trabajo es poder inferir el beneficio o deterioro relativo que puede obtener una aplicación al ejecutarse en un determinado tipo de *core* (rápido o lento) hemos agrupado los *benchmarks* haciendo uso del algoritmo de k-medias o *k-means clustering* y se han estudiado los resultados obtenidos para diferentes valores de *k*. El proceso seguido se detalla a continuación:

- Con las medidas recogidas a partir de los contadores hardware obtenidos para cada uno de los *benchmarks* en cada una de las arquitecturas, dividimos estos en dos grandes grupos para estudiar por separado como se ha descrito en la sección 3.1.

- Una vez separados los benchmark, ejecutamos en Matlab el algoritmo de *k-means clustering* para los *benchmarks* del primer y segundo grupo por separado. Para ello, se crea una matriz en dicho programa donde cada fila es un *benchmark* y cada columna una medida de los contadores hardware. Este proceso se realiza tres veces ya que los valores de las medidas son distintos dependiendo de la arquitectura en la que se hayan ejecutado los *benchmarks*.

- Tomamos como base los grupos creados en el procesador *Intel Atom*. Es por ello por lo que se calculan los grupos formados para los valores de las medidas obtenidas en

él y se comparan con los grupos creados tanto para el *Core2* a 1.86Ghz como para el *Core2* a 1.60Ghz comparando los elementos comunes para poder generar los grupos.

- Una vez creados estos grupos, obtenemos el speedup medio de cada uno de ellos con la intención de cumplir el objetivo inicial.

3.2.1. Clasificación con k-means para CINT2006 haciendo uso de todas las medidas

A continuación exponemos los resultados obtenidos al realizar las pruebas para el conjunto de *benchmarks* de enteros. La figura 32, junto al cuadro 1 muestran los grupos creados por *k-means* comparando las medidas obtenidas a partir de los contadores hardware en el *Intel Atom* con las obtenidas en el *Intel Core2* a 1.86Ghz. Puede observarse que la gran mayoría de ellos pertenecen al primer grupo pero a la hora de discriminar un programa podemos tener en cuenta dos speedup principalmente, el del grupo 1 y el del grupo 3 con valores de 2.6 y 3.52 respectivamente.

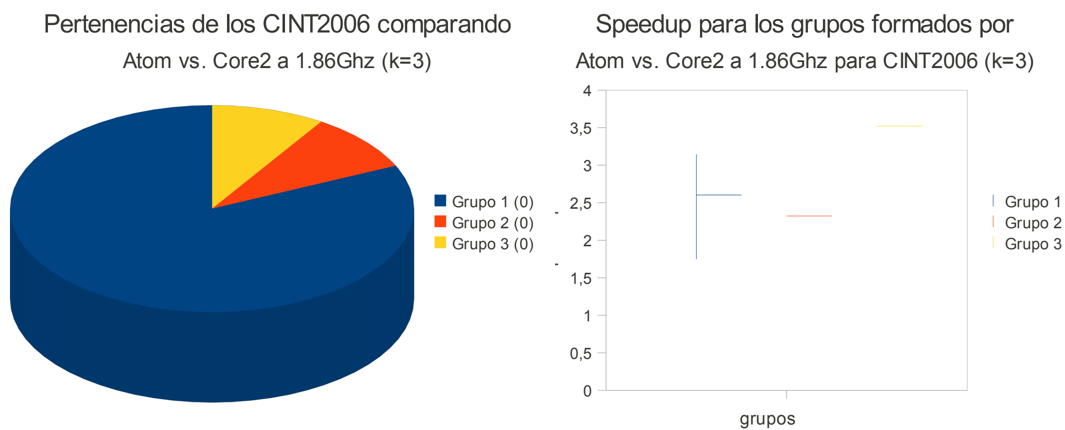


Figura 32: Resultados obtenidos al comparar los grupos creados con las medidas tomadas en el Atom y con los del Core2 a 1.86Ghz para un valor de k=3 utilizando todas las medidas descritas en la sección 2.1. Entre paréntesis se muestran las discrepancias entre grupos, que para este caso, no existen. En la figura de la derecha, el eje de abcisas representa los grupos y el de ordenadas el speedup de cada uno de ellos.

Benchmark	Grupo al que pertenece
libquantum	1
astar	2
gobmk	1
gcc	1
hmmer	1
bzip2	1
sjeng	1
h264ref	1
perlbench	1
xalancbmk	1
mcf	3

Cuadro 1: Grupos creados por k-means con un valor de k=3 y todas las medidas

Buscando una clasificación mejor, ejecutamos el algoritmo *k-means clustering* nuevamente pero esta vez con un valor para $k = 4$. Los resultados se muestran en la figura 33 donde podemos observar que existen discrepancias entre los grupos formados por los valores de las medidas para los *benchmarks* ejecutados en una y otra arquitectura (se muestran entre paréntesis). Además, observamos que los speedup entre 1.6 y 2.8 tienen una alta probabilidad de pertenecer al primer conjunto, siendo los speedup superiores a 3 e inferiores a 3.5 los que podríamos englobar en el conjunto 2. Para el tercer conjunto tendríamos un speedup de 3.52. En el cuadro 2 se muestra que *benchmarks* de CINT2006 pertenecen a cada grupo y como existen discrepancias entre los grupos creados con las medidas tomadas en el Atom y las tomadas en el Core2.

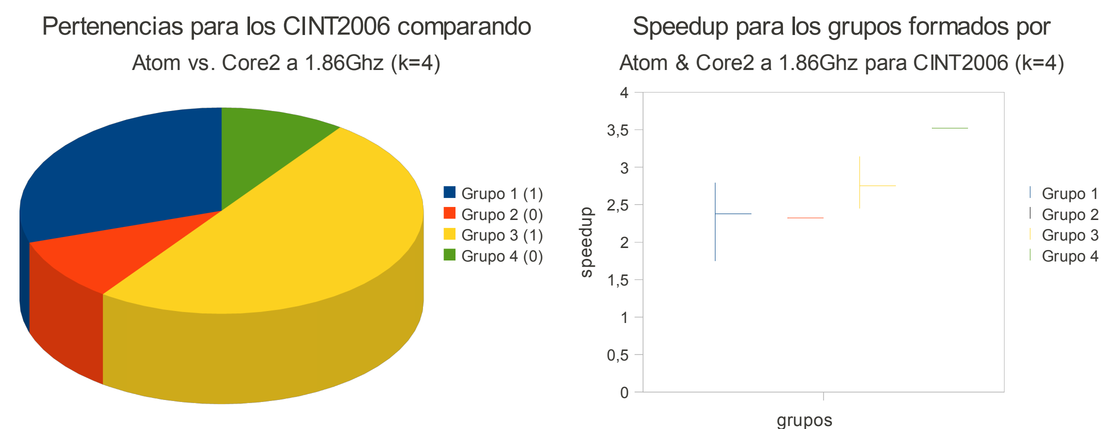


Figura 33: Resultados obtenidos al comparar los grupos creados con las medidas obtenidas a partir de los contadores hardware del Core2 a 1.86Ghz y del Atom para un valor de k=4

Benchmark	Grupos en Atom	Grupos en Core2 a 1.86Ghz
libquantum	1	1
astar	2	2
gobmk	3	1
gcc	3	3
hmmer	1	1
bzip2	3	3
sjeng	3	3
h264ref	1	1
perlbench	3	3
xalancbmk	3	3
mcf	4	4

Cuadro 2: Grupos creados por k-means con un valor de k=4 y todas las medidas

Para continuar con el estudio de la clasificación de los SPEC para inferir el beneficio que puede obtener una aplicación realizamos las pruebas con el valor de k=5 observando los resultados en la figura 34, la cual nos muestra como aumentan las discrepancias entre los grupos creados con las medidas obtenidas en el Atom y las obtenidas en el Core2 (estas se muestran entre paréntesis junto al grupo correspondiente. El valor de cero indica que no existían discrepancias). Observamos también que con una alta probabilidad, valores de speedup comprendidos entre 1.6 y 3 podrían englobarse en un mismo grupo, con lo que llegamos a la conclusión de que estos valores obtenidos para este valor de k no son congruentes.

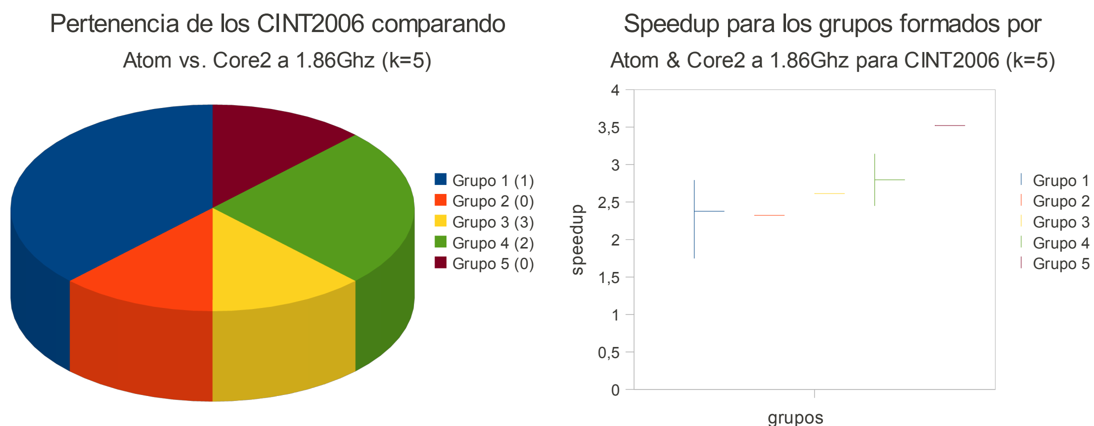


Figura 34: Resultados obtenidos al comparar los grupos creados con las medidas obtenidas en el Atom y con los del Core2 a 1.86Ghz para CINT2006 y un valor de k=5 y donde se pueden observar las discrepancias

Utilizando el *Core2* con una frecuencia igual a 1.60Ghz, los resultados que se obtienen se describen a continuación.. En este caso, también tenemos en cuenta todas las medidas

obtenidas a partir de los contadores hardware a la hora de clasificar los *benchmarks* con el algoritmo de *k-means*. La figura 35 describe los grupos formados y los speedup medios de dichos grupos para un valor de k=3 y en la que se puede observar como no existen discrepancias entre grupos y que los speedup obtenidos son inferiores al caso con el Core2 a 1.86Ghz. Sin embargo, al igual que en su caso homólogo, existen dos grandes grupos de speedup en los que se podría englobar un programa. Es más, los grupos creados para este caso son muy similares a los creados teniendo en cuenta las medidas adquiridas para el Core2 a 1.86Ghz.

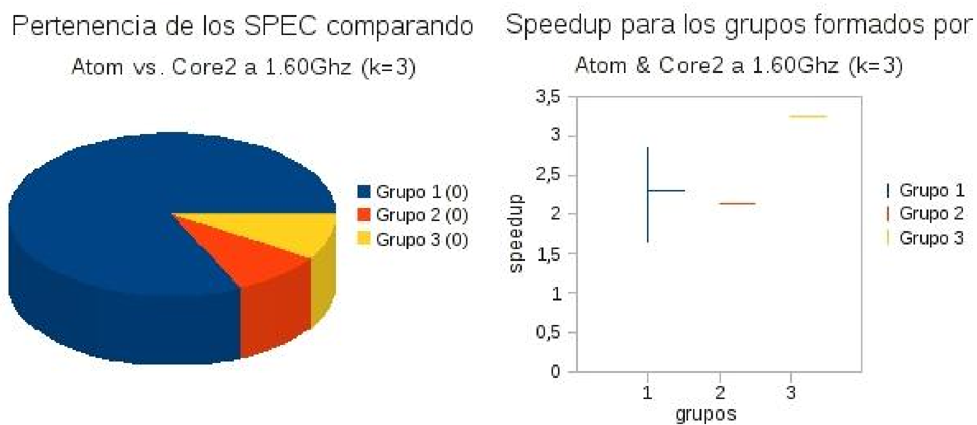
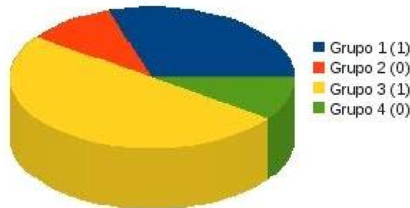


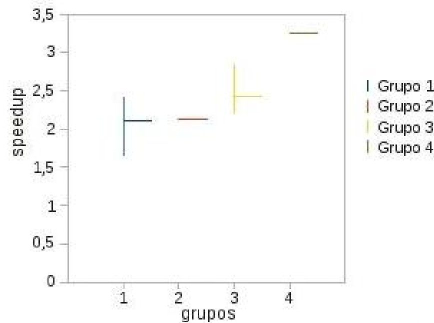
Figura 35: Resultados obtenidos al comparar los grupos creados con las medidas obtenidas en el Atom y con los del Core2 a 1.60Ghz para un valor de k=3 y donde se pueden observar que no existen discrepancias

Las pruebas realizadas para valores de k=4 y k=5 se resumen en la figura 36 que demuestran que con valores de k superiores a 4 empiezan a existir discrepancias entre los grupos, al igual que ocurría para el caso del Core2 a 1.86Ghz. Los speedup de los grupos obtenidos para estos valores de k también pueden analizarse en dicha figura, notando que, para el caso de k=4, los speedup obtenidos por el grupo 2 y para valores bajos del grupo 3 pueden englobarse dentro del grupo 1. En el caso de k=5 ocurre algo similar, pero esta vez podemos incluir dentro del grupo 1 los speedup logrados para los grupos 2, 3 y valores bajos del grupo 4. El último grupo, el 4 o 5 dependiendo del valor de k, es el caso del *benchmark* **mcf**, que siempre lo agrupa en un conjunto distinto al del resto de pruebas.

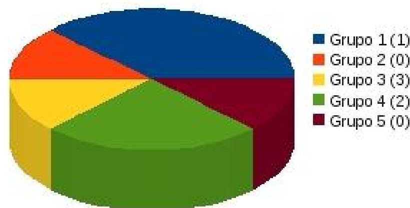
Pertenecia de los SPEC comparando
Atom vs. Core2 a 1.60Ghz (k=4)



Speedup para los grupos formados por
Atom & Core2 a 1.60Ghz (k=4)



Pertenecia de los SPEC comparando
Atom vs. Core2 a 1.60Ghz (k=5)



Speedup para los grupos formados por
Atom & Core2 a 1.60Ghz (k=5)

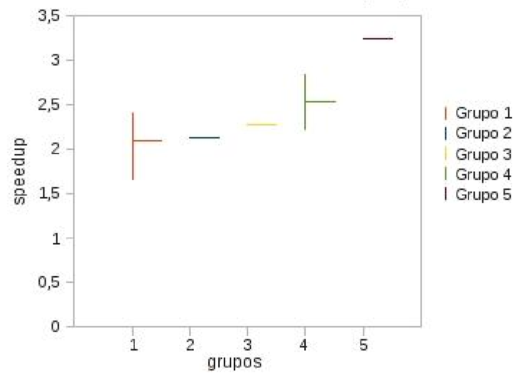


Figura 36: Resultados obtenidos al comparar los grupos creados con las medidas para el Atom y el Core2 a 1.60Ghz para valores de k=4 y k=5 y donde se pueden observar que empiezan a existir discrepancias entre los grupos

3.2.2. Clasificación con k-means para CINT2006 con un conjunto reducido de medidas

Con el objetivo de extender el estudio y saber si con menos medidas la clasificación que se realiza es mejor, las pruebas siguientes las hemos realizado teniendo en cuenta sólo algunas de estas obtenidas a partir de los contadores hardware. Después de probar con distintas combinaciones de medidas, finalmente las escogidas para continuar con el estudio han sido: *Branch instructions per thousand instructions*, *Misspredicted branch instructions per thousand instructions*, *LLC request per thousand instructions*, *LLC miss per thousand instructions* y *Floating point instruction percentage*. Además, podría ser útil, ya que a la hora de obtener las medidas de un programa del cual se desea inferir su factor de ganancia, no sería necesario obtener todas sino un grupo reducido de ellas. Las pruebas realizadas comprenden diferentes valores para el parámetro k, con la finalidad de poder comparar los resultados con

los obtenidos cuando tenemos en cuenta las todas las medidass. La figura 37 muestra los grupos creados con *k-means* con los valores de las cinco medidas anteriormente descritas y obtenidas tanto en el Atom como en el Core2 a 1.86Ghz, y en la cual se puede contemplar como no existen discrepancias entre los grupos creados en una y otra arquitectura. También se observa como, valores del speedup entre 1.7 y 3.1 podrían englobarse en un mismo conjunto, teniendo el otro un valor para el speedup de 3.2.

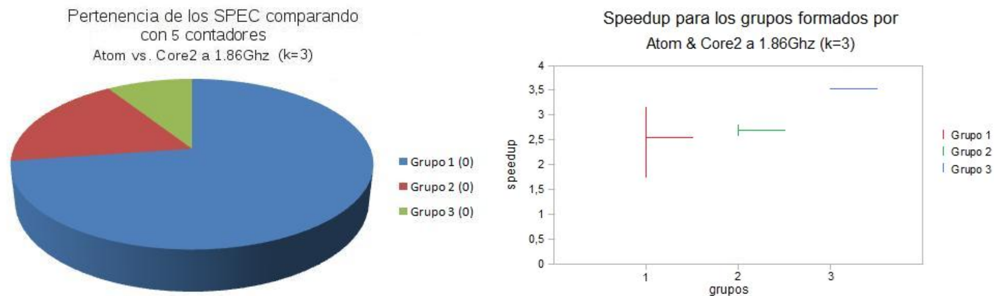


Figura 37: Grupos creados para el Atom y el Core2 a 1.86Ghz y los speedup medios de cada uno de estos grupos con un valor de k=3.

Repetimos las pruebas, pero esta vez para un valor de k=4 con la meta de establecer un valor para el cual los resultados sean razonables. Los resultados de dicho ensayo se observan en la figura 38 en la que se observa como conseguimos mantener la ausencia de discrepancias. Si recordamos la figura 33, en el que se tenían en cuenta todas las medidas, ya aparecían discrepancias entre los grupos, así pues, parece que con menos medidas mejoramos en este sentido. Otro aspecto a destacar es que aparece un nuevo rango de speedup, teniendo el primer conjunto un intervalo de speedup entre 1.6 y 2.6, el segundo un intervalo entre 2.6 y 3.2 y el tercer conjunto uno de 3.5, lo que parece que mejora también los resultados obtenidos cuando se utilizaban todas las medidas.

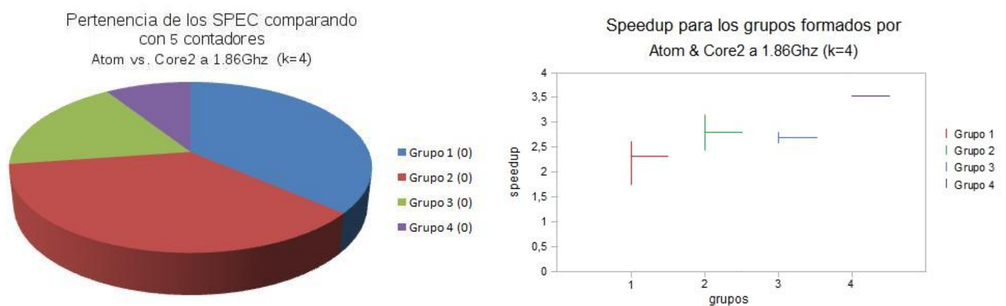


Figura 38: Grupos creados por *k-means* con k=4 para el Atom y el Core2 a 1.86Ghz y los speedup medios de cada uno de estos grupos.

Con el fin de continuar comparando los resultados obtenidos en el caso de estudio anterior,

el de haber tenido en cuenta todas las medidas, con este, en el que tenemos en cuenta solo algunas de ellas, la figura 39 presenta dichos resultados para el caso en el que el valor de k es 5. Se puede ver que el número de discrepancias entre los grupos formados en arquitecturas diferentes sigue siendo igual a cero para todos los grupos pero, con respecto a los speedup que caracterizan a cada grupo, no obtenemos mejoras respecto al caso de k=4, ya que seguimos teniendo un primer conjunto con un intervalo entre 1.6 y 2.6 (que incluye al grupo 1, al grupo 3 y a los valores bajos del grupo 2), un segundo con un intervalo entre 2.6 y 3.2 (incluye a los valores altos del grupo 2 y al grupo 4) y un tercer conjunto con un valor de 3.5 (donde solo se encuentra el grupo 5).

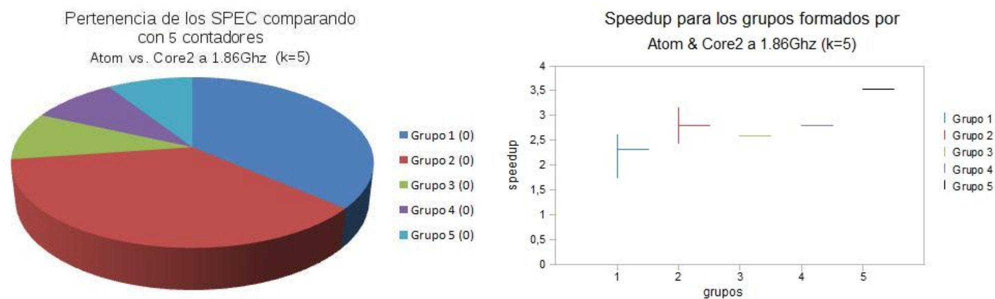


Figura 39: Grupos creados para el Atom y el Core2 a 1.86Ghz y los speedup medios de cada uno de estos grupos considerando un valor de k=5.

Con estos resultados podemos concluir que el valor de k para el que se obtienen los resultados más razonables es para k=4.

Al igual que se hizo cuando tuvimos en cuenta todas las medidas, en este caso también estudiamos los resultados obtenidos al comparar el Atom con el Core2 a 1.60Ghz con el deseo de completar el estudio. Las figuras 40,41 y 42 ofrecen estos resultados, que si los analizamos resulta que son similares al caso expuesto en el que se comparaban el Atom con el Core2 a 1.86Ghz con el conjunto reducido de medidas obtenidas.

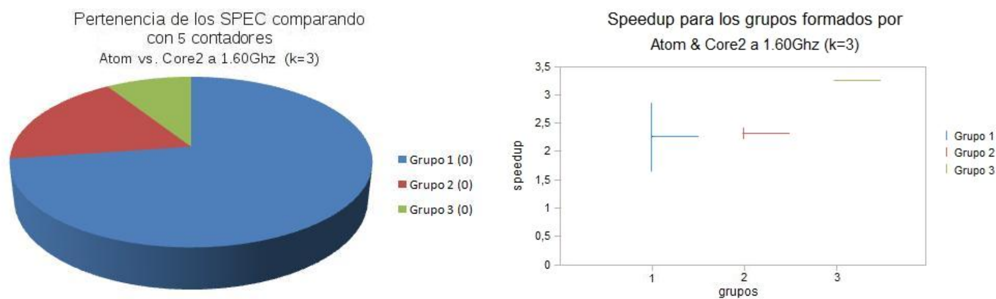


Figura 40: Grupos creados para el Atom y el Core2 a 1.6Ghz y los speedup medios de cada uno de estos grupos considerando un valor de k=3.

Para este caso (figura 40) obtenemos dos conjuntos de speedup, el formado por los grupos 1 y 2 que comprenden un intervalo entre 1.6 y 3.2, y el formado por el grupo 3 con un valor del speedup de 3.2.

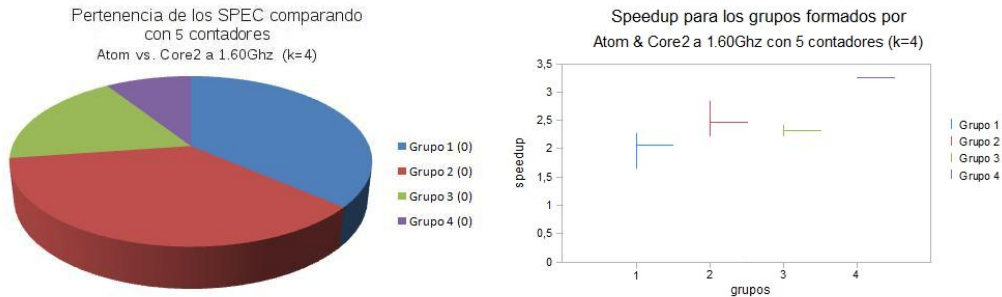


Figura 41: Grupos creados para el Atom y el Core2 a 1.6Ghz y los speedup medios de cada uno de estos grupos con un valor de k=4.

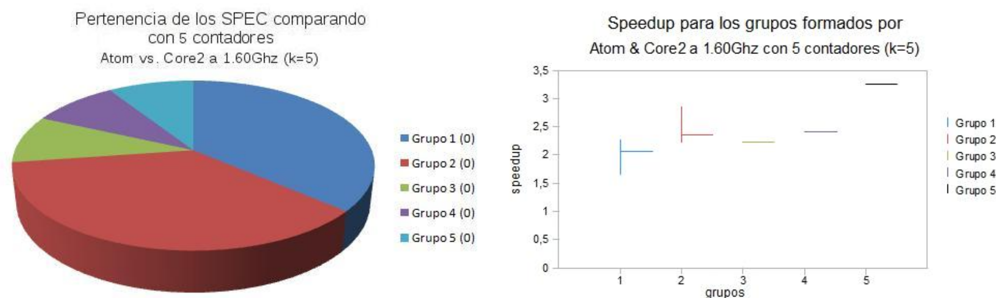


Figura 42: Grupos creados por *k-means* con k=5 para el Atom y el Core2 a 1.6Ghz y los speedup medios de cada uno de estos grupos.

Como comentamos antes, los resultados obtenidos para el caso del Core2 a 1.6Ghz son similares a los alcanzados en el del Core2 a 1.86Ghz, deduciendo de las gráficas que el valor óptimo para k es 4, ya que se obtienen los resultados más razonables. Con estos resultados podemos suponer que el uso de menos medidas beneficia nuestro estudio, con lo que no sería necesario utilizar todas las descritas en este trabajo y así poder hacer uso de un conjunto reducido de ellas con el propósito de inferir el Factor de Ganancia de una determinada aplicación al ejecutarse en un determinado tipo de *core*, que era el objetivo que perseguíamos.

3.2.3. Clasificación con k-means para CFP2006 con todas las medidas

Con la finalidad de completar el estudio realizado, seguimos los pasos planteados en el apartado anterior para los *benchmarks* de punto flotante, es decir, aquellos pertenecientes al

grupo CFP2006. En este caso analizaremos los resultados obtenidos para el mismo valor de k comparando el Atom primero con el Core2 a 1.86Ghz y después con el Core2 a 1.60Ghz. En primer lugar analizaremos dichos *benchmarks* teniendo en cuenta los valores de todas las medidas obtenidas para a continuación repetir las pruebas con tan solo 5 de ellos (*Branch instructions per thousand instructions*, *Misspredicted branch instructions per thousand instructions*, *LLC request per thousand instructions*, *LLC miss per thousand instructions* y *Floating point instruction percentage*).

Los primeros resultados que presentamos son los alcanzados para un valor de k=3 y se muestran el las figuras 43 y 44 para las comparaciones con el Core2 a 1.86Ghz y a 1.60Ghz respectivamente. En ambas observamos a la izquierda los grupos formados comparando los grupos creados para el Atom y para el Core2 y escogiendo los comunes. Es por ello por lo que aparecen discrepancias entre ambos (se muestran estas entre paréntesis junto al nombre del grupo). A la derecha percibimos los speedup de cada grupo viendo en ambos casos como el calculado para el grupo 1 engloba al resto. Es por ello por lo que no obtenemos resultados que podamos considerar como congruentes.

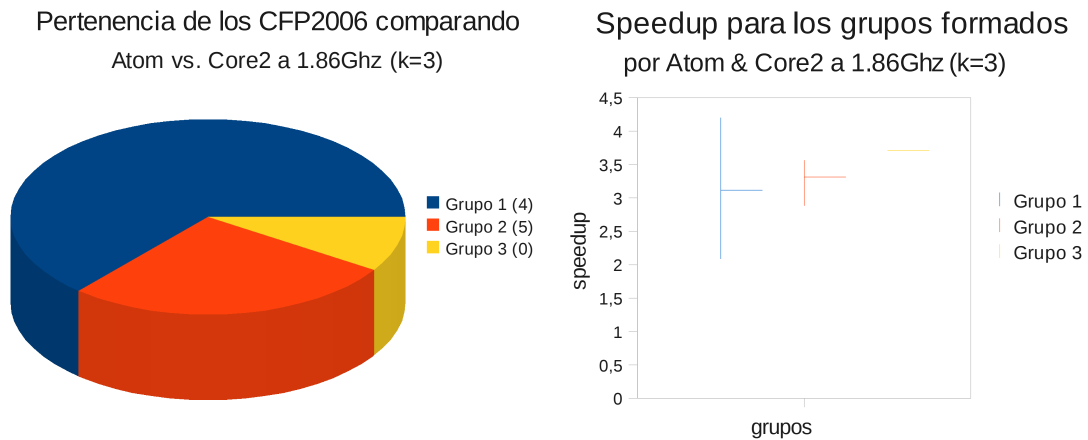
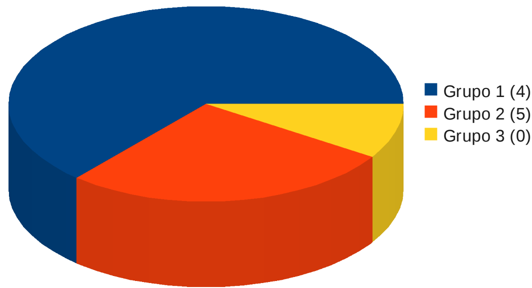


Figura 43: Grupos creados por *k-means* con k=3 para el Atom y el Core2 a 1.86Ghz y los speedup medios de cada uno de estos grupos para los *benchmarks* pertenecientes a CFP2006.

Pertenencia de los CFP2006 comparando
Atom vs. Core2 a 1.60Ghz (k=3)



Speedup para los grupos formados por
Atom & Core2 a 1.60Ghz (k=3)

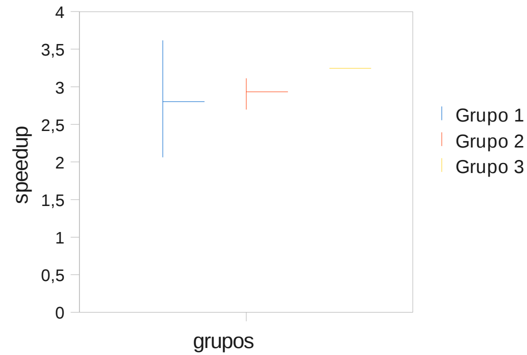
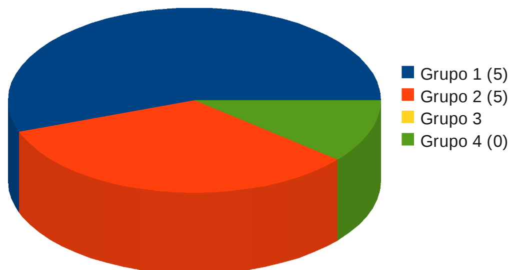


Figura 44: Grupos creados por *k-means* con k=3 para el Atom y el Core2 a 1.6Ghz y los speedup medios de cada uno de estos grupos para los *benchmarks* pertenecientes a CFP2006.

Si realizamos el mismo ensayo con *k-means* pero esta vez con un valor de k=4, el speedup y los agrupamientos obtenidos para este caso se observan en las figuras 45 y 46. En el primer conjunto de gráficas (figura 45) se detecta que el número de divergencias entre los grupos creados por el Atom y los creados por el Core2 a 1.86Ghz aumenta. Asimismo, no existen coincidencias para englobar los *benchmarks* que pertenecen a un determinado grupo en el Atom con los pertenecientes a otro en el Core2, es decir, existe una discrepancia absoluta. Es por ello por lo que a pesar de haber lanzado las pruebas con un valor de k=4 obtenemos solo 3 conjuntos. Los speedup conseguidos para cada grupo pueden englobarse en el primero de ellos, con lo que nos encontramos con el problema del caso anterior.

Pertenencia de los CFP2006 comparando
Atom vs. Core2 a 1.86Ghz (k=4)



Speedup para los grupos formados por
Atom & Core2 a 1.86Ghz (k=4)

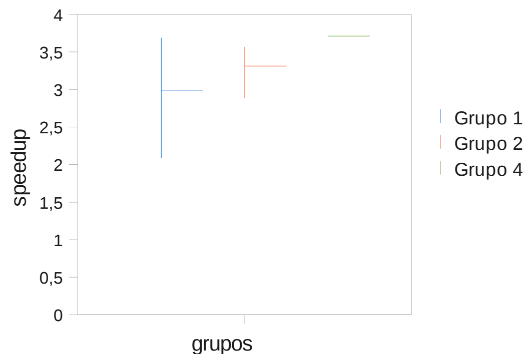


Figura 45: Grupos creados por *k-means* con k=4 para el Atom y el Core2 a 1.86Ghz y los speedup medios de cada uno de estos grupos para los *benchmarks* pertenecientes a CFP2006.

Si examinamos los resultados obtenidos para el caso en el que se comparan los grupos con

el Core2 a 1.60Ghz (figura 46) vemos las mismas características aparecidas para el caso anterior, en la que solo existen coincidencias para tres de los 4 grupos que deberían crearse, existiendo también discrepancias dentro de ellos. Si nos referimos a los speedup, la conclusión es similar al caso del Core2 a 1.86Ghz y k=4, ya que uno de los grupos abarcaría al resto.

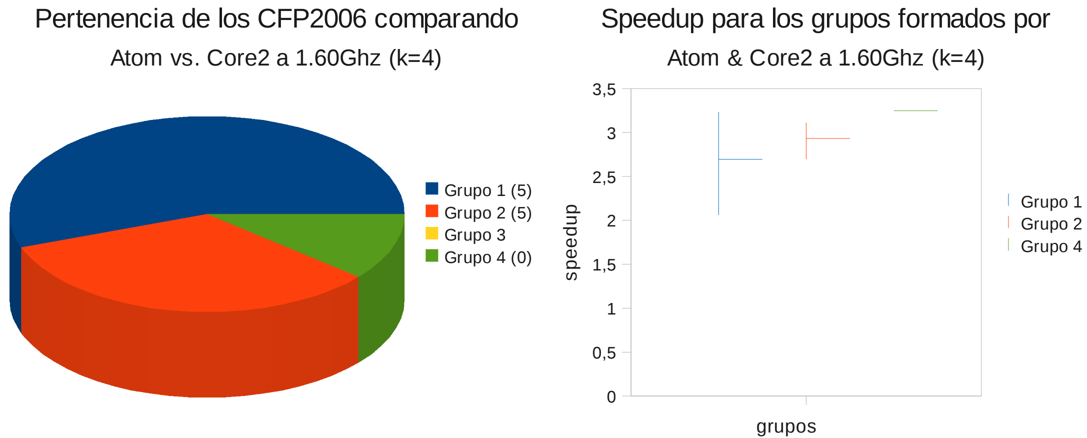


Figura 46: Grupos creados para el Atom y el Core2 a 1.6Ghz y los speedup medios de cada uno de estos grupos con un valor de k=4 para los *benchmarks* pertenecientes a CFP2006.

Para finalizar con el estudio teniendo en cuenta todos las medidas obtenidas a partir de los contadores hardware, exponemos los resultados obtenidos para el caso en el que k=5. La figura 47 evidencia los resultados obtenidos para el caso anterior, obteniendo en este caso 4 grupos en lugar de 5. También observamos que se mantienen las discrepancias entre algunos grupos y que los speedup podrían englobarse todos en un grupo.

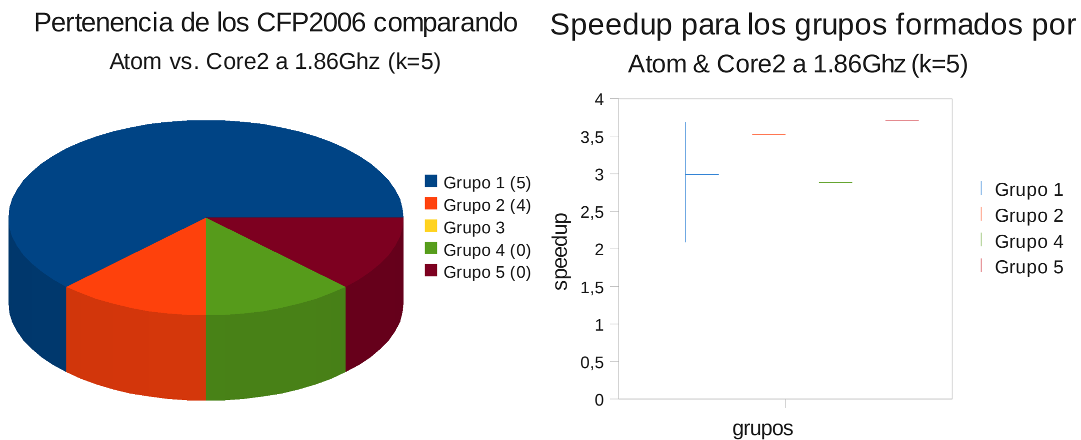


Figura 47: Grupos creados por *k-means* con k=5 para el Atom y el Core2 a 1.86Ghz y los speedup medios de cada uno de estos grupos para los *benchmarks* pertenecientes a CFP2006.

Los resultados obtenidos para el caso del Core2 a 1.60Ghz se muestran en la figura 48. Observamos como estos son similares al caso anterior, donde aparecen discrepancias para algunos grupos y no todos ellos están completos (para el grupo 3 no hay coincidencias). En el caso de los speedup también vemos como un solo grupo incluye al resto.

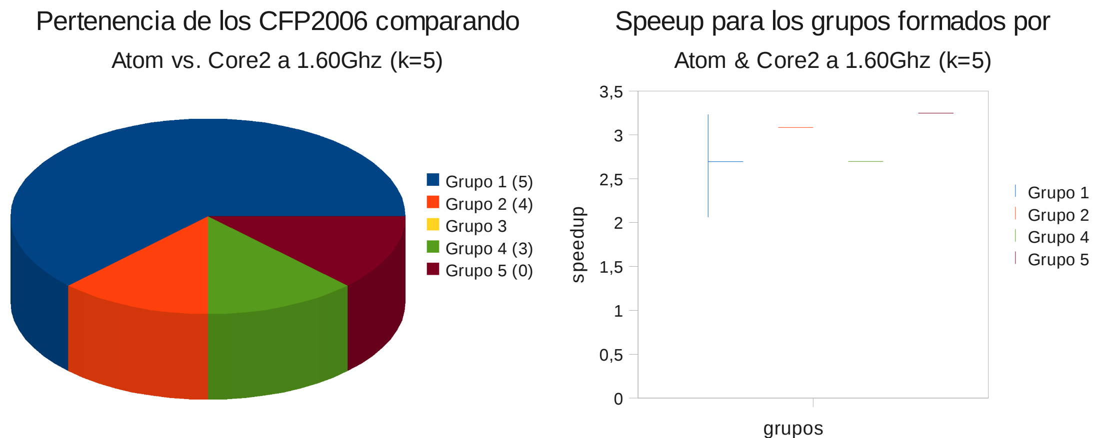
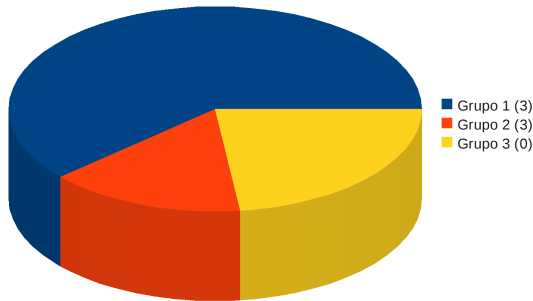


Figura 48: Grupos creados por *k-means* con k=5 para el Atom y el Core2 a 1.60Ghz y los speedup medios de cada uno de estos grupos para los *benchmarks* pertenecientes a CFP2006.

3.2.4. Clasificación con k-means para CFP2006 con un conjunto de las medidas obtenidas

A continuación revelamos los resultados obtenidos para el caso en el que utilizamos el algoritmo de *k-means clustering* para los benchmarks de CFP2006 con las medidas *Branch instructions per thousand instructions*, *Misspredicted branch instructions per thousand instructions*, *LLC request per thousand instructions*, *LLC miss per thousand instructions* y *Floating point instruction percentage* con el objetivo de estudiar si la clasificación mejora, ya que como hemos podido examinar, los resultados para los *benchmarks* de punto flotante con todas las medidas obtenidas no son válidos para nuestro propósito.
Como ha sucedido en todas las pruebas realizadas anteriormente, empezamos este estudio con un valor de *k-means clustering* para un valor de k=3. Las figuras 49 y 50 siguen evidenciando que no somos capaces de obtener resultados que nos sirvan para inferir el Factor de Ganancia en estos casos, a pesar de que hemos reducido el número de medidas. En la figura 49 vemos como el número de discrepancias es menor que en el mismo caso pero teniendo en cuenta todas las medidas obtenidas. También observamos como el speedup de un solo grupo engloba al resto. Esto se debe a que incluye en el mismo grupo al *benchmark* con mayor y menor speedup de todos (**calculix** tiene 4.2 y **milc** tiene 2.09 como puede observarse en la figura 11).

Pertenencia para CFP2006 comparando
Atom vs. Core2 a 1.86Ghz con 5 contadores (k=3)



Speedup para los grupos formados por
Atom vs. Core2 a 1.86Ghz con 5 contadores (k=3)

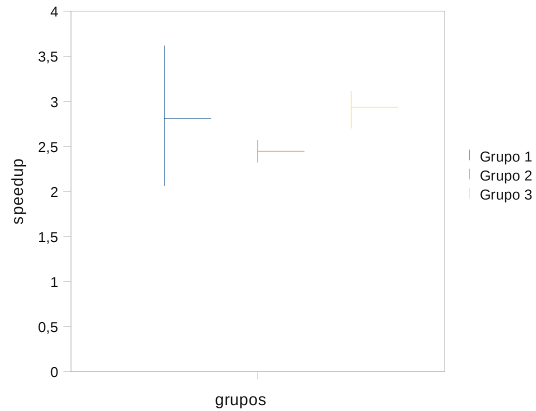
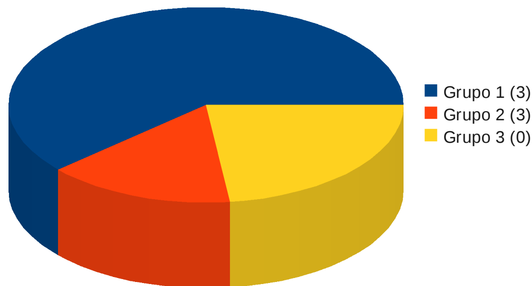


Figura 49: Grupos creados para el Atom y el Core2 a 1.86Ghz y los speedup medios de cada uno de estos grupos con un valor de k=3 para los *benchmarks* pertenecientes a CFP2006.

La figura 50 demuestra resultados similares a los obtenidos en el caso en el que comparábamos con el Core2 a 1.86Ghz, viendo que también existen discrepancias entre grupos, siendo para este caso mayores.

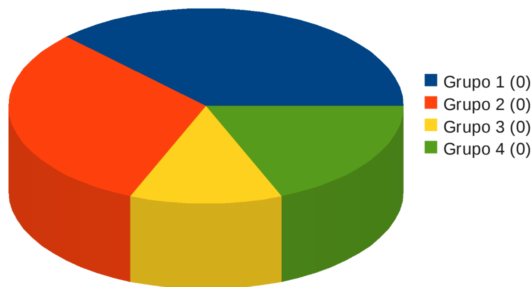
Pertenencia de los CFP2006 comparando
Atom vs. Core2 a 1.60Ghz con 5 contadores (k=3)



Speedup para los grupos formados por
Atom & Core2 a 1.60Ghz con 5 contadores (k=3)

Figura 50: Grupos creados para el Atom y el Core2 a 1.60Ghz y los speedup medios de cada uno de estos grupos con un valor de k=3 para los *benchmarks* pertenecientes a CFP2006.

Siguiendo con el estudio propuesto, los resultados obtenidos para el caso en el que k=4 se muestran en las figuras 51 y 52, en las que se observan estos cuando comparamos el Atom con el Core2 a 1.86Ghz y con el Core2 a 1.60Ghz respectivamente.



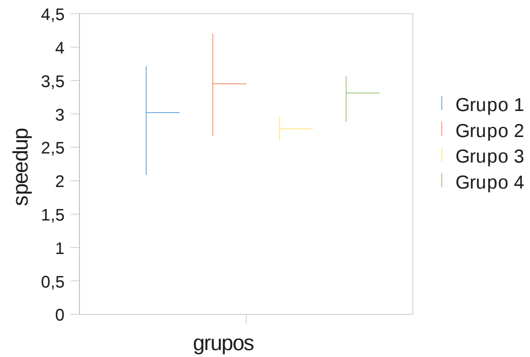
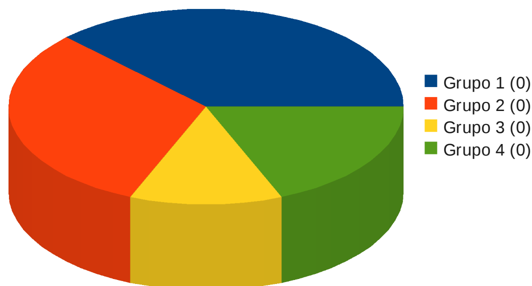


Figura 51: Resultados obtenidos al comparar los grupos creados con el conjunto de medidas obtenidas en el Core2 a 1.86Ghz y con los del Atom para un valor de k=4 para CFP2006

Tanto en la figura 51 como en la 52 observamos que para este caso no existen discrepancias, es decir, los grupos obtenidos con *k-means* en el Atom son idénticos a los obtenidos en el Core2 con 1.86GHz y 1.60Ghz. Sin embargo, el problema de los speedup sigue apareciendo, con lo que sigue sin ser útil esta medida a la hora de inducir el beneficio que se obtiene para una aplicación al ejecutarse en un *core* rápido o lento.



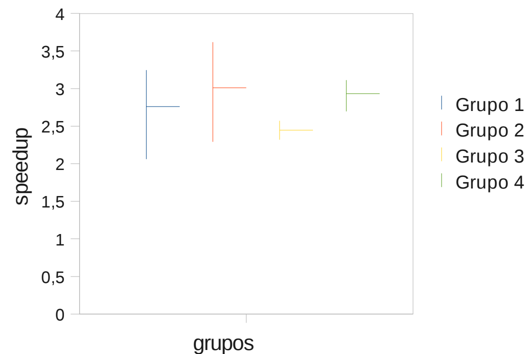


Figura 52: Resultados obtenidos al comparar los grupos creados en el Core2 a 1.60Ghz y con los del Atom para un valor de k=4 para CFP2006

Para acabar con las pruebas realizadas sobre los SPEC pertenecientes a CFP2006 nos disponemos a presentar los resultados logrados con un valor de k=5. La figura 53 nos muestra como vuelven a aparecer las discrepancias entre los grupos creados en el Atom y en el Core2 a 1.86Ghz, no existiendo posibilidad de incluir un quinto grupo como se aprecia en la parte izquierda de la figura. Además, los speedup siguen sin aclararnos a que grupo podría

pertenecer un determinado programa.

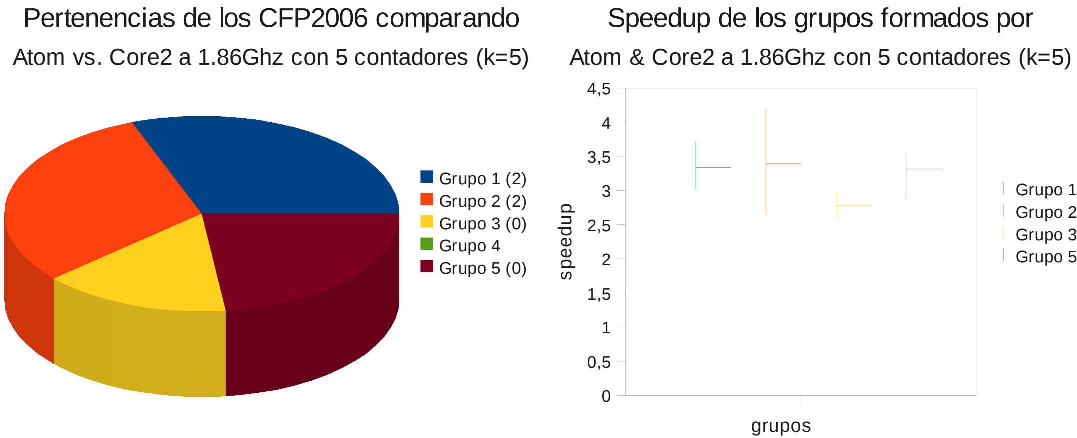


Figura 53: Resultados obtenidos al comparar los grupos creados en el Core2 a 1.86Ghz y con los del Atom para un valor de k=5 para CFP2006

En esta misma línea están los resultados obtenidos con el Core2 a 1.60Ghz y mostrados en la figura 54 donde seguimos sin poder diferenciar conjuntos distintos para el speedup (a la derecha de la imagen) y observando que las discrepancias se mantienen iguales (zona izquierda de la imagen).

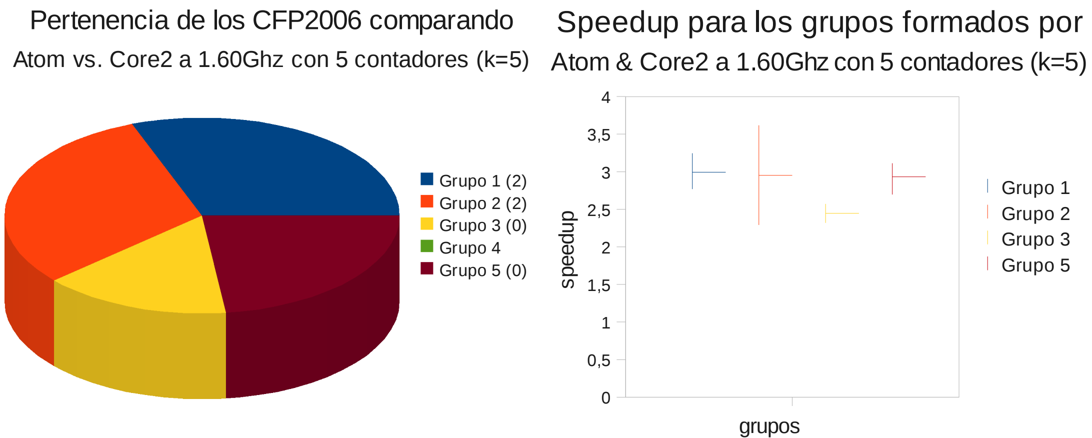


Figura 54: Resultados obtenidos al comparar los grupos creados en el Core2 a 1.60Ghz y con los del Atom para un valor de k=5 para CFP2006

A la vista de estos resultados podemos concluir que este método no es útil para el propósito planteado en el caso de programas de coma flotante, como queda patente en la última sección descrita, dejando esta línea de trabajo abierta con la meta de encontrar algún método que nos ayude a alcanzar el objetivo propuesto para este tipo de programas.

4. Conclusiones y Trabajo Futuro

De los resultados obtenidos en las pruebas realizadas podemos deducir que, como se ha demostrado anteriormente, no es efectivo utilizar todas las medidas descritas en la sección 2.1 a la hora de clasificar los programas de prueba. Es por ello por lo que se ha escogido un grupo reducido de estas, haciendo posible que la clasificación realizada mejore. Otra de las conclusiones que se pueden sacar es que con valores de k igual a 4, conseguimos resultados razonables con el algoritmo de *k-means clustering*, especialmente en el caso en el que utilizamos el conjunto reducido de medidas. Esto es beneficioso a la hora de clasificar un programa, ya que un número reducido de estas, para las que hay que obtener valores nos proporciona una buena medida, con lo que no haría falta sacar el valor de todas las medidas a partir de los contadores hardware para inferir el Factor de Ganancia de ese programa.

Aunque las pruebas realizadas han variado el valor de k entre 3 y 9 se ha decidido mostrar tan solo hasta 5 ya que conforme aumentaba el valor de k las discrepancias entre grupos aumentaba. En el caso de k=9 además, no era efectivo, ya que solo disponíamos de 11 *benchmarks*, con lo que la clasificación que se obtenía englobaba casi a un *benchmark* por grupo.

Un inconveniente encontrado en esta rama de investigación es que, como ha quedado demostrado, este sistema no es válido para los *benchmarks* de punto flotante, no consiguiendo el objetivo deseado. Esta línea de trabajo sigue abierta y se esperan tener resultados en un breve periodo de tiempo, quedando pendiente un estudio mucho más exhaustivo para este caso en particular, incluyendo otro tipo de medidas o pudiendo realizar las clasificaciones con algún otro algoritmo de *clustering* buscando la optimización de un método útil para este tipo de *benchmarks*.

Como líneas de trabajo futuro se propone incluir más programas de prueba para evaluar los grupos formados o intentar clasificar los *benchmarks* con otros tipos de algoritmos de *clusterings* como el *jerárquico* [19](crea una jerarquía de árbol con similitudes entre los vectores, denominada *dendrogram*), el *Fuzzy C-means* [19, 11](variación del *k-means clustering* donde todos los vectores tienen un grado de pertenencia a cada cluster, y los centroides respectivos se calculan en base a ese grado) o el de *mapas auto-organizativos* [36, 18](aplicando *self organizing maps* (SOM) a los datos, los cluster pueden ser definidos por puntos definidos en un grid ajustado a los datos. Para el caso de *clustering* el algoritmo hace uso de un grid de una dimensión). Otra rama de investigación sería utilizar otras arquitecturas para obtener y comparar los valores de las medidas mencionadas en este trabajo.

Referencias

[1] Cfp2006: http://www.spec.org/cpu2006/cfp2006/.

[2] Cint2006: http://www.spec.org/cpu2006/cint2006/.

[3] Vattani A. k-means requires exponentially many iterations even in the plane. *Proceedings of the 25th Symposium on Computational Geometry (SoCG)*, 2009.

[4] Hansen P. Aloise D., Deshpande A. and P. Popat. Np-hardness of euclidean sum-of-squares clustering. *Machine Learning*, 2009.

[5] Manthey B. Arthur D. and Roeglin H. k-means has polynomial smoothed complexity. *Proceedings of the 50th Symposium on Foundations of Computer Science (FOCS)*, 2009.

[6] MacQueen J. B. Some Methods for classification and Analysis of Multivariate Observations. *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability. University of California*, 1967.

[7] Michela Becchi and Patrick Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In *Proc. of Computing Frontiers '06*, pages 29–40, 2006.

[8] E. Berg and E. Hagersten. StatCache: a Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *ISPASS'04*.

[9] Arthur D. and Vassilvitskii S. How slow is the k-means method? *Proceedings of the 22nd Symposium on Computational Geometry (SoCG)*, 2006.

[10] Johan de Galas. The Quest for More Processing Power: is the Single Core CPU Doomed? *http://www.anandtech.com/cpuchipsets/showdoc.aspx?i=2377*, February 2005.

[11] Hart PE Duda RO and Stork DG. Pattern classification and scene analysis. *New York: John Wiley & Sons*, 2002.

[12] Alexandra Fedorova, Juan Carlos Saez, Daniel Shelepov, and Manuel Prieto. Maximizing Power Efficiency with Asymmetric Multicore Systems. *Commun. ACM*, 52(12):48–57, 2009.

[13] Vincent W. Freeh, David K. Lowenthal, Feng Pan, Nandini Kappiah, Rob Springer, and Barry L. Rountree. Analyzing the energy-time trade-off in high-performance computing applications. *IEEE Trans. Parallel Distrib. Syst.*, 18(6):835–848, 2007. Member-Femal,, Mark E.

[14] Michael Gschwind. The cell broadband engine: exploiting multiple levels of parallelism in a chip multiprocessor. *Int. J. Parallel Program.*, 35(3):233–262, 2007.

[15] Steinhaus H. Sur la division des corps matériels en parties. *Bull. Acad. Polon. Sci.*, 1956.

[16] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, 41(7):33–38, 2008.

[17] KatohÑ. Inaba M. and Imai H. Applications of weighted voronoi diagrams and randomization to variance-based k-clustering. *Proceedings of 10th ACM Symposium on Computational Geometry*, 1994.

[18] Wang J, Smeland E Delabie J, Aasheim HC, and Myklebost O. Clustering of the som easily reveals distinct gene expression patterns: results of a reanalysis of lymphoma study. *BMC Bioinformatics. 2002;3:36*, 2002.

[19] Murty MN Jain AK and Flynn PJ. Data clustering: a review. *ACM Comput. Surveys, 31:264–323*, 1999.

[20] D. Jayatilaka and D. J. Grimwood. Tonto: A fortran based object-oriented system for quantum chemistry and crystallography. *Computational Science — ICCS. Volume 2660/2003, 707, DOI: 10.1007/3-540-44864-0_15*, 2003.

[21] Rakesh Kumar, Keith I. Farkas, and Norman Jouppi et al. Single-ISA Heterogeneous Multi-Core Architectures: the Potential for Processor Power Reduction. In *Proc. of MICRO 36*, 2003.

[22] Rakesh Kumar, Dean M. Tullsen, and Parthasarathy Ranganathan et al. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proc. of ISCA '04*, 2004.

[23] Tong Li, Dan Baumberger, and David A. Koufaty et al. Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures. In *Proc. of SC '07*, pages 1–11.

[24] Tong Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating System Support for Overlapping-ISA Heterogeneous Multi-Core Architectures. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1 –12, 9-14 2010.

[25] Nimbhorkar P. Mahajan M. and Varadarajan K. The planar k-means problem is np-hard. *Lecture Notes in Computer Science*, 2009.

[26] Terje Mathisen. Pentium secrets. *BYTE*, 1994.

[27] Jeffrey C. Mogul, Jayaram Mudigonda, Nathan Binkert, Parthasarathy Ranganathan, and Vanish Talwar. Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems. *IEEE Micro*, 28(3):26–41, 2008.

[28] Lloyd S. P. Least square quantization in pcm. *Bell Telephone Laboratories Paper*, 1957.

[29] Lloyd S. P. Least squares quantization in pcm. *Published in journal IEEE Transactions on Information Theory*, 1982.

[30] d'Humieres D. Qian Y.H. and Lallemand P. Lattice bgk models for navier-stokes equation. *Europhys. Lett. 17(6): 479-484*, 1992.

[31] Dasgupta S. and Freund Y. Random projection trees for vector quantization. *Information Theory, IEEE Transactions*, 2009.

[32] Juan Carlos Saez, Alexandra Fedorova, Manuel Prieto, and Hugo Vegas. Operating System Support for Mitigating Software Scalability Bottlenecks on Asymmetric Multicore Processors. In *Proc. of ACM Computing Frontiers '10*, 2010.

[33] Juan Carlos Saez, Daniel Shelepov, Alexandra Fedorova, and Manuel Prieto. Leveraging Workload Diversity through OS Scheduling to Maximize Performance on Single-ISA Heterogeneous Multicore Systems. *To be published in Journal of Parallel and Distributed Computing*, 2010.

[34] D. Shelepov, J.C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z.F. Huang, S. Blagodurov, and V. Kumar. HASS: A Scheduler for Heterogeneous Multicore Systems. *SIGOPS Oper. Syst. Rev.*, 43(2):66–75, 2009.

[35] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The stampede approach to thread-level speculation. *ACM TOCS*, 23(3):253–300, 2005.

[36] Wong G Toronen P, Kolehmainen M and Castren E. Analysis of gene expression data using self-organizing maps. *FEBS Lett. 1999;451:142–146*, 1999.

[37] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing Cache Contention in Multicore Processors Via Scheduling. In *ASPLOS'10*, 2010.